Creating Menus

Menus are an important part of an activity's user interface, which provide users a familiar way to perform actions. Android offers a simple framework for you to add standard menus to your application.

There are three types of application menus:

Options Menu
The primary collection of menu items for an activity, which appears when the user touches the MENU button. When your application is running on Android 3.0 or later, you can provide quick access to select menu items by placing them directly in the Action Bar, as "action items."

Context Menu
A floating list of menu items that appears when the user touches and holds a view that's registered to provide a context menu.

Submenu
A floating list of menu items that appears when the user touches a menu item that contains a nested menu.

This document shows you how to create each type of menu, using XML to define the content of the menu and callback methods in your activity to respond when the user selects an item.

Creating a Menu Resource

Instead of instantiating a `Menu` in your application code, you should define a menu and all its items in an XML menu resource, then inflate the menu resource (load it as a programmable object) in your application code. Using a menu resource to define your menu is a good practice because it separates the content for the menu from your application code. It's also easier to visualize the structure and content of a menu in XML.

To create a menu resource, create an XML file inside your project's `res/menu/` directory and build the menu with the following elements:

`<menu>`
Defines a `Menu`, which is a container for menu items. A `<menu>` element must be the root node for the file and can hold one or more `<item>` and `<group>` elements.

`<item>`
Creates a `MenuItem`, which represents a single item in a menu. This element may contain a nested `<menu>` element in order to create a submenu.

`<group>`
An optional, invisible container for `<item>` elements. It allows you to categorize menu items so they share properties such as active state and visibility. See the section about Menu groups.

Here's an example menu named `game_menu.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:icon="@drawable/ic_new_game"
          android:title="@string/new_game" />
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
          android:title="@string/help" />
</menu>
```

This example defines a menu with two items. Each item includes the attributes:

`android:id`
: A resource ID that's unique to the item, which allows the application can recognize the item when the user selects it.

`android:icon`
: A reference to a drawable to use as the item's icon.

`android:title`
: A reference to a string to use as the item's title.

There are many more attributes you can include in an `<item>`, including some that specify how the item may appear in the Action Bar. For more information about the XML syntax and attributes for a menu resource, see the Menu Resource reference.

Inflating a Menu Resource

From your application code, you can inflate a menu resource (convert the XML resource into a programmable object) using `MenuInflater.inflate()`. For example, the following code inflates the `game_menu.xml` file defined above, during the `onCreateOptionsMenu()` callback method, to use the menu as the activity's Options Menu:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

The `getMenuInflater()` method returns a `MenuInflater` for the activity. With this object, you can call `inflate()`, which inflates a menu resource into a `Menu` object. In this example, the menu resource defined by `game_menu.xml` is inflated into the `Menu` that was passed into `onCreateOptionsMenu()`. (This callback method for the Options Menu is discussed more in the next section.)

Creating an Options Menu

The Options Menu is where you should include basic activity actions and necessary navigation items (for example, a button to open the application settings). Items in the Options Menu are accessible in two distinct ways: the MENU button or in the Action Bar (on devices running Android 3.0 or higher).

When running on a device with Android 2.3 and lower, the Options Menu appears at the bottom of the screen, as shown in figure 1. When opened, the first visible portion of the Options Menu is the icon menu. It holds the first six menu items. If you add more than six items to the Options Menu, Android places the sixth item and those after it into the overflow menu, which the user can open by touching the "More" menu item.

On Android 3.0 and higher, items from the Options Menu is placed in the Action Bar, which appears at the top of the activity in place of the traditional title bar. By default all items from the Options Menu are placed in the overflow menu,

which the user can open by touching the menu icon on the right side of the Action Bar. However, you can place select menu items directly in the Action Bar as "action items," for instant access, as shown in figure 2.

When the Android system creates the Options Menu for the first time, it calls your activity's `onCreateOptionsMenu()` method. Override this method in your activity and populate the `Menu` that is passed into the method, `Menu` by inflating a menu resource as described above in Inflating a Menu Resource. For example:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

You can also populate the menu in code, using `add()` to add items to the `Menu`.

> **Note:** On Android 2.3 and lower, the system calls `onCreateOptionsMenu()` to create the Options Menu when the user opens it for the first time, but on Android 3.0 and greater, the system creates it as soon as the activity is created, in order to populate the Action Bar.

Figure 1. Screenshot of the Options Menu in the Browser.

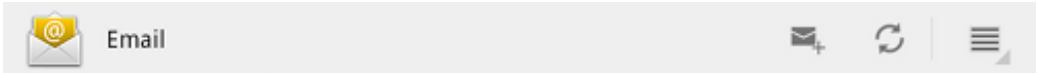

Figure 2. Screenshot of the Action Bar in the Email application, with two action items from the Options Menu, plus the overflow menu.

Responding to user action

When the user selects a menu item from the Options Menu (including action items in the Action Bar), the system calls your activity's `onOptionsItemSelected()` method. This method passes the `MenuItem` that the user selected. You can identify the menu item by calling `getItemId()`, which returns the unique ID for the menu item (defined by the `android:id` attribute in the menu resource or with an integer given to the `add()` method). You can match this ID against known menu items and perform the appropriate action. For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle item selection
    switch (item.getItemId()) {
    case R.id.new_game:
        newGame();
        return true;
    case R.id.help:
        showHelp();
        return true;
    default:
        return super.onOptionsItemSelected(item);
    }
}
```

In this example, `getItemId()` queries the ID for the selected menu item and the switch statement compares the ID against the resource IDs that were assigned to menu items in the XML resource. When a switch case successfully handles the menu item, it returns `true` to indicate that the item selection was handled. Otherwise, the default statement passes the menu item to the super class, in case it can handle the item selected. (If you've directly extended the `Activity` class, then the super class returns `false`, but it's a good practice to pass unhandled menu items to the super class instead of directly returning `false`.)

Additionally, Android 3.0 adds the ability for you to define the on-click behavior for a menu item in the menu resource XML, using the `android:onClick` attribute. So you don't need to implement `onOptionsItemSelected()`. Using the `android:onClick` attribute, you can specify a method to call when the user selects the menu item. Your activity

must then implement the method specified in the `android:onClick` attribute so that it accepts a single `MenuItem` parameter—when the system calls this method, it passes the menu item selected.

> **Tip:** If your application contains multiple activities and some of them provide the same Options Menu, consider creating an activity that implements nothing except the `onCreateOptionsMenu()` and `onOptionsItemSelected()` methods. Then extend this class for each activity that should share the same Options Menu. This way, you have to manage only one set of code for handling menu actions and each descendant class inherits the menu behaviors.
>
> If you want to add menu items to one of your descendant activities, override `onCreateOptionsMenu()` in that activity. Call `super.onCreateOptionsMenu(menu)` so the original menu items are created, then add new menu items with `menu.add()`. You can also override the super class's behavior for individual menu items.

Changing menu items at runtime

Once the activity is created, the `onCreateOptionsMenu()` method is called only once, as described above. The system keeps and re-uses the `Menu` you define in this method until your activity is destroyed. If you want to change the Options Menu any time after it's first created, you must override the `onPrepareOptionsMenu()` method. This passes you the `Menu` object as it currently exists. This is useful if you'd like to remove, add, disable, or enable menu items depending on the current state of your application.

On Android 2.3 and lower, the system calls `onPrepareOptionsMenu()` each time the user opens the Options Menu.

On Android 3.0 and higher, you must call `invalidateOptionsMenu()` when you want to update the menu, because the menu is always open. The system will then call `onPrepareOptionsMenu()` so you can update the menu items.

> **Note:** You should never change items in the Options Menu based on the `View` currently in focus. When in touch mode (when the user is not using a trackball or d-pad), views cannot take focus, so you should never use focus as the basis for modifying items in the Options Menu. If you want to provide menu items that are context-sensitive to a `View`, use a Context Menu.

If you're developing for Android 3.0 or higher, be sure to also read Using the Action Bar.

Creating a Context Menu

A context menu is conceptually similar to the menu displayed when the user performs a "right-click" on a PC. You should use a context menu to provide the user access to actions that pertain to a specific item in the user interface. On Android, a context menu is displayed when the user performs a "long press" (press and hold) on an item.

You can create a context menu for any View, though context menus are most often used for items in a `ListView`. When the user performs a long-press on an item in a ListView and the list is registered to provide a context menu, the list item signals to the user that a context menu is available by animating its background color—it transitions from orange to white before opening the context menu. (The Contacts application demonstrates this feature.)

In order for a View to provide a context menu, you must "register" the view for a context menu. Call `registerForContextMenu()` and pass it the `View` you want to give a context menu. When this View then receives a long-press, it displays a context menu.

To define the context menu's appearance and behavior, override your activity's context menu callback methods, `onCreateContextMenu()` and `onContextItemSelected()`.

Register a ListView

If your activity uses a `ListView` and you want all list items to provide a context menu, register all items for a context menu by passing the `ListView` to `registerForContextMenu()`. For example, if you're using a `ListActivity`, register all list items like this:

```
registerForContextMenu(getListView());
```

For example, here's an `onCreateContextMenu()` that uses the `context_menu.xml` menu resource:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenuInfo menuInfo) {
```

```
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.context_menu, menu);
}
```

MenuInflater is used to inflate the context menu from a menu resource. (You can also use add() to add menu items.) The callback method parameters include the View that the user selected and a ContextMenu.ContextMenuInfo object that provides additional information about the item selected. You might use these parameters to determine which context menu should be created, but in this example, all context menus for the activity are the same.

Then when the user selects an item from the context menu, the system calls onContextItemSelected(). Here is an example of how you can handle selected items:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId()) {
    case R.id.edit:
        editNote(info.id);
        return true;
    case R.id.delete:
        deleteNote(info.id);
        return true;
    default:
        return super.onContextItemSelected(item);
    }
}
```

The structure of this code is similar to the example for Creating an Options Menu, in which getItemId() queries the ID for the selected menu item and a switch statement matches the item to the IDs that are defined in the menu resource. And like the options menu example, the default statement calls the super class in case it can handle menu items not handled here, if necessary.

In this example, the selected item is an item from a ListView. To perform an action on the selected item, the application needs to know the list ID for the selected item (it's position in the ListView). To get the ID, the application calls getMenuInfo(), which returns a AdapterView.AdapterContextMenuInfo object that includes the list ID for the selected item in the id field. The local methods editNote() and deleteNote() methods accept this list ID to perform an action on the data specified by the list ID.

> **Note:** Items in a context menu do not support icons or shortcut keys.

Creating Submenus

A submenu is a menu that the user can open by selecting an item in another menu. You can add a submenu to any menu (except a submenu). Submenus are useful when your application has a lot of functions that can be organized into topics, like items in a PC application's menu bar (File, Edit, View, etc.).

When creating your menu resource, you can create a submenu by adding a <menu> element as the child of an <item>. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file"
          android:icon="@drawable/file"
          android:title="@string/file" >
        <!-- "file" submenu -->
        <menu>
            <item android:id="@+id/create_new"
```

```
                android:title="@string/create_new" />
            <item android:id="@+id/open"
                android:title="@string/open" />
        </menu>
    </item>
</menu>
```

When the user selects an item from a submenu, the parent menu's respective on-item-selected callback method receives the event. For instance, if the above menu is applied as an Options Menu, then the `onOptionsItemSelected()` method is called when a submenu item is selected.

You can also use `addSubMenu()` to dynamically add a `SubMenu` to an existing `Menu`. This returns the new `SubMenu` object, to which you can add submenu items, using `add()`

Other Menu Features

Here are some other features that you can apply to most menu items.

Menu groups

A menu group is a collection of menu items that share certain traits. With a group, you can:

- Show or hide all items with `setGroupVisible()`
- Enable or disable all items with `setGroupEnabled()`
- Specify whether all items are checkable with `setGroupCheckable()`

You can create a group by nesting `<item>` elements inside a `<group>` element in your menu resource or by specifying a group ID with the the `add()` method.

Here's an example menu resource that includes a group:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/item1"
          android:icon="@drawable/item1"
          android:title="@string/item1" />
    <!-- menu group -->
    <group android:id="@+id/group1">
        <item android:id="@+id/groupItem1"
              android:title="@string/groupItem1" />
        <item android:id="@+id/groupItem2"
              android:title="@string/groupItem2" />
    </group>
</menu>
```

The items that are in the group appear the same as the first item that is not in a group—all three items in the menu are siblings. However, you can modify the traits of the two items in the group by referencing the group ID and using the methods listed above.

Checkable menu items

A menu can be useful as an interface for turning options on and off, using a checkbox for stand-alone options, or radio buttons for groups of mutually exclusive options. Figure 2 shows a submenu with items that are checkable with radio buttons.

> **Note:** Menu items in the Icon Menu (from the Options Menu) cannot display a checkbox or radio button. If you choose to make items in the Icon Menu checkable, you must manually indicate the checked state by swapping the icon and/or text each time the state changes.

You can define the checkable behavior for individual menu items using the `android:checkable` attribute in the `<item>` element, or for an entire group with the `android:checkableBehavior` attribute in the `<group>` element. For example, all items in this menu group are checkable with a radio button:

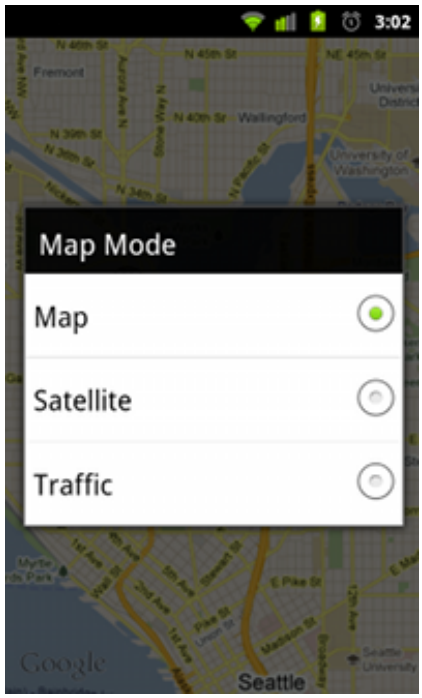

Figure 3. Screenshot of a submenu with checkable items.

```
<?xml version="1.0" encoding="utf-8"?>
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item android:id="@+id/red"
              android:title="@string/red" />
        <item android:id="@+id/blue"
              android:title="@string/blue" />
    </group>
</menu>
```

The `android:checkableBehavior` attribute accepts either:

`single`
: Only one item from the group can be checked (radio buttons)

`all`
: All items can be checked (checkboxes)

`none`
: No items are checkable

You can apply a default checked state to an item using the `android:checked` attribute in the `<item>` element and change it in code with the `setChecked()` method.

When a checkable item is selected, the system calls your respective item-selected callback method (such as `onOptionsItemSelected()`). It is here that you must set the state of the checkbox, because a checkbox or radio button does not change its state automatically. You can query the current state of the item (as it was before the user selected it) with `isChecked()` and then set the checked state with `setChecked()`. For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
  case R.id.vibrate:
  case R.id.dont_vibrate:
    if (item.isChecked()) item.setChecked(false);
    else item.setChecked(true);
    return true;
  default:
    return super.onOptionsItemSelected(item);
  }
```

```
}
```

If you don't set the checked state this way, then the visible state of the item (the checkbox or radio button) will not change when the user selects it. When you do set the state, the activity preserves the checked state of the item so that when the user opens the menu later, the checked state that you set is visible.

> **Note:** Checkable menu items are intended to be used only on a per-session basis and not saved after the application is destroyed. If you have application settings that you would like to save for the user, you should store the data using Shared Preferences.

Shortcut keys

To facilitate quick access to items in the Options Menu when the user's device has a hardware keyboard, you can add quick-access shortcut keys using letters and/or numbers, with the `android:alphabeticShortcut` and `android:numericShortcut` attributes in the `<item>` element. You can also use the methods `setAlphabeticShortcut(char)` and `setNumericShortcut(char)`. Shortcut keys are *not* case sensitive.

For example, if you apply the "s" character as an alphabetic shortcut to a "save" menu item, then when the menu is open (or while the user holds the MENU button) and the user presses the "s" key, the "save" menu item is selected.

This shortcut key is displayed as a tip in the menu item, below the menu item name (except for items in the Icon Menu, which are displayed only if the user holds the MENU button).

> **Note:** Shortcut keys for menu items only work on devices with a hardware keyboard. Shortcuts cannot be added to items in a Context Menu.

Dynamically adding menu intents

Sometimes you'll want a menu item to launch an activity using an `Intent` (whether it's an activity in your application or another application). When you know the intent you want to use and have a specific menu item that should initiate the intent, you can execute the intent with `startActivity()` during the appropriate on-item-selected callback method (such as the `onOptionsItemSelected()` callback).

However, if you are not certain that the user's device contains an application that handles the intent, then adding a menu item that invokes it can result in a non-functioning menu item, because the intent might not resolve to an activity. To solve this, Android lets you dynamically add menu items to your menu when Android finds activities on the device that handle your intent.

To add menu items based on available activities that accept an intent:

1. Define an intent with the category `CATEGORY_ALTERNATIVE` and/or `CATEGORY_SELECTED_ALTERNATIVE`, plus any other requirements.
2. Call `Menu.addIntentOptions()`. Android then searches for any applications that can perform the intent and adds them to your menu.

If there are no applications installed that satisfy the intent, then no menu items are added.

> **Note:** `CATEGORY_SELECTED_ALTERNATIVE` is used to handle the currently selected element on the screen. So, it should only be used when creating a Menu in `onCreateContextMenu()`.

For example:

```
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);

    // Create an Intent that describes the requirements to fulfill, to be included
    // in our menu. The offering app must include a category value of
Intent.CATEGORY_ALTERNATIVE.
    Intent intent = new Intent(null, dataUri);
    intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

```
    // Search and populate the menu with acceptable offering applications.
    menu.addIntentOptions(
         R.id.intent_group,  // Menu group to which new items will be added
         0,      // Unique item ID (none)
         0,      // Order for the items (none)
         this.getComponentName(),   // The current activity name
         null,   // Specific items to place first (none)
         intent, // Intent created above that describes our requirements
         0,      // Additional flags to control items (none)
         null);  // Array of MenuItems that correlate to specific items (none)

    return true;
}
```

For each activity found that provides an intent filter matching the intent defined, a menu item is added, using the value in the intent filter's `android:label` as the menu item title and the application icon as the menu item icon. The `addIntentOptions()` method returns the number of menu items added.

> **Note:** When you call `addIntentOptions()`, it overrides any and all menu items by the menu group specified in the first argument.

Allowing your activity to be added to other menus

You can also offer the services of your activity to other applications, so your application can be included in the menu of others (reverse the roles described above).

To be included in other application menus, you need to define an intent filter as usual, but be sure to include the `CATEGORY_ALTERNATIVE` and/or `CATEGORY_SELECTED_ALTERNATIVE` values for the intent filter category. For example:

```
<intent-filter label="Resize Image">
    ...
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    ...
</intent-filter>
```

Read more about writing intent filters in the Intents and Intent Filters document.

For a sample application using this technique, see the Note Pad sample code.

← Back to User Interface ↑ Go to top