MODULE IV

ANDROID MENUS

The key class in Android menu support is *android.view.Menu.* Every activity in Android is associated with one menu object of this type. The menu object then contains a number of menu items and submenus. Menu items are represented by *android.view.MenuItem.* Submenus are represented by *android.view.SubMenu.*

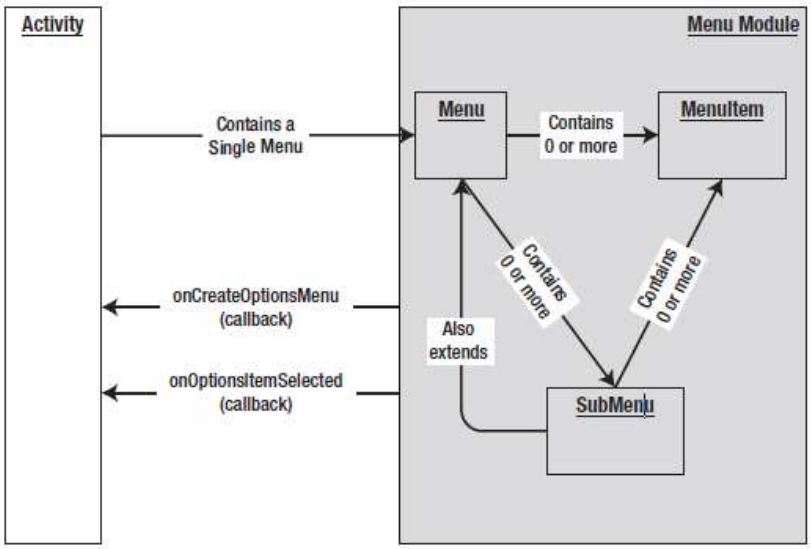

Figure 7–1. *Structure of Android menu-related classes*

A Menu object contains a set of menu items. A menu item carries the following attributes:

- *Name*: A string title
- *Menu item ID*: An integer
- *Group ID*: An integer representing which group this item should be part of
- *Sort order*: An integer identifying the order of this menu item when it is displayed in the menu.

The name and menu item ID attributes are self explanatory. We can group menu items together by assigning each one a group ID. Multiple menu items that carry the same group ID are considered part of the same group. The sort-order attribute demands a bit of coverage. If one menu item carries an order number of 4 and another menu item carries a order number of 6, the first menu item will appear above the second menu item in the menu. Some of these menu item sort-ordernumber ranges are reserved for certain kinds of menus. These are called menu categories. The available menu categories are as follows:

- *Secondary*: Secondary menu items, which are considered less important
- *System*: This sort-order range is reserved for menu items added by the Android system.
- *Alternative*: They're usually contributed by external applications that provide alternative ways to deal with the data that is under consideration.
- *Container*: In Android, the parents of views, such as layouts, are considered containers. The documentation is not clear about whether this

category pertains to layouts, but it is as good a guess as any. Most likely, container-related menu items can be placed in this range.

CREATING MENUS

In the Android SDK, we don't need to create a menu object from scratch. Because an activity is associated with a single menu, Android creates this single menu for that activity and passes it to the *onCreateOptionsMenu()* callback method of the activity class. (As the name of the method indicates, menus in Android are also known as options menus). Starting with 3.0, this method is called as part of activity creation. This change is due to the fact that the action bar is always present in an activity. A menu item that we create in this method for the options menu may sit in an action bar. Because an action bar is always visible (unlike the options menu), the action bar must know its menu items from the beginning. So Android cannot wait until the user opens an options menu to call the *onCreateOptionsMenu()* method. This callback menu setup method allows us to populate the single passed-in menu with a set of menu items (see Listing 7–1).

Listing 7–1. Signature for the onCreateOptionsMenu Method

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
        // populate menu items
         ....
                                                    ...return true; }
```

Once the menu items are populated, the code should return true to make the menu visible. If this method returns false, the menu is invisible. The code in Listing 7–2 shows how to add three menu items using a single group ID along with incremental menu item IDs and order IDs.

Listing 7–2. Adding Menu Items

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //call the base class to include system menus
    super.onCreateOptionsMenu(menu); menu.add(0 // Group
        ,1 // item id
        ,0 //order
        ,"append"); // title
    menu.add(0,2,1,"item2");
    menu.add(0,3,2,"clear");
    //It is important to return true to see the menu return true;
}
```

WORKING WITH MENU GROUPS

Following code shows how to work with menu groups. Using Group IDs to Create Menu Groups

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
//Group 1
int group1 = 1;
menu.add(group1,1,1,"g1.item1");
menu.add(group1,2,2,"g1.item2");
//Group 2
int group2 = 2;
menu.add(group2,3,3,"g2.item1");
menu.add(group2,4,4,"g2.item2");
return true; // it is important to return true
}
```

Notice how the menu item IDs and the order IDs are independent of the groups. Android provides a set of methods on the *android.view.Menu* class that are based on group IDs. We can manipulate a group's menu items using these methods:

- removeGroup(id):- removes all menu items from that group, given the group ID.
- setGroupCheckable(id, checkable, exclusive):- We can use this method to show a check mark on a menu item when that menu item is selected.
- setGroupEnabled(id,boolean enabled):- We can enable or disable menu items in a given group
- setGroupVisible(id,visible):- we can control the visibility of a group of menu items

RESPONDING TO MENU ITEMS

There are multiple ways of responding to menu item clicks in Android. We can use the *onOptionsItemSelected()* method of the activity class; we can use stand-alone listeners, or we can use intents.

a) Through onOptionsItemSelected()

When a menu item is clicked, Android calls the onOptionsItemSelected() callback method on the Activity class.

Listing 7–4. Signature and Body of the onOptionsItemSelected Method

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
     ....
//for items handled
return true;
//for the rest
...return super.onOptionsItemSelected(item); } }
```

The key pattern here is to examine the menu item ID through the *getItemId*() method of the *MenuItem* class and do what's necessary. If *onOptionsItemSelected*() handles a menu item, it returns true. The menu event will not be further propagated. For the menu item callbacks that *onOptionsItemSelected*() doesn't deal with, *onOptionsItemSelected*() should call the parent method through super.*onOptionsItemSelected*(). The default implementation of the *onOptionsItemSelected*() method returns false so that the normal processing can take place. Normal processing includes alternative means of invoking responses for a menu.

b) Through Listeners

A listener implies object creation and a registry of the listener. So this is the overhead that the performance refers to in the first sentence of this paragraph. However, we may choose to give more importance to reuse and clarity, in which case listeners provide flexibility. This approach is a two-step process. In the first step, we implement the *OnMenuClickListener* interface. Then, we take an instance of this implementation and pass it to the menu item. When the menu item is clicked, the menu item calls the *onMenuItemClick*() method of the *OnMenuClickListener* interface.

Listing 7–5. Using a Listener as a Callback for a Menu Item Click

```
//Step 1
public class MyResponse implements OnMenuClickListener
{
        //some local variable to work on
        //...
        //Some constructors
        @override
        boolean onMenuItemClick(MenuItem item)
        {
        //do our thing
        return true;
        }
}
//Step 2
MyResponse myResponse = new MyResponse(...);
menuItem.setOnMenuItemClickListener(myResponse);
...
```

The *onMenuItemClick*() method is called when the menu item has been invoked. This code executes as soon as the menu item is clicked, even before the *onOptionsItemSelected*() method is called. If *onMenuItemClick*() returns true, no other callbacks are executed—including the *onOptionsItemSelected*() callback method. This means that the listener code takes precedence over the *onOptionsItemSelected*() method.

c) Using Intent

We can also associate a menu item with an intent by using the *MenuItem's* method *setIntent(intent)*. By default, a menu item has no intent associated with it. But when an intent is associated with a menu item, and nothing else handles the menu item, then the default behavior is to invoke the intent using *startActivity*(*intent*).

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 *SubMenu* object 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.

Listing 7–7. Adding Submenus

```
private void addSubMenu(Menu menu)
{
    //Secondary items are shown just like everything else
    int base=Menu.FIRST + 100;
    SubMenu sm =
    menu.addSubMenu(base,base+1,Menu.NONE,"submenu");
    sm.add(base,base+2,base+2,"sub item1");
    sm.add(base,base+3,base+3,"sub item2");
    sm.add(base,base+4,base+4,"sub item3");
    //submenu item icons are not supported
    item1.setIcon(R.drawable.icon48x48_2);
```

```
        //the following is ok however
        sm.setIcon(R.drawable.icon48x48_1);
        //This will result in runtime exception
        //sm.addSubMenu("try this");

    }
```

NOTE: *SubMenu*, as a subclass of the Menu object, continues to carry the *addSubMenu*() method. The compiler won't complain if we add a submenu to another submenu, but we'll get a runtime exception if we try to do it.

The Android SDK documentation also suggests that submenus do not support icon menu items. When we add an icon to a menu item and then add that menu item to a submenu, the menu item ignores that icon, even if we don't see a compile-time or runtime error. However, the submenu itself can have an icon.

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, depending 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 the long click.

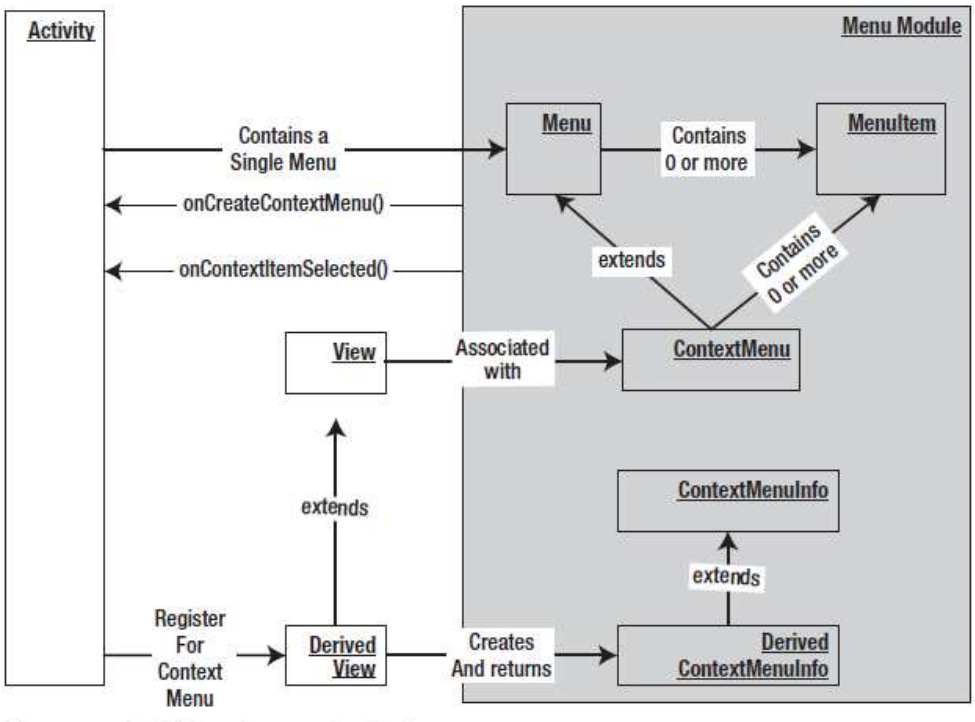

Figure 7–3. *Activities, views, and context menus*

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 *invalidateOptionsMenu*(), which in turn invokes the *onPrepareOptionsMenu*(). We can call this method any time something changes in our application state that would require a change to the menu.

LOADING MENU THROUGH XML

We have created all our menus programmatically. This is not the most convenient way to create menus, because for every menu, we have to provide several IDs and define constants for each of those IDs. No doubt this is tedious. Instead, we can define menus through XML files, which is possible in Android because menus are also resources. The XML approach to menu creation offers several advantages, such as

- he ability to name menus
- Order them automatically
- Give them IDs
- We can also get localization support for the menu text.

Follow these steps to work with XML-based menus:

1. Define an XML file with menu tags.
2. Place the file in the /res/menu subdirectory. The name of the file is arbitrary, and we can have as many files as we want. Android automatically generates a resource ID for this menu file.
3. Use the resource ID for the menu file to load the XML file into the menu.
4. Respond to the menu items using the resource IDs generated for each menu item.

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 a

button click or a click on an image view. Figure 7–4 shows a pop-up menu invoked against a view.

Figure 7–4. *Pop-up menu attached to a text view*

To create a pop-up menu like the one in Figure 7–4, start with a regular XML menu file as shown in Listing 7–18.

Listing 7–18. A Sample XML File for a Pop-up Menu

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- This group uses the default category. -->
    <group android:id="@+id/menuGroup_Popup">
        <item android:id="@+id/popup_menu_1"
            android:title="Menu 1" />
        <item android:id="@+id/popup_menu_2"
            android:title="Menu 2" />
    </group>
</menu>
```

Assuming the code in Listing 7–18 is in a file called popup_menu.xml. As we can see, a pop-up menu behaves much like an options menu. The key differences are as follows:

- A pop-up menu is used on demand, whereas an options menu is always available.
- A pop-up menu is anchored to a view, whereas an options menu belong to the entire activity.
- A pop-up menu uses its own menu item callback, whereas the options menu uses the onOptionsItemSelected() callback on the activity.

FRAGMENTS IN ANDROID

A fragment is a piece of activity which enable more modular activity design. It will not be wrong if we say a fragment is a kind of sub-activity. A fragment has its own layout and behavior with life cycle. We can add or remove fragments in an activity while the activity is running. We can combine multiple fragments in a single activity to build a multi-plane UI. A fragment can be used in multiple activities. A fragment can implement a behavior that has no user

interface component. Fragments were added to the android version HoneyComb with API 11.0

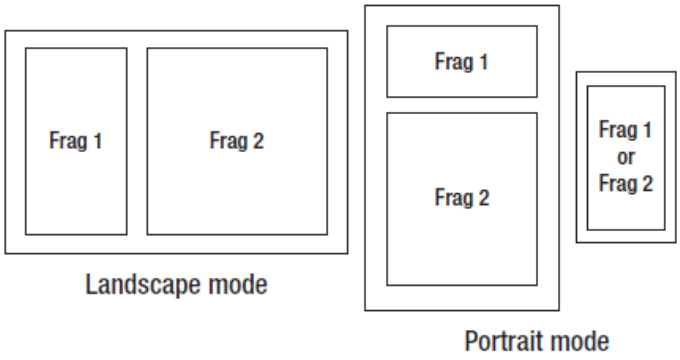

Figure 8–1. *Fragments used for a tablet UI and for a smartphone UI*

STRUCTURE OF FRAGMENT

A fragment is like a sub-activity: it has a fairly specific purpose and almost always displays a user interface. But where an activity is sub-classed below Context, a fragment is extended from Object in package android.app. A fragment is not an extension of Activity. A fragment can have a view hierarchy to engage with a user. It can be created (inflated) from an XML layout specification or created in code. A fragment has a bundle that serves as its initialization arguments.

FRAGMENT LIFE CYCLE

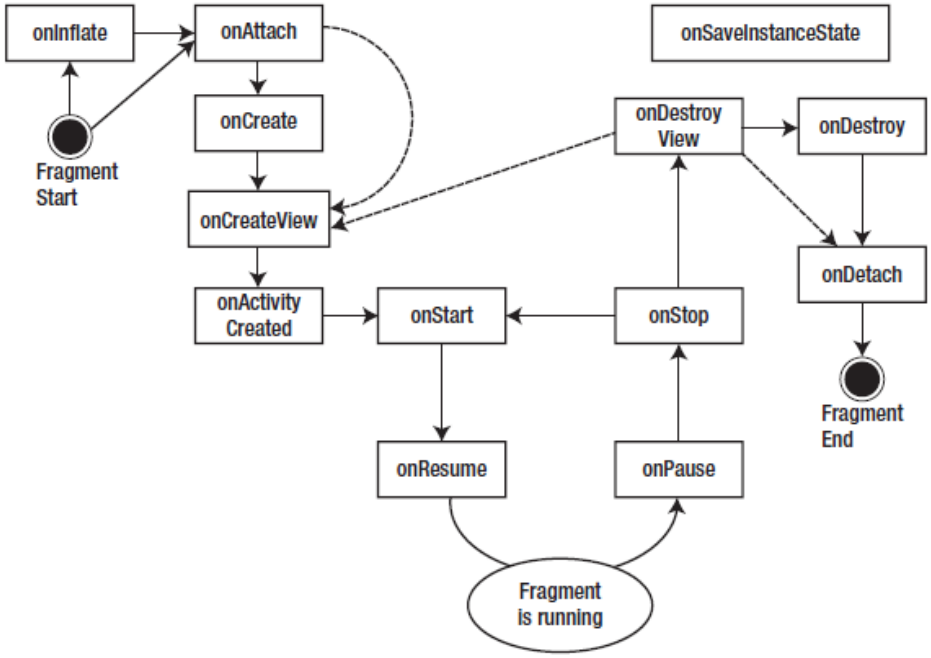

Figure 8–2. *Lifecycle of a fragment*

A fragment is very dependent on the activity in which it lives and can go through multiple steps while its activity goes through one.

FRAGMENT MANAGER

The FragmentManager is a component that takes care of the fragments belonging to an activity. This includes fragments on the back stack and fragments that may just be hanging around. Fragments should only be created within the context of an activity. The FragmentManager class is used to access and manage these fragments for an activity. Besides getting a fragment transaction, we can also get a fragment using the fragment's ID, its tag, or a combination of bundle and key. For this, the getter methods include findFragmentById(), findFragmentByTag(), and getFragment(). The getFragment() method would be used in conjunction with putFragment(), which also takes a bundle, a key, and the fragment to be put.

The bundle is most likely going to be the savedState bundle, and putFragment() will be used in the onSaveInstanceState() callback to save the state of the current activity (or another fragment). The getFragment() method would probably be called in onCreate() to correspond to putFragment(), although for a fragment, the bundle is available to the other callback methods, as described earlier. Obviously, we can't use the getFragmentManager() method on a fragment that has not been attached to an activity yet. But it's also true that we can attach a fragment to an activity without making it visible to the user yet. If we do this, we should associate a String tag to the fragment so we can get to it in the future. We'd most likely use this method of FragmentTransaction to do this:

public FragmentTransaction add (Fragment fragment, String tag)

The fragment back stack is also the domain of the fragment manager. Whereas a fragment transaction is used to put fragments onto the back stack, the fragment manager can take fragments off the back stack. This is usually done using the fragment's ID or tag, but it can be done based on position in the back stack or just to pop the top-most fragment. Finally, the fragment manager has methods for some debugging features, such as turning on debugging messages to LogCat using enableDebugLogging() or dumping the current state of the fragment manager to a stream using dump().

SAVING FRAGMENT STATE

Another interesting class was introduced in Android 3.2: Fragment.SavedState. Using the saveFragmentInstanceState() method of FragmentManager, we can pass this method a fragment, and it returns an object

representing the state of that fragment. We can then use that object when initializing a fragment, using Fragment's setInitialSavedState() method.

PERSISTENCE OF FRAGMENTS

When we play with this sample application, make sure we rotate the device (pressing Ctrl+F11 rotates the device in the emulator). We will see that the device rotates, and the fragments rotate right along with it. If we watch the LogCat messages, we will see a lot of them for this application. During a device rotation, pay careful attention to the messages about fragments; not only does the activity get destroyed and recreated, but the fragments do also.

So far, we only wrote a tiny bit of code on the titles fragment to remember the current position in the titles list across restarts. We didn't do anything in the details fragment code to handle reconfigurations, and that's because we didn't need to. Android will take care of hanging onto the fragments that are in the fragment manager, saving them away, and then restoring them when the activity is being re-created. We should realize that the fragments we get back after the reconfiguration is complete are very likely not the same fragments in memory that we had before. These fragments have been reconstructed for we. Android saved the arguments bundle and the knowledge of which type of fragment it was, and it stored the saved-state bundles for each fragment that contain saved-state information about the fragment to use to restore it on the other side.

COMMUNICATIONS WITH FRAGMENTS

The fragment manager knows about all fragments attached to the current activity, the activity or any fragment in that activity can ask for any other fragment using the getter methods described earlier. Once the fragment reference has been obtained, the activity or fragment could cast the reference appropriately and then call methods directly on that activity or fragment. This would cause our fragments to have more knowledge about the other fragments than might normally be desired, but don't forget that we're running this application on a mobile device, so cutting corners can sometimes be justified.
A code snippet is provided in Listing 8–12 to show how one fragment might communicate directly with another fragment.
Listing 8–12. Direct Fragment-to-Fragment Communication

```
FragmentOther fragOther =
(FragmentOther)getFragmentManager().findFragmentByTag("other");
fragOther.callCustomMethod( arg1, arg2 );
```

In Listing 8–12, the current fragment has direct knowledge of the class of the other fragment and also which methods exist on that class. This may be okay because these fragments are part of one application, and it can be easier to simply accept the fact that some fragments will know about other fragments.

STARTACTIVITY() AND SETTARGETFRAGMENT()

A feature of fragments that is very much like activities is the ability of a fragment to stat an activity. Fragment has a startActivity() method and startActivityForResult() method. These work just like the ones for activities; when a result is passed back, it will cause the onActivityResult() callback to fire on the fragment that started the activity. There's another communication mechanism we should know about. When one fragment wants to start another fragment, there is a feature that lets the calling fragment set its identity with the called fragment. The following example of what it might look like.

Listing 8–13. Fragment-to-Target-Fragment Setup

```
mCalledFragment = new CalledFragment();
mCalledFragment.setTargetFragment(this, 0);
fm.beginTransaction().add(mCalledFragment, "work").commit();
```

With these few lines, we've created a new CalledFragment object, set the target fragment on the called fragment to the current fragment, and added the called fragment to the fragment manager and activity using a fragment transaction. When the called fragment starts to run, it will be able to call getTargetFragment(), which will return a reference to the calling fragment. With this reference, the called fragment could invoke methods on the calling fragment or even access view components directly. For example, the called fragment could set text in the UI of the calling fragment directly.

Listing 8–14. Target Fragment-to-Fragment Communication

```
TextView tv = (TextView)
getTargetFragment().getView().findViewById(R.id.text1);
tv.setText("Set from the called fragment");
```

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.

WORKING WITH TOAST

The alert messages are commonly used for debugging JavaScript on error pages. If we are pressed to use a similar approach for infrequent debug messages, we can use the Toast object in Android. A Toast is like an alert dialog that has a message and displays for a certain amount of time and then goes

away. It does not have any buttons. So it can be said that it is a transient alert message. It's called Toast because it pops up like toast out of a toaster. The following example shows an example of how we can show a message using Toast.

Listing 9–10. Using Toast for Debugging

```
//Create a function to wrap a message as a toast
//show the toast
public void reportToast(String message)
{
String s = MainActivity.LOGTAG + ":" + message;
Toast.makeText(activity, s, Toast.LENGTH_SHORT).show();
}
```

The makeText() method in Listing 9–10 can take not only an activity but any context object, such as the one passed to a broadcast receiver or a service, for example. This extends the use of Toast outside of activities.

IMPLEMENTING ACTION BAR

ActionBar was introduced in the Android 3.0 SDK for tablets and is now available for phones as well in 4.0. It allows we to customize the title bar of an activity. Prior to the 3.0 SDK release, the title bar of an activity merely contained the title of an activity. Android ActionBar is modeled similar to the menu/title bar of a web browser. An action bar is owned by an activity and follows its lifecycle. An action bar can take one of three forms: tabbed action bar, list action bar, or standard action bar. We see how these various action bars look and behave in each of the modes.

Home Icon area: The icon at upper left on the action bar is sometimes called the Home icon. This is similar to a web site navigation context, where clicking the Home icon takes we to a starting point. When we transfer the user to the home activity, don't start a new home activity; instead, transfer to it by using an intent flag that clears the stack of all activities on top of the home activity. We see later that clicking this Home icon sends a callback to the option menu with menu ID android.R.id.home.

- Title area: The Title area displays the title for the action bar.
- Tabs area: The Tabs area is where the action bar paints the list of tabs specified. The content of this area is variable. If the action bar navigation mode is tabs, then tabs are shown here. If the mode is listnavigation mode, then a navigable list of drop-down items is shown. In standard mode, this area is ignored and left empty.
- Action Icon area: Following the Tabs area, the Action Icon area shows some of the option menu items as icons. We see how to choose which option menus are displayed as action icons in the example later.
- Menu Icon area: Last is the Menu Icon area. It is a single standard menu icon. When we click this menu icon, we see the expanded menu. This expanded menu looks different or shows up in a different location

depending on the size of the Android device. We can also attach a search view as if it is an action icon of the menu.

Figure 10–1 shows a typical action bar in tabbed navigation mode.

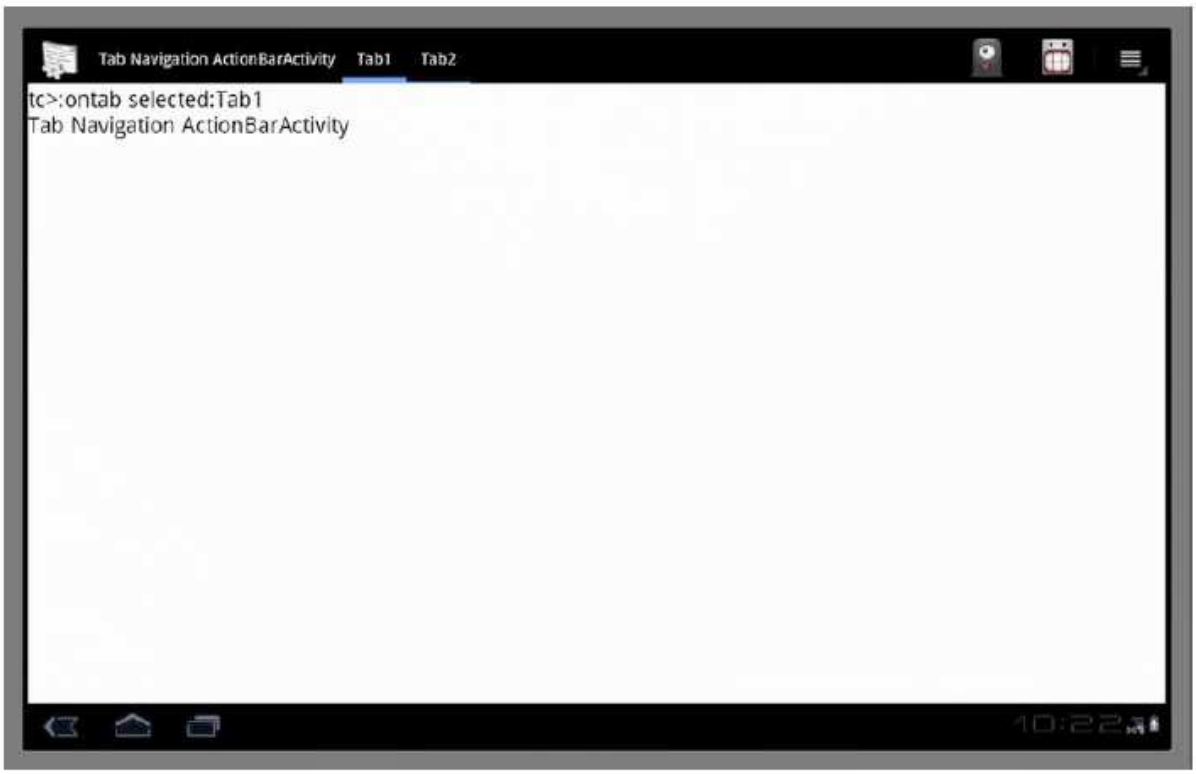

Figure 10–1. *An activity with a tabbed action bar*

TABBED NAVIGATION ACTION BAR ACTIVITY

Each of the action bar divided into separate tabs. Each tab contains their own activities. The common behavior in a base class and allow each of the derived activities, including this tabbed action bar activity, to configure the action bar. The difficult to explain these common files without the context of at least one action bar activity. Following is a list of files that are needed for this tabbed action bar:

- DebugActivity.java: Base class activity that allows for a debug text view as shown in Figure 10-1 (Listing 10-2)
- BaseActionBarActivity.java: Derived from DebugActivity and allows for common navigation (such as responding to common actions including switching between the three activities) (Listing 10-3)
- IReportBack.java: An interface that works as a communication vehicle between the debug activity and the various listeners of the action bar (Listing 10-1)
- BaseListener.java: Base listener class that works with the DebugActivity and the various actions that gets invoked from the action bar. Acts as a base class for both tab listeners and list navigation listeners (Listing 10-4)
- TabNavigationActionBarActivity.java: inherits from BaseActionBarActivity.java and configures the action bar as a tabbed

action bar. Most of the code pertaining to the tabbed action bar is in this class (Listing 10-6)

- TabListener.java: Required to add a tab to the tabbed action bar. This where we respond to tab clicks. In our case this simply logs a message to the debug view through the BaseListener (Listing 10-5)
- AndroidManifest.xml: Where activities are defined to be invoked (Listing 10-13)
- Layout/main.xml: Layout file for the DebugActivity. Because all the three status bar activities inherit this base DebugActivity, they all share this layout file (Listing 10-7)
- menu/menu.xml: A set of menu items to test the menu interaction with the action bar. The menu file is also shared across all the derived status bar activities (Listing 10-9)

IMPLEMENTING BASE ACTIVITY CLASSES

A number of the base classes use the IReportBack interface. It is introduced in

Listing 10–1. IReportBack.java

```
//IReportBack.java
package com.androidbook.actionbar;
public interface IReportBack
{
        public void reportBack(String tag, String message);
        public void reportTransient(String tag, String message);
}
```

A class that implements this interface takes a message and reports it on a screen, like a debug message. This is done through the reportBack() method. The method reportTransient() does the same thing except it uses a Toast to report that message to the user. In this example, the class that implements IReportBack is DebugActivity. This allows DebugActivity to pass itself around without exposing all of its internals. The source code for DebugActivity is presented in

Listing 10–2. DebugActivity with a Debug Text View

```
//DebugActivity.java
package com.androidbook.actionbar;
//
//Use CTRL-SHIFT-O to import dependencies
//
public abstract class DebugActivity extends Activity implements IReportBack
{
//Derived classes needs first protected abstract boolean
onMenuItemSelected(MenuItem item);
//private variables set by constructor
private static String tag=null;
private int menuId = 0;
private int layoutid = 0;
private int debugTextViewId = 0;
```

```
public DebugActivity(int inMenuId, int inLayoutId, int inDebugTextViewId,
String inTag)
{
tag = inTag;
menuId = inMenuId;
layoutid = inLayoutId;
debugTextViewId = inDebugTextViewId;
}
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(this.layoutid);
//We need the following to be able to scroll
//the text view.
TextView tv = this.getTextView();
tv.setMovementMethod(
ScrollingMovementMethod.getInstance());
}
@Override
public boolean onCreateOptionsMenu(Menu menu){
super.onCreateOptionsMenu(menu);
MenuInflater inflater = getMenuInflater();
inflater.inflate(menuId, menu);
return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item){
appendMenuItemText(item);
if (item.getItemId() == R.id.menu_da_clear){
this.emptyText();
return true;
}
boolean b = onMenuItemSelected(item);
if (b == true)
{
return true;
}
return super.onOptionsItemSelected(item);
}
protected TextView getTextView(){
return
(TextView)this.findViewById(this.debugTextViewId);
}
protected void appendMenuItemText(MenuItem menuItem){
String title = menuItem.getTitle().toString();
appendText("MenuItem:" + title);
}
```

```
protected void emptyText(){
TextView tv = getTextView();
tv.setText("");
}
protected void appendText(String s){
TextView tv = getTextView();
tv.setText(s + "\n" + tv.getText());
Log.d(tag,s);
}
public void reportBack(String tag, String message)
{
this.appendText(tag + ":" + message);
Log.d(tag,message);
}
public void reportTransient(String tag, String message)
{
String s = tag + ":" + message;
Toast mToast =
Toast.makeText(this, s, Toast.LENGTH_SHORT);
mToast.show();
reportBack(tag,message);
Log.d(tag,message);
}
}//eof-class
```

The primary goal of this base activity class is to present an activity with a debug text view in it. This text view is used to log messages coming from the reportBack() method. We use this activity as the base activity for all of the action bar activities.

TABBED LISTENER

Before we are able to work with a tabbed action bar, we need a tabbed listener. A tabbed listener allows we to respond to the click events on the tabs. We derive our tabbed listener from a base listener that allows we to log tab actions. Listing 10–4 shows the base listener that uses the IReportBack for logging.

Listing 10–4. A Common Listener for Action Bar Enabled Activities

```
//BaseListener.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class BaseListener
{
protected IReportBack mReportTo;
protected Context mContext;
public BaseListener(Context ctx, IReportBack target)
{
mReportTo = target;
mContext = ctx;  }   }
```

This base class holds a reference to an implementation of IReportBack and also the activity that can be used as a context. This tabbed listener documents the callbacks from the action bar tabs to the debug text. In this case, the DebugActivity from Listing 10–2 is the implementer of IReportBack and also plays the role of the context. Now that we have a base listener, Listing 10–5 shows the tabbed listener.

Listing 10–5. Tab Listener to Respond to Tab Actions

```
// TabListener.java
package com.androidbook.actionbar;
//
//Use CTRL-SHIFT-O to import dependencies
//
public class TabListener extends BaseListener
implements ActionBar.TabListener
{
private static String tag = "tc>";
public TabListener(Context ctx,
IReportBack target)
{
super(ctx, target);
}
public void onTabReselected(Tab tab,
FragmentTransaction ft)
{
this.mReportTo.reportBack(tag,
"ontab re selected:" + tab.getText());
}
public void onTabSelected(Tab tab,
FragmentTransaction ft)
{
this.mReportTo.reportBack(tag,
"ontab selected:" + tab.getText());
}
public void onTabUnselected(Tab tab,
FragmentTransaction ft)
{
this.mReportTo.reportBack(tag,
"ontab un selected:" + tab.getText());
}
}
```

This tabbed listener documents the callbacks from the action bar tabs to the debug text view of Figure 10–1.

TABBED ACTION BAR

With the tabbed listener in place, we can finally construct the tabbed navigation activity. This is presented in Listing 10–6.

Listing 10–6. Tab-Navigation Enabled Action Bar Activity

```
// TabNavigationActionBarActivity.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class TabNavigationActionBarActivity
extends BaseActionBarActivity
{
private static String tag =
"Tab Navigation ActionBarActivity";
public TabNavigationActionBarActivity()
{
super(tag);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
workwithTabbedActionBar();
}
public void workwithTabbedActionBar()
{
ActionBar bar = this.getActionBar();
bar.setTitle(tag);
bar.setNavigationMode(
ActionBar.NAVIGATION_MODE_TABS);
TabListener tl = new TabListener(this,this);
Tab tab1 = bar.newTab();
tab1.setText("Tab1");
tab1.setTabListener(tl);
bar.addTab(tab1);
Tab tab2 = bar.newTab();
tab2.setText("Tab2");
tab2.setTabListener(tl);
bar.addTab(tab2);
}
}//eof-class
```

We now look at the code for this activity (Listing 10–6) in the following subsections, which draw attention to each aspect of working with a tabbed action bar. We start with getting access to the action bar belonging to an activity.

DEBUG TEXT VIEW LAYOUT

As the tabs of the action bar are clicked, the tab listeners are set up in such a way that debug messages are sent to the debug text view. Listing 10–7 shows the layout file for the DebugActivity, which in turn contains the debug text view.

Listing 10–7. Debug Activity Text View Layout File

```
<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/main.xml -->
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:gravity="fill"
        >
<TextView android:id="@+id/textViewId"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="@android:color/white"
        android:text="Initial Text Message"
        android:textColor="@android:color/black"
        android:textSize="25sp"

        android:scrollbars="vertical"
        android:scrollbarStyle="insideOverlay"
        android:scrollbarSize="25dip"
        android:scrollbarFadeDuration="0"
        />
</LinearLayout>
```

There are a few things worth noting about this layout. We set the background color of the text view to white. This lets we capture screens in brighter light. The text size is also set to a large font to aid screen capture. We also set up the text view so that it is enabled for scrolling. Although typically layouts use ScrollView, a text view is already enabled for scrolling by itself. In addition to enabling the scrolling properties in the XML file for the text view, we need to call the setMovementMethod() method on the text view as shown in **Listing 10–8**. Enabling Text View for Scrolling

```
TextView tv = this.getTextView();
tv.setMovementMethod(
ScrollingMovementMethod.getInstance());
```

This code is extracted from the DebugActivity (Listing 10–2).

As the text view is scrolled, notice that the scrollbar appears and then fades away. This is not a good indicator if there is text beyond visible range. We can tell the scrollbar to stay by setting the fade duration to 0. See Listing 10–7 for how to set this parameter.

ACTION BAR AND MENU INTERACTION

This example also demonstrates how menus interact with the action bar. So, we need to set up a menu file. This file is presented in Listing 10–9.

Listing 10–9. Menu XML File for This Project

```
<!-- /res/menu/menu.xml -->
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
<!-- This group uses the default category. -->
<group android:id="@+id/menuGroup_Main">
<item android:id="@+id/menu_action_icon1"
      android:title="Action Icon1"
      android:icon="@drawable/creep001"
      android:showAsAction="ifRoom"/>
<item android:id="@+id/menu_action_icon2"
      android:title="Action Icon2"
      android:icon="@drawable/creep002"
      android:showAsAction="ifRoom"/>
<item android:id="@+id/menu_icon_test"
      android:title="Icon Test"
      android:icon="@drawable/creep003"/>
<item android:id="@+id/menu_invoke_listnav"
      android:title="Invoke List Nav"
      />
<item android:id="@+id/menu_invoke_standardnav"
      android:title="Invoke Standard Nav"
      />
<item android:id="@+id/menu_invoke_tabnav"
      android:title="Invoke Tab Nav"
      />
<item android:id="@+id/menu_da_clear"
      android:title="clear" />
</group>
</menu>
```

Figure 10–2. *An activity with a tabbed action bar and expanded menu*

LIST NAVIGATION ACTION BAR ACTIVITY

Because our base classes are carrying the most of the work, it is fairly easy to implement and test the list action bar navigation activity. We need the following additional files to implement this activity:

- SimpleSpinnerArrayAdapter.java: Needed to set up the list navigation bar along with the listener. This class provides the rows required by a drop-down navigation list (Listing 10–12).
- ListListener.java: Acts as a listener to the list navigation activity. This class needs to be passed to the action bar when setting it up as a list action bar (Listing 10–13).
- ListNavigationActionBarActivity.java: Implements the list navigation action bar activity (Listing 10–14).

Once we have these three new files, we need to update the following two files:

- BaseActionBarActivity.java: Uncomment the invocation of the list action bar activity (Listing 10–3).
- AndroidManifest.xml: Define the new list navigation action bar activity in the manifest file (Listing 10–11).

SPINNER ADAPTER

To be able to initialize the action bar with list navigation mode, we need the following two things:

- A spinner adapter that can tell the list navigation what the list of navigation text is
- A list navigation listener so that when one of the list items is picked we can get a call back

Listing 10–12 presents the SimpleSpinnerArrayAdapter that implements the SpinnerAdapter interface. the goal of this class is to give a list of items to show.

Listing 10–12. Creating a Spinner Adapter for List Navigation

```
//SimpleSpinnerArrayAdapter.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class SimpleSpinnerArrayAdapter
extends ArrayAdapter<String>
implements SpinnerAdapter
{
public SimpleSpinnerArrayAdapter(Context ctx)
{
super(ctx,
android.R.layout.simple_spinner_item,
new String[]{"one","two"});
this.setDropDownViewResource(
android.R.layout.simple_spinner_dropdown_item);
}
public View getDropDownView(
int position, View convertView, ViewGroup parent)
{
return super.getDropDownView(
position, convertView, parent);
}
}
```

There is no SDK class that directly implements the SpinnerAdapter interface required by list navigation. So, we derive this class from an ArrayAdapter and provide a simple implementation for the SpinnerAdapter. At the end of the chapter is a reference URL on spinner adapters for further reading. Let's move on now to the list navigation listener.

LIST LISTENER

This is a simple implementing the ActionBar.OnNavigationListener. Listing 10–13 shows the code for this class.

Listing 10–13. Creating a List Listener for List Navigation

```
//ListListener.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class ListListener
extends BaseListener
```

```
implements ActionBar.OnNavigationListener
{
public ListListener(
Context ctx, IReportBack target)
{
super(ctx, target);
}
public boolean onNavigationItemSelected(
int itemPosition, long itemId)
{
this.mReportTo.reportBack(
"list listener","ItemPostion:" + itemPosition);
return true;
}
}
```

Like the tabbed listener in Listing 10–5, we inherit from our BaseListener so that we can log events to the debug text view through the IReportBack interface.

LIST ACTION BAR

We now have what we require to set up a list navigation action bar. The source code for the list navigation action bar activity is shown in Listing 10–14. This class is very similar to the tabbed activity we coded earlier.

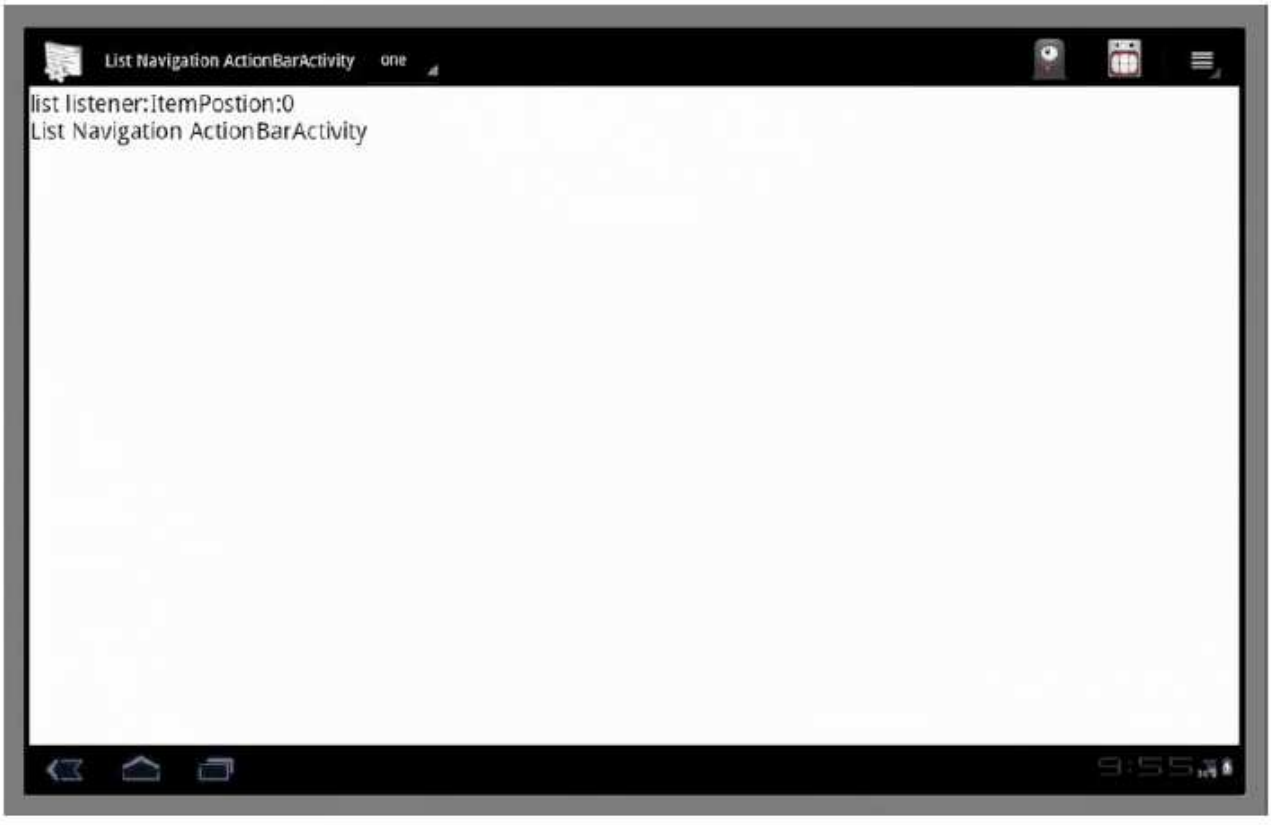

Figure 10–3. *An activity with a list navigation action bar*

Listing 10–14. List Navigation Action Bar Activity

```
// ListNavigationActionBarActivity.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class ListNavigationActionBarActivity
extends BaseActionBarActivity
{
private static String tag=
"List Navigation ActionBarActivity";
public ListNavigationActionBarActivity()
{
super(tag);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
workwithListActionBar();
}
public void workwithListActionBar()
{
ActionBar bar = this.getActionBar();
bar.setTitle(tag);
bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
bar.setListNavigationCallbacks(
new SimpleSpinnerArrayAdapter(this),
new ListListener(this,this));
}
}//eof-class
```

The important code is highlighted in Listing 10–14. The code is quite simple: we take a spinner adapter and a list listener and set them as list navigation callbacks on the action bar.

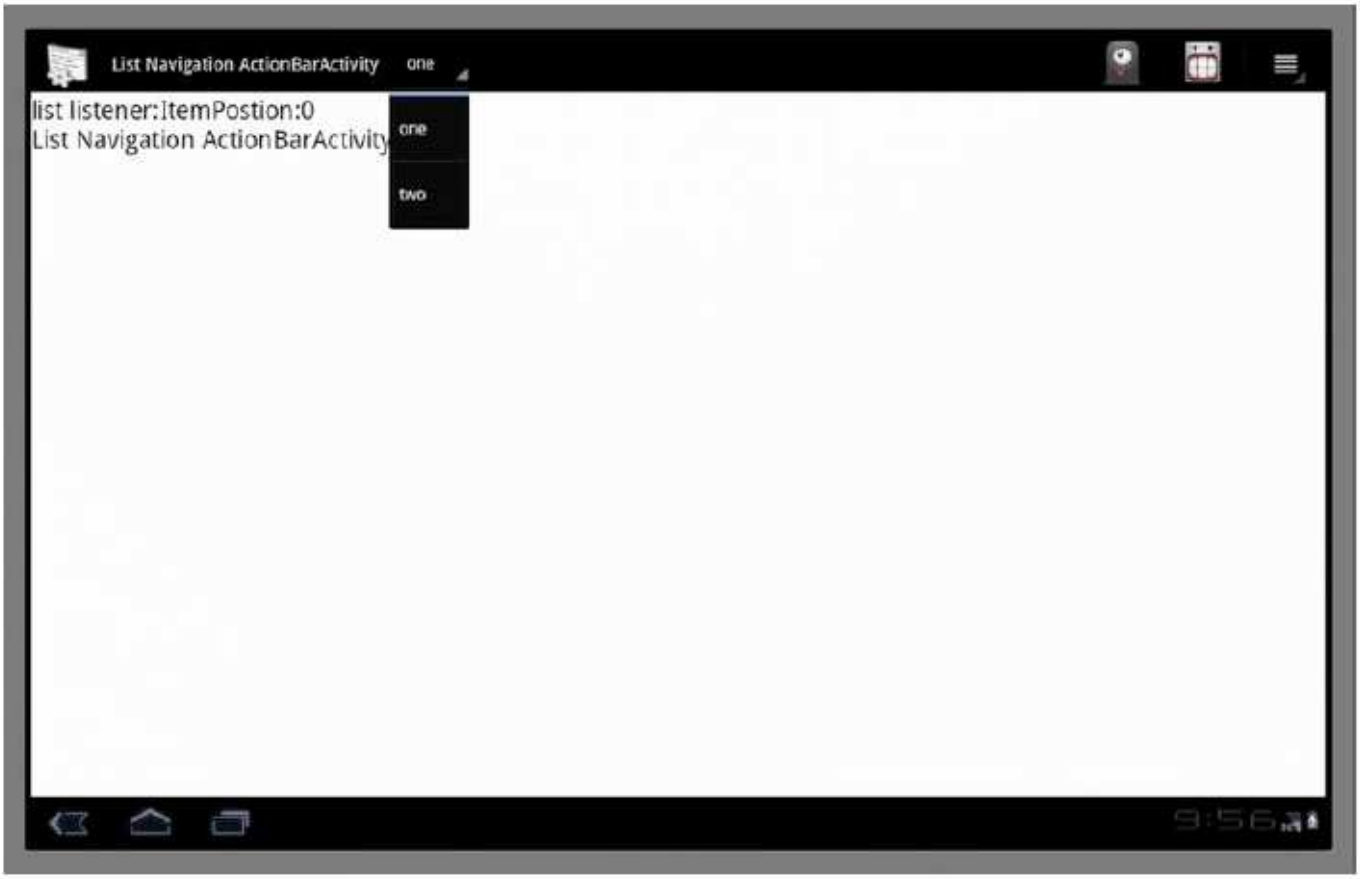

Figure 10–4. *An activity with an opened navigation list*

STANDARD NAVIGATION ACTION BAR ACTIVITY

The nature of a standard navigation action bar. We set up an activity and set its action bar navigation mode as standard. We then see what the standard navigation looks like and examine its behavior. As in the case of ListNavigationActionBarActivity, because our base classes are carrying most of the work, it is easy to implement and test the standard action bar navigation activity. We need the following additional file:

- StandardNavigationActionBarActivity.java: This is the implementation file for configuring the action bar as a standard navigation mode action bar (Listing 10–17).

We also need to update the following two files:

- BaseActionBarActivity.java: Uncomment the invocation of the standard action bar activity in response to a menu item (see Listing 10–18 for changes and Listing 10–3 for the original file).
- AndroidManifest.xml: Define this new activity in the manifest file (see Listing 10–19 for this activity's definition so we can add this to the main Android manifest file in Listing 10–11).

We used tabbed listeners while setting up the tabbed action bar and list listeners for setting up the list navigation action bar. For a standard action bar, there are no listeners other than the menu callbacks. The menu callbacks don't need to be specially set up because they are hooked up automatically by the SDK. As a result, it is quite easy to set up the action bar in the standard

navigation mode. Listing 10–17 presents the source code for the standard navigation action bar activity.

Listing 10–17. Standard Navigation Action Bar Activity

```
//StandardNavigationActionBarActivity.java
package com.androidbook.actionbar;
//Use CTRL-SHIFT-O to import dependencies
public class StandardNavigationActionBarActivity
extends BaseActionBarActivity
{
private static String tag=
"Standard Navigation ActionBarActivity";
public StandardNavigationActionBarActivity()
{
super(tag);
}
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
workwithStandardActionBar();
}
public void workwithStandardActionBar()
{
ActionBar bar = this.getActionBar();
bar.setTitle(tag);
bar.setNavigationMode(ActionBar.NAVIGATION_MODE_STANDARD);
//test to see what happens if we were to attach tabs
attachTabs(bar);
}
public void attachTabs(ActionBar bar)
{
TabListener tl = new TabListener(this,this);
Tab tab1 = bar.newTab();
tab1.setText("Tab1");
tab1.setTabListener(tl);
bar.addTab(tab1);
Tab tab2 = bar.newTab();
tab2.setText("Tab2");
tab2.setTabListener(tl);
bar.addTab(tab2);
}
}//eof-class
```

The only thing necessary to set up an action bar as a standard navigation action bar is to set its navigation mode as such. That portion of the code is highlighted in Listing 10– 17.

ACTION BAR AND SEARCH VIEW

In Android 4.0, because the action bar is available on phones, there is an increasing interest in using it as a search facility. This section shows how to use a search widget in the action bar. We will provide code snippets that we can use to modify the project we have seen so far to include a search widget. Although we present only snippets, we can see the full code in the downloadable project for this chapter. A search view widget is a search box that fits between our tabs and the menu icons in the action bar, as shown in Figure 10–7.

Figure 10–5. *An activity with a standard navigation action bar*

Figure 10–7. *An action bar search view*

We need to do the following to use search in our action bar:

1. Define a menu item pointing to a search view provided by the SDK. We also need an activity into which we can load this menu. This is often called the search invoker activity.
2. Create another activity that can take the query from the search view in step 1 and provide results. This is often called the search results activity.
3. Create an XML file that allows we to customize the search view widget.

4. This file is often called searchable.xml and resides in the res/xml subdirectory.
5. Declare the search results activity in the manifest file. This definition needs to point to the XML file defined in step 3.
6. In our menu setup for the search invoker activity, indicate that the search view needs to target the search results activity from step 2.

We will provide code snippets for each of these steps. As mentioned earlier, the complete code is available in the downloadable project. In fact, when we run the project for this chapter, the search view is visible on all the action bars presented in the previous sections of this chapter: tab, list, and standard.

ACTION BAR AND FRAGMENTS.

The action bar is generally recommended for use with fragments when we're dealing with tablets. Because fragments are inside an activity, and an activity owns the action bar, we don't need the abstraction of a base class to ensure the same action bar for each activity. All fragments share the same activity, so they also share the same action bar. The solution is simpler.