Introducing the Android Menu System

If you've ever tried to navigate a mobile phone menu system using a stylus or trackball, you'll know that traditional menu systems are awkward to use on mobile devices.

To improve the usability of application menus, Android features a three-stage menu system optimized for small screens:

❑ **The Icon Menu** This compact menu (shown in Figure 4-4) appears along the bottom of the screen when the Menu button is pressed. It displays the icons and text for up to six Menu Items (or submenus).

Figure 4-4

This icon menu does *not* display checkboxes, radio buttons, or the shortcut keys for Menu Items, so it's generally good practice not to assign checkboxes or radio buttons to icon menu items, as they will not be available.

If more than six Menu Items have been defi ned, a *More* item is included that, when selected, displays the expanded menu. Pressing the Back button closes the icon menu.

❑ **The Expanded Menu** The expanded menu is triggered when a user selects the **More** Menu Item from the icon menu. The expanded menu (shown in Figure 4-5) displays a scrollable list of *only* the Menu Items that weren't visible in the icon menu. This menu displays full text, shortcut keys, and checkboxes/radio buttons as appropriate.

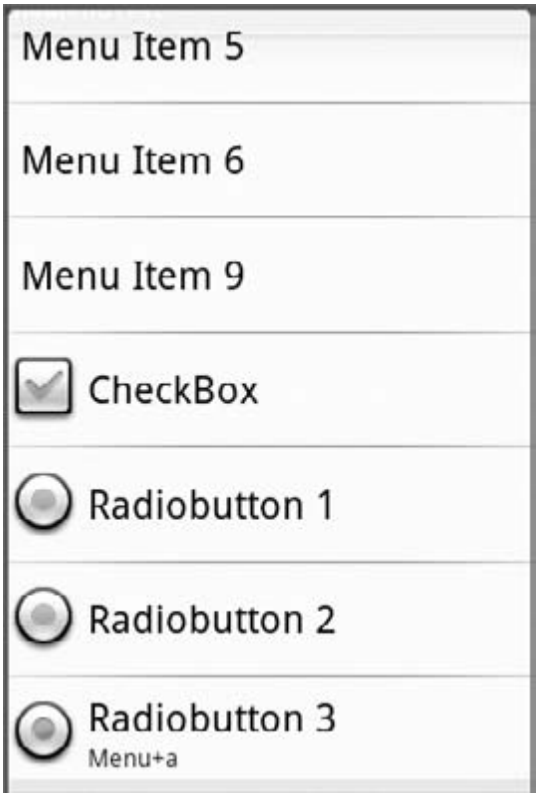

Figure 4-5

It does not, however, display icons. As a result, you should avoid assigning icons to Menu Items that are likely to appear only in the expanded menu.

Pressing Back from the expanded menu returns to the icon menu.

You cannot force Android to display the expanded menu instead of the icon menu. As a result, special care must be taken with Menu Items that feature checkboxes or radio buttons to ensure that they are either available only in the extended menu, or that their state information is also indicated using an icon or change in text.

❑ **Submenus** The traditional "expanding hierarchical tree" can be awkward to navigate using a mouse, so it's no surprise that this metaphor is particularly ill-suited for use on mobile devices. The Android alternative is to display each submenu in a fl oating window. For example, when a user selects a submenu such as the creatively labeled *Submenu* from Figure 4-5, its items are displayed in a fl oating menu Dialog box, as shown in Figure 4-6.

Figure 4-6

Note that the name of the submenu is shown in the header bar and that each Menu Item is displayed with its full text, checkbox (if any), and shortcut key. Since Android does not support nested submenus, you can't add a submenu to a submenu (trying will result in an exception).

As with the extended menu, icons are not displayed in the submenu items, so it's good practice to avoid assigning icons to submenu items.

Pressing the Back button closes the fl oating window without navigating back to the extended or icon menus.

Defining an Activity Menu

To defi ne a menu for an Activity, override its onCreateOptionsMenu method. This method is triggered the fi rst time an Activity's menu is displayed.

The onCreateOptionsMenu receives a Menu object as a parameter. You can store a reference to, and continue to use, the Menu reference elsewhere in your code until the next time that onCreateOptionsMenu is called.

You should always call through to the base implementation as it automatically includes additional system menu options where appropriate.

Use the add method on the Menu object to populate your menu. For each Menu Item, you must specify:

❑ A group value to separate Menu Items for batch processing and ordering

❑ A unique identifi er for each Menu Item. For effi ciency reasons, Menu Item selections are generally

handled by the onOptionsItemSelected event handler, so this unique identifi er is important to determine which Menu Item was pressed. It is convention to declare each menu ID as a private static variable within the Activity class. You can use the Menu. FIRST static constant and simply increment that value for each subsequent item.

❑ An order value that defi nes the order in which the Menu Items are displayed

❑ The menu text, either as a character string or as a string resource

When you have fi nished populating the menu, return True to allow Android to display the menu.
The following skeleton code shows how to add a single item to an Activity menu:

```
static final private int MENU_ITEM = Menu.FIRST;
@Override
public boolean onCreateOptionsMenu(Menu menu) {
super.onCreateOptionsMenu(menu);
// Group ID
int groupId = 0;
// Unique menu item identifier. Used for event handling.
int menuItemId = MENU_ITEM;
// The order position of the item
int menuItemOrder = Menu.NONE;
// Text to be displayed for this menu item.
int menuItemText = R.string.menu_item;
// Create the menu item and keep a reference to it.
MenuItem menuItem = menu.add(groupId, menuItemId,
menuItemOrder, menuItemText);
return true;
}
```

Like the Menu object, each Menu Item reference returned by a call to add is valid until the next call to onCreateOptionsMenu. Rather than maintaining a reference to each item, you can fi nd a particular Menu Item by passing its ID into the Menu.findItem method.

Menu Item Options

Android supports most of the traditional Menu Item options you're probably familiar with, including icons, shortcuts, checkboxes, and radio buttons, as described below:

❑ **Checkboxes and Radio Buttons** Checkboxes and radio buttons on Menu Items are visible in expanded menus and submenus, as shown in Figure 4-6. To set a Menu Item as a checkbox, use the setCheckable method. The state of that checkbox is controlled using setChecked.

A *radio button group* is a group of items displaying circular buttons, where only one item can be selected at any given time. Checking one of these items will automatically unselect any other item in the same group, that is currently checked. To create a radio button group, assign the same group identifi er to each item, then call Menu.setGroupCheckable, passing in that group identifi er and setting the exclusive parameter to True.

Checkboxes are not visible in the icon menu, so Menu Items that feature checkboxes should be reserved for submenus and items that appear only in the expanded menu. The following code snippet shows how to add a checkbox and a group of three radio buttons:

```
// Create a new check box item.
menu.add(0, CHECKBOX_ITEM, Menu.NONE, "CheckBox").setCheckable(true);
// Create a radio button group.
menu.add(RB_GROUP, RADIOBUTTON_1, Menu.NONE, "Radiobutton 1");
menu.add(RB_GROUP, RADIOBUTTON_2, Menu.NONE, "Radiobutton 2");
menu.add(RB_GROUP, RADIOBUTTON_3, Menu.NONE,
"Radiobutton 3").setChecked(true);
menu.setGroupCheckable(RB_GROUP, true, true);
```

❑ **Shortcut Keys** You can specify a keypad shortcut for a Menu Item using the setShortcut method. Each call to setShortcut requires two shortcut keys, one for use with the numeric keypad and a second to support a full keyboard. Neither key is case-sensitive.

The code below shows how to set a shortcut for both modes:

```
// Add a shortcut to this menu item, '0' if using the numeric keypad
// or 'b' if using the full keyboard.
menuItem.setShortcut('0', 'b');
```

❑ **Condensed Titles** The icon menu does not display shortcuts or checkboxes, so it's often necessary to modify its display text to indicate its state. The following code shows how to set the icon menu–specifi c text for a Menu Item:

```
menuItem.setTitleCondensed("Short Title");
```

❑ **Icons** Icon is a drawable resource identifi er for an icon to be used in the Menu Item. Icons are only displayed in the icon menu; they are not visible in the extended menu or submenus. The following snippet shows how to apply an icon to a Menu Item:

```
menuItem.setIcon(R.drawable.menu_item_icon);
```

❑ **Menu Item Click Listener** An event handler that will execute when the Menu Item is selected. For effi ciency reasons, this is discouraged; instead, Menu Item selections should be handled by the onOptionsItemSelected handler as shown later in this section. To apply a click listener to a Menu Item, use the pattern shown in the following code snippet:

```
menuItem.setOnMenuItemClickListener(new OnMenuItemClickListener() {
public boolean onMenuItemClick(MenuItem _menuItem) {
[ ... execute click handling, return true if handled ... ]
return true;
}
});
```

❑ **Intents** An Intent assigned to a Menu Item is triggered when clicking a Menu Item isn't handled by either a MenuItemClickListener or the Activity's onOptionsItemSelected handler. When triggered, Android will execute startActivity, passing in the specifi ed Intent. The following code snippet shows how to specify an Intent for a Menu Item:

```
menuItem.setIntent(new Intent(this, MyOtherActivity.class));
```

Dynamically Updating Menu Items

By overriding your activity's onPrepareOptionsMenu method, you can modify your menu based on the application state each time it's displayed. This lets you dynamically disable/enable each item, set visibility, and modify text at runtime.

To modify Menu Items dynamically, you can either keep a reference to them when they're created in the onCreateOptionsMenu method, or you can use menu.findItem as shown in the following skeleton code, where onPrepareOptionsMenu is overridden:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
super.onPrepareOptionsMenu(menu);
MenuItem menuItem = menu.findItem(MENU_ITEM);
[ ... modify menu items ... ]
return true;
}
```

Handling Menu Selections

Android handles all of an Activity's Menu Item selections using a single event handler, the onOptions ItemSelected method. The Menu Item selected is passed in to this method as the MenuItem parameter.

To react to the menu selection, compare the item.getItemId value to the Menu Item identifi ers you used when populating the menu, and react accordingly, as shown in the following code:

```
public boolean onOptionsItemSelected(MenuItem item) {
super.onOptionsItemSelected(item);
// Find which menu item has been selected
switch (item.getItemId()) {
// Check for each known menu item
case (MENU_ITEM):
[ ... Perform menu handler actions ... ]
return true;
}
// Return false if you have not handled the menu item.
return false;
}
```

Submenus and Context Menus

Context menus are displayed using the same fl oating window as the submenus shown in Figure 4-5. While their appearance is the same, the two menu types are populated differently.

Creating Submenus

Submenus are displayed as regular Menu Items that, when selected, reveal more items. Traditionally, submenus are displayed using a hierarchical tree layout. Android uses a different approach to simplify menu navigation for small-screen devices. Rather than a tree structure, selecting a submenu presents a single fl oating window that displays all of its Menu Items.

You can add submenus using the addSubMenu method. It supports the same parameters as the add method used to add normal Menu Items, allowing you to specify a group, unique identifi er, and text string for each submenu. You can also use the setHeaderIcon and setIcon methods to specify an icon to display in the submenu's header bar or the regular icon menu, respectively.

The Menu Items within a submenu support the same options as those assigned to the icon or extended menus. However, unlike traditional systems, Android does not support nested submenus. The code snippet below shows an extract from an implementation of the onCreateMenuOptions code that adds a submenu to the main menu, sets the header icon, and then adds a submenu Menu Item:

```
SubMenu sub = menu.addSubMenu(0, 0, Menu.NONE, "Submenu");
sub.setHeaderIcon(R.drawable.icon);
sub.setIcon(R.drawable.icon);
MenuItem submenuItem = sub.add(0, 0, Menu.NONE, "Submenu Item");
```

Using Context Menus

Context Menus are contextualized by the currently focused View and are triggered by pressing the trackball, middle D-pad button, or the View for around 3 seconds.

You defi ne and populate Context Menus similarly to the Activity menu. There are two options available for creating Context Menus for a particular View.

Creating Context Menus

The fi rst option is to create a generic Context Menu for a View class by overriding a View's onCreate ContextMenu handler as shown below:

```
@Override
public void onCreateContextMenu(ContextMenu menu) {
super.onCreateContextMenu(menu);
menu.add("ContextMenuItem1");
}
```

The Context Menu created here will be available from any Activity that includes this View class.

The more common alternative is to create Activity-specifi c Context Menus by overriding the onCreate ContextMenu method and registering the Views that should use it. Register Views using register ForContextMenu and passing them in, as shown in the following code:

```
@Override
public void onCreate(Bundle icicle) {
super.onCreate(icicle);
EditText view = new EditText(this);
setContentView(view);
registerForContextMenu(view);
}
```

Once a View has been registered, the onCreateContextMenu handler will be triggered whenever a Context Menu should be displayed for that View.

Override onCreateContextMenu and check which View has triggered the menu creation to populate the menu parameter with the appropriate contextual items, as shown in the following code snippet:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
super.onCreateContextMenu(menu, v, menuInfo);
menu.setHeaderTitle("Context Menu");
menu.add(0, menu.FIRST, Menu.NONE,
```

```
"Item 1").setIcon(R.drawable.menu_item);
menu.add(0, menu.FIRST+1, Menu.NONE, "Item 2").setCheckable(true);
menu.add(0, menu.FIRST+2, Menu.NONE, "Item 3").setShortcut('3', '3');
SubMenu sub = menu.addSubMenu("Submenu");
sub.add("Submenu Item");
}
```

As shown above, the ContextMenu class supports the same add method as the Menu class, so you can populate a Context Menu in the same way as Activity menus — including support for submenus — but icons will never be displayed. You can also specify the title and icon to display in the Context Menu's header bar.

Android supports late runtime population of Context Menus using Intent Filters. This mechanism lets you populate a Context Menu by specifying the kind of data presented by the current View, and asking other Android applications if they support any actions for it. The most common example of this behavior is the cut/copy/paste Menu Items available on EditText controls. This process is covered in detail in the next chapter.

Handling Context Menu Selections

Context Menu Item selections are handled similarly to the Activity menu. You can attach an Intent or Menu Item Click Listener directly to each Menu Item, or use the preferred technique of overriding the onContextItemSelected method on the Activity.

This event handler is triggered whenever a Context Menu Item is selected within the Activity. A skeleton implementation is shown below:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
super.onContextItemSelected(item);
[ ... Handle menu item selection ... ]
return false;
}
```

To-Do List Example Continued

In the following example, you'll be adding some simple menu functions to the To-Do List application you started in Chapter 2 and continued to improve previously in this chapter.

You will add the ability to remove items from Context and Activity Menus, and improve the use of screen space by displaying the text entry box only when adding a new item.

1. Start by importing the packages you need to support menu functionality into the ToDoList Activity class.

```
import android.view.Menu;
import android.view.MenuItem;
import android.view.ContextMenu;
import android.widget.AdapterView;
```

2. Then add private static fi nal variables that defi ne the unique IDs for each Menu Item.

```
static final private int ADD_NEW_TODO = Menu.FIRST;
static final private int REMOVE_TODO = Menu.FIRST + 1;
```

3. Now override the onCreateOptionsMenu method to add two new Menu Items, one to add and the other to remove the to-do item. Specify the appropriate text, and assign icon resources and shortcut keys for each item.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
super.onCreateOptionsMenu(menu);
// Create and add new menu items.
MenuItem itemAdd = menu.add(0, ADD_NEW_TODO, Menu.NONE,
R.string.add_new);
MenuItem itemRem = menu.add(0, REMOVE_TODO, Menu.NONE,
R.string.remove);
// Assign icons
itemAdd.setIcon(R.drawable.add_new_item);
itemRem.setIcon(R.drawable.remove_item);
// Allocate shortcuts to each of them.
itemAdd.setShortcut('0', 'a');
itemRem.setShortcut('1', 'r');
return true; }
```

If you run the Activity, pressing the Menu button should appear as shown in Figure 4-7.

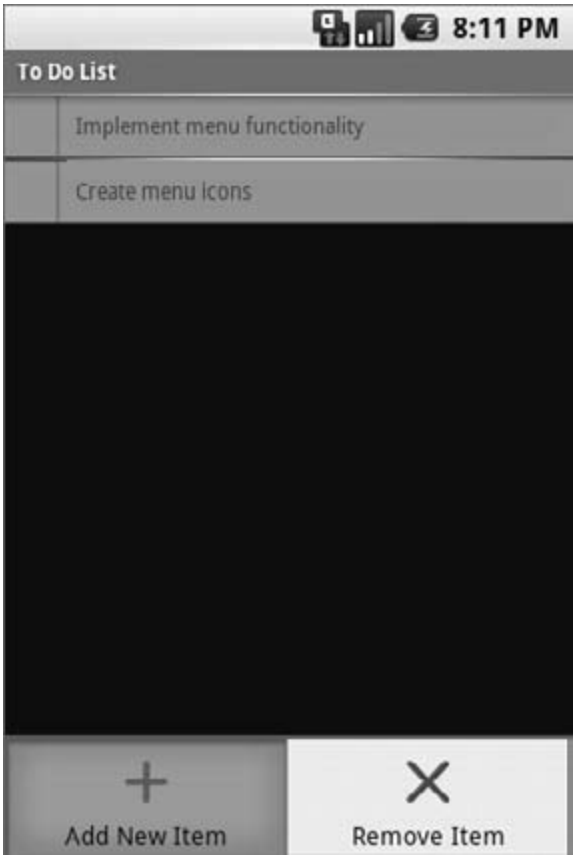

Figure 4-7

4. Having populated the Activity Menu, create a Context Menu. First, modify onCreate to register the ListView to receive a Context Menu. Then override onCreateContextMenu to populate the menu with a "remove" item.

```
@Override
public void onCreate(Bundle icicle) {
[ ... existing onCreate method ... ]
registerForContextMenu(myListView);
}
@Override
public void onCreateContextMenu(ContextMenu menu,
View v,
ContextMenu.ContextMenuInfo menuInfo) {
super.onCreateContextMenu(menu, v, menuInfo);
menu.setHeaderTitle("Selected To Do Item");
menu.add(0, REMOVE_TODO, Menu.NONE, R.string.remove);
}
```

5. Now modify the appearance of the menu based on the application context, by overriding the onPrepareOptionsMenu method. The menu should be customized to show "cancel" rather than "delete" if you are currently adding a new Menu Item.

```
private boolean addingNew = false;
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
super.onPrepareOptionsMenu(menu);
int idx = myListView.getSelectedItemPosition();
String removeTitle = getString(addingNew ?
R.string.cancel : R.string.remove);
MenuItem removeItem = menu.findItem(REMOVE_TODO);
```

```
removeItem.setTitle(removeTitle);
removeItem.setVisible(addingNew || idx > -1);
return true;
}
```

6. For the code in Step 5 to work, you need to increase the scope of the todoListItems and ListView control beyond the onCreate method. Do the same thing for the ArrayAdapter and EditText to support the *add* and *remove* actions when they're implemented later.

```
private ArrayList<String> todoItems;
private ListView myListView;
private EditText myEditText;
private ArrayAdapter<String> aa;
@Override
public void onCreate(Bundle icicle) {
super.onCreate(icicle);
// Inflate your view
setContentView(R.layout.main);
// Get references to UI widgets
myListView = (ListView)findViewById(R.id.myListView);
myEditText = (EditText)findViewById(R.id.myEditText);
todoItems = new ArrayList<String>();
int resID = R.layout.todolist_item;
aa = new ArrayAdapter<String>(this, resID, todoItems);
myListView.setAdapter(aa);
myEditText.setOnKeyListener(new OnKeyListener() {
public boolean onKey(View v, int keyCode, KeyEvent event) {
if (event.getAction() == KeyEvent.ACTION_DOWN)
if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER)
{
todoItems.add(0, myEditText.getText().toString());
myEditText.setText("");
aa.notifyDataSetChanged();
return true;
}
return false;
}
});
registerForContextMenu(myListView);
}
```

7. Next you need to handle Menu Item clicks. Override the onOptionsItemSelected and onContextItemSelected methods to execute stubs that handle the new Menu Items.

7.1. Start by overriding onOptionsItemSelected to handle the Activity menu selections. For the *remove* menu option, you can use the getSelectedItemPosition method on the List View to fi nd the currently highlighted item.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
super.onOptionsItemSelected(item);
int index = myListView.getSelectedItemPosition();
switch (item.getItemId()) {
case (REMOVE_TODO): {
if (addingNew) {
cancelAdd();
}
else {
removeItem(index);
}
return true;
}
case (ADD_NEW_TODO): {
addNewItem();
return true;
}
}
return false;
}
```

7.2. Next override onContextItemSelected to handle Context Menu Item selections. Note that you are using the AdapterView specifi c implementation of ContextMenuInfo. This includes a reference to the View that triggered the Context Menu and the position of the data it's displaying in the underlying Adapter. Use the latter to fi nd the index of the item to remove.

```
@Override
public boolean onContextItemSelected(MenuItem item) {
super.onContextItemSelected(item);
switch (item.getItemId()) {
case (REMOVE_TODO): {
AdapterView.AdapterContextMenuInfo menuInfo;
menuInfo =(AdapterView.AdapterContextMenuInfo)item.getMenuInfo();

int index = menuInfo.position;
removeItem(index);
return true;
}
}
return false;
}
```

7.3. Create the stubs called in the Menu Item selection handlers you created above.

```
private void cancelAdd() {
}
private void addNewItem() {
}
private void removeItem(int _index) {
}
```

8. Now implement each of the stubs to provide the new functionality.

```
private void cancelAdd() {
addingNew = false;
myEditText.setVisibility(View.GONE);
}
private void addNewItem() {
addingNew = true;
myEditText.setVisibility(View.VISIBLE);
myEditText.requestFocus();
}
private void removeItem(int _index) {
todoItems.remove(_index);
aa.notifyDataSetChanged();
}
```

9. Next you need to hide the text entry box after you've added a new item. In the onCreate method, modify the onKeyListener to call the cancelAdd function after adding a new item.

```
myEditText.setOnKeyListener(new OnKeyListener() {
public boolean onKey(View v, int keyCode, KeyEvent event) {
if (event.getAction() == KeyEvent.ACTION_DOWN)
if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER)
{
todoItems.add(0, myEditText.getText().toString());
myEditText.setText("");
aa.notifyDataSetChanged();
cancelAdd();
return true;
}
return false;
}
});
```

10. Finally, to ensure a consistent UI, modify the main.xml layout to hide the text entry box until the user chooses to add a new item.

```
<EditText
android:id="@+id/myEditText"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text=""
android:visibility="gone" />
```

Running the application should now let you trigger the Activity menu to add or remove items from the list, and a Context Menu on each item should offer the option of removing it.

Summary

You now know the basics of creating intuitive User Interfaces for Android applications. You learned about Views and layouts and were introduced to the Android menu system.

Activity screens are created by positioning Views using Layout Managers that can be created in code or as resource fi les. You learned how to extend, group, and create new View-based controls to provide customized appearance and behavior for your applications.

In this chapter, you:

- ❑ Were introduced to some of the controls and widgets available as part of the Android SDK.
- ❑ Learned how to use your custom controls within Activities.
- ❑ Discovered how to create and use Activity Menus and Context Menus.
- ❑ Extended the To-Do List Example to support custom Views and menu-based functions.
- ❑ Created a new CompassView control from scratch.

Having covered the fundamentals of Android UI design, the next chapter focuses on binding application components using Intents, Broadcast Receivers, and Adapters. You will learn how to start new Activities and broadcast and consume requests for actions. Chapter 5 also introduces Internet connectivity and looks at the Dialog class.