Menus

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, you should use the `Menu` APIs to present user actions and other options in your activities.

Beginning with Android 3.0 (API level 11), Android-powered devices are no longer required to provide a dedicated *Menu* button. With this change, Android apps should migrate away from a dependence on the traditional 6-item menu panel and instead provide an action bar to present common user actions.

Although the design and user experience for some menu items have changed, the semantics to define a set of actions and options is still based on the `Menu` APIs. This guide shows how to create the three fundamental types of menus or action presentations on all versions of Android:

Options menu and action bar

The options menu is the primary collection of menu items for an activity. It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

If you're developing for Android 2.3 or lower, users can reveal the options menu panel by pressing the *Menu* button.

On Android 3.0 and higher, items from the options menu are presented by the action bar as a combination of on-screen action items and overflow options. Beginning with Android 3.0, the *Menu* button is deprecated (some devices don't have one), so you should migrate toward using the action bar to provide access to actions and other options.

See the section about Creating an Options Menu.

Context menu and contextual action mode

A context menu is a floating menu that appears when the user performs a long-click on an element. It provides actions that affect the selected content or context frame.

When developing for Android 3.0 and higher, you should instead use the contextual action mode to enable actions on selected content. This mode displays action items that affect the selected content in a bar at the top of the screen and allows the user to select multiple items.

See the section about Creating Contextual Menus.

Popup menu

A popup menu displays a list of items in a vertical list that's anchored to the view that invoked the menu. It's good for providing an overflow of actions that relate to specific content or to provide options for a second part of a command. Actions in a popup menu should **not** directly affect the corresponding content—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity.

See the section about Creating a Popup Menu.

Defining a Menu in XML

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, you should define a menu and all its items in an XMLmenu resource. You can then inflate the menu resource (load it as a `Menu` object) in your activity or fragment.

Using a menu resource is a good practice for a few reasons:

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your application's behavioral code.
- It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the app resources framework.

To define the menu, 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. For more information, see the section about Creating 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"
          android:showAsAction="ifRoom"/>
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
          android:title="@string/help" />
</menu>
```

The `<item>` element supports several attributes you can use to define an item's appearance and behavior. The items in the above menu include the following 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.

`android:showAsAction`

Specifies when and how this item should appear as an action item in the action bar.

These are the most important attributes you should use, but there are many more available. For information about all the supported attributes, see the Menu Resource document.

You can add a submenu to an item in any menu (except a submenu) by adding a `<menu>` element as the child of an `<item>`. 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.). 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: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>
```

To use the menu in your activity, you need to inflate the menu resource (convert the XML resource into a programmable object) using `MenuInflater.inflate()`. In the following sections, you'll see how to inflate a menu for each menu type.

Creating an Options Menu

Figure 1. Options menu in the Browser, on Android 2.3.

The options menu is where you should include actions and other options that are relevant to the current activity context, such as "Search," "Compose email," and "Settings."

Where the items in your options menu appear on the screen depends on the version for which you've developed your application:

- If you've developed your application for **Android 2.3.x (API level 10) or lower**, the contents of your options menu appear at the bottom of the screen when the user presses the *Menu* button, as shown in figure 1. When opened, the first visible portion is the icon menu, which holds up to six menu items. If your menu includes more than six items, Android places the sixth item and the rest into the overflow menu, which the user can open by selecting *More*.
- If you've developed your application for **Android 3.0 (API level 11) and higher**, items from the options menu are available in the action bar. By default, the system places all items in the action overflow, which the user can reveal with the action overflow icon on the right side of the action bar (or by pressing the device *Menu* button, if available). To enable quick access to important actions, you can promote a few items to appear in the action bar by adding `android:showAsAction="ifRoom"` to the corresponding `<item>` elements (see figure 2).

For more information about action items and other action bar behaviors, see the Action Bar guide.

Note: Even if you're *not* developing for Android 3.0 or higher, you can build your own action bar layout for a similar effect. For an example of how you can support older versions of Android with an action bar, see the Action Bar Compatibility sample.

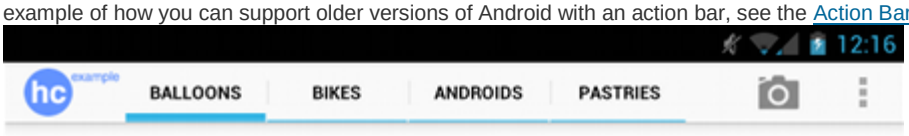

Figure 2. Action bar from the Honeycomb Gallery app, showing navigation tabs and a camera action item (plus the action overflow button).

You can declare items for the options menu from either your `Activity` subclass or a `Fragment` subclass. If both your activity and fragment(s) declare items for the options menu, they are combined in the UI. The activity's items appear first, followed by those of each fragment in the order in which each fragment is added to the activity. If necessary, you can re-order the menu items with the `android:orderInCategory` attribute in each `<item>` you need to move.

To specify the options menu for an activity, override `onCreateOptionsMenu()` (fragments provide their own `onCreateOptionsMenu()` callback). In this method, you can inflate your menu resource (defined in XML) into the `Menu` provided in the callback. For example:

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

You can also add menu items using `add()` and retrieve items with `findItem()` to revise their properties with `MenuItem` APIs.

If you've developed your application for Android 2.3.x and lower, the system calls `onCreateOptionsMenu()` to create the options menu when the user opens the menu for the first time. If you've developed for Android 3.0 and higher, the system calls `onCreateOptionsMenu()` when starting the activity, in order to show items to the action bar.

Handling click events

When the user selects an 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` selected. You can identify the 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 to 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);
    }
}
```

When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should call the superclass implementation of `onOptionsItemSelected()` (the default implementation returns false).

If your activity includes fragments, the system first calls `onOptionsItemSelected()` for the activity then for each fragment (in the order each fragment was added) until one returns `true` or all fragments have been called.

> **Tip:** Android 3.0 adds the ability for you to define the on-click behavior for a menu item in XML, using the `android:onClick` attribute. The value for the attribute must be the name of a method defined by the activity using the menu. The method must be public and accept a single `MenuItem` parameter—when the system calls this method, it passes the menu item selected. For more information and an example, see the Menu Resource document.
>
> **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 can manage 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 the 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

After the system calls `onCreateOptionsMenu()`, it retains an instance of the `Menu` you populate and will not call `onCreateOptionsMenu()` again unless the menu is invalidated for some reason. However, you should use `onCreateOptionsMenu()` only to create the initial menu state and not to make changes during the activity lifecycle.

If you want to modify the options menu based on events that occur during the activity lifecycle, you can do so in the `onPrepareOptionsMenu()` method. This method passes you the `Menu` object as it currently exists so you can modify it, such as add, remove, or disable items. (Fragments also provide an `onPrepareOptionsMenu()` callback.)

On Android 2.3.x and lower, the system calls `onPrepareOptionsMenu()` each time the user opens the options menu (presses the *Menu* button).

On Android 3.0 and higher, the options menu is considered to always be open when menu items are presented in the action bar. When an event occurs and you want to perform a menu update, you must call `invalidateOptionsMenu()` to request that the system call `onPrepareOptionsMenu()`.

> **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.

Creating Contextual Menus

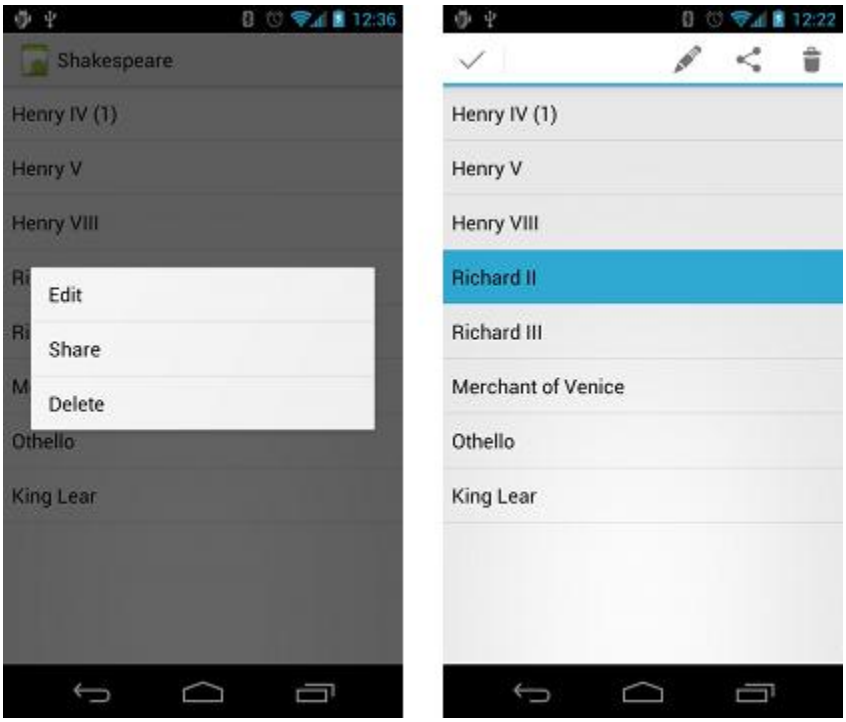

Figure 3. Screenshots of a floating context menu (left) and the contextual action bar (right).

A contextual menu offers actions that affect a specific item or context frame in the UI. You can provide a context menu for any view, but they are most often used for items in a `ListView`, `GridView`, or other view collections in which the user can perform direct actions on each item.

There are two ways to provide contextual actions:

- In a floating context menu. A menu appears as a floating list of menu items (similar to a dialog) when the user performs a long-click (press and hold) on a view that declares support for a context menu. Users can perform a contextual action on one item at a time.
- In the contextual action mode. This mode is a system implementation of `ActionMode` that displays a *contextual action bar* at the top of the screen with action items that affect the selected item(s). When this mode is active, users can perform an action on multiple items at once (if your app allows it).

> **Note:** The contextual action mode is available on Android 3.0 (API level 11) and higher and is the preferred technique for displaying contextual actions when available. If your app supports versions lower than 3.0 then you should fall back to a floating context menu on those devices.

Creating a floating context menu

To provide a floating context menu:

1. Register the `View` to which the context menu should be associated by calling `registerForContextMenu()` and pass it the `View`.

   If your activity uses a `ListView` or `GridView` and you want each item to provide the same context menu, register all items for a context menu by passing the `ListView` or `GridView` to `registerForContextMenu()`.
2. Implement the `onCreateContextMenu()` method in your `Activity` or `Fragment`.

   When the registered view receives a long-click event, the system calls your `onCreateContextMenu()` method. This is where you define the menu items, usually by inflating a menu resource. For example:

```
@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` allows you to inflate the context menu from a menu resource. The callback method parameters include the `View` that the user selected and a `ContextMenu.ContextMenuInfo` object that provides additional information about the item selected. If your activity has several views that each provide a different context menu, you might use these parameters to determine which context menu to inflate.
3. Implement `onContextItemSelected()`.

   When the user selects a menu item, the system calls this method so you can perform the appropriate action. For example:

```
@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 `getItemId()` method queries the ID for the selected menu item, which you should assign to each menu item in XML using the `android:id` attribute, as shown in the section about Defining a Menu in XML.

When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should pass the menu item to the superclass implementation. If your activity includes fragments, the activity receives this callback first. By calling the superclass when unhandled, the system passes the event to the respective callback method in each fragment, one at a time (in the order each fragment was added) until `true` or `false` is returned. (The default implementation for `Activity` and `android.app.Fragment` return`false`, so you should always call the superclass when unhandled.)

Using the contextual action mode

The contextual action mode is a system implementation of `ActionMode` that focuses user interaction toward performing contextual actions. When a user enables this mode by selecting an item, a *contextual action bar* appears at the top of the screen to present actions the user can perform on the currently selected item(s). While this mode is enabled, the user can select multiple items (if you allow it), deselect items, and continue to navigate within the activity (as much as you're willing to allow). The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the BACK button, or selects the *Done* action on the left side of the bar.

> **Note:** The contextual action bar is not necessarily associated with the action bar. They operate independently, even though the contextual action bar visually overtakes the action bar position.

If you're developing for Android 3.0 (API level 11) or higher, you should usually use the contextual action mode to present contextual actions, instead of the floating context menu.

For views that provide contextual actions, you should usually invoke the contextual action mode upon one of two events (or both):

- The user performs a long-click on the view.
- The user selects a checkbox or similar UI component within the view.

How your application invokes the contextual action mode and defines the behavior for each action depends on your design. There are basically two designs:

- For contextual actions on individual, arbitrary views.

- For batch contextual actions on groups of items in a `ListView` or `GridView` (allowing the user to select multiple items and perform an action on them all).

The following sections describe the setup required for each scenario.

Enabling the contextual action mode for individual views

If you want to invoke the contextual action mode only when the user selects specific views, you should:

1. Implement the `ActionMode.Callback` interface. In its callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other lifecycle events for the action mode.
2. Call `startActionMode()` when you want to show the bar (such as when the user long-clicks the view).

For example:

1. Implement the `ActionMode.Callback` interface:

```
private ActionMode.Callback mActionModeCallback = new ActionMode.Callback()
{

    // Called when the action mode is created; startActionMode() was called
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate a menu resource providing context menu items
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
        return true;
    }

    // Called each time the action mode is shown. Always called after
onCreateActionMode, but
    // may be called multiple times if the mode is invalidated.
    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        return false; // Return false if nothing is done
    }

    // Called when the user selects a contextual menu item
    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        switch (item.getItemId()) {
            case R.id.menu_share:
                shareCurrentItem();
```

```
                mode.finish(); // Action picked, so close the CAB
                return true;
            default:
                return false;
        }
    }

    // Called when the user exits the action mode
    @Override
    public void onDestroyActionMode(ActionMode mode) {
        mActionMode = null;
    }
};
```

Notice that these event callbacks are almost exactly the same as the callbacks for the options menu, except each of these also pass the `ActionMode` object associated with the event. You can use `ActionMode` APIs to make various changes to the CAB, such as revise the title and subtitle with `setTitle()` and `setSubtitle()` (useful to indicate how many items are selected).

Also notice that the above sample sets the `mActionMode` variable null when the action mode is destroyed. In the next step, you'll see how it's initialized and how saving the member variable in your activity or fragment can be useful.

2. Call `startActionMode()` to enable the contextual action mode when appropriate, such as in response to a long-click on a `View`:

```
someView.setOnLongClickListener(new View.OnLongClickListener() {
    // Called when the user long-clicks on someView
    public boolean onLongClick(View view) {
        if (mActionMode != null) {
            return false;
        }

        // Start the CAB using the ActionMode.Callback defined above
        mActionMode = getActivity().startActionMode(mActionModeCallback);
        view.setSelected(true);
        return true;
    }
});
```

When you call `startActionMode()`, the system returns the `ActionMode` created. By saving this in a member variable, you can make changes to the contextual action bar in response to other events. In the above sample, the `ActionMode` is used to ensure that the `ActionMode` instance is not recreated if it's already active, by checking whether the member is null before starting the action mode.

Enabling batch contextual actions in a ListView or GridView

If you have a collection of items in a `ListView` or `GridView` (or another extension of `AbsListView`) and want to allow users to perform batch actions, you should:

- Implement the `AbsListView.MultiChoiceModeListener` interface and set it for the view group with `setMultiChoiceModeListener()`. In the listener's callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other callbacks inherited from the `ActionMode.Callback` interface.
- Call `setChoiceMode()` with the `CHOICE_MODE_MULTIPLE_MODAL` argument.

For example:

```
ListView listView = getListView();
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
listView.setMultiChoiceModeListener(new MultiChoiceModeListener() {

    @Override
    public void onItemCheckedStateChanged(ActionMode mode, int position,
                                          long id, boolean checked) {
        // Here you can do something when items are selected/de-selected,
        // such as update the title in the CAB
    }

    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        // Respond to clicks on the actions in the CAB
        switch (item.getItemId()) {
            case R.id.menu_delete:
                deleteSelectedItems();
                mode.finish(); // Action picked, so close the CAB
                return true;
            default:
                return false;
        }
    }

    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate the menu for the CAB
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context, menu);
        return true;
    }

    @Override
    public void onDestroyActionMode(ActionMode mode) {
        // Here you can make any necessary updates to the activity when
        // the CAB is removed. By default, selected items are deselected/unchecked.
    }

    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        // Here you can perform updates to the CAB due to
        // an invalidate() request
        return false;
    }
});
```

That's it. Now when the user selects an item with a long-click, the system calls the `onCreateActionMode()` method and displays the contextual action bar with the specified actions. While the contextual action bar is visible, users can select additional items.

In some cases in which the contextual actions provide common action items, you might want to add a checkbox or a similar UI element that allows users to select items, because they might not discover the long-click behavior. When a user selects the checkbox, you can invoke the contextual action mode by setting the respective list item to the checked state with `setItemChecked()`.

Creating a Popup Menu

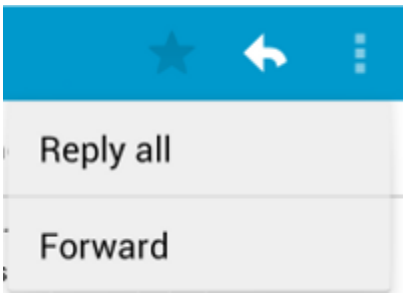

Figure 4. A popup menu in the Gmail app, anchored to the overflow button at the top-right.

A `PopupMenu` is a modal menu anchored to a `View`. It appears below the anchor view if there is room, or above the view otherwise. It's useful for:

- Providing an overflow-style menu for actions that *relate to* specific content (such as Gmail's email headers, shown in figure 4).

> **Note:** This is not the same as a context menu, which is generally for actions that *affect* selected content. For actions that affect selected content, use the contextual action mode or floating context menu.

- Providing a second part of a command sentence (such as a button marked "Add" that produces a popup menu with different "Add" options).
- Providing a drop-down similar to `Spinner` that does not retain a persistent selection.

> **Note:** `PopupMenu` is available with API level 11 and higher.

If you define your menu in XML, here's how you can show the popup menu:

1. Instantate a `PopupMenu` with its constructor, which takes the current application `Context` and the `View` to which the menu should be anchored.
2. Use `MenuInflater` to inflate your menu resource into the `Menu` object returned by `PopupMenu.getMenu()`. On API level 14 and above, you can use `PopupMenu.inflate()` instead.
3. Call `PopupMenu.show()`.

For example, here's a button with the `android:onClick` attribute that shows a popup menu:

```
<ImageButton
    android:layout width="wrap content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic overflow holo dark"
    android:contentDescription="@string/descr_overflow_button"
    android:onClick="showPopup" />
```

The activity can then show the popup menu like this:

```
public void showPopup(View v) {
    PopupMenu popup = new PopupMenu(this, v);
    MenuInflater inflater = popup.getMenuInflater();
    inflater.inflate(R.menu.actions, popup.getMenu());
    popup.show();
}
```

In API level 14 and higher, you can combine the two lines that inflate the menu with `PopupMenu.inflate()`.

The menu is dismissed when the user selects an item or touches outside the menu area. You can listen for the dismiss event using `PopupMenu.OnDismissListener`.

Handling click events

To perform an action when the user selects a menu item, you must implement the `PopupMenu.OnMenuItemClickListener` interface and register it with your `PopupMenu` by calling `setOnMenuItemclickListener()`. When the user selects an item, the system calls the `onMenuItemClick()` callback in your interface.

For example:

```
public void showMenu(View v) {
    PopupMenu popup = new PopupMenu(this, v);

    // This activity implements OnMenuItemClickListener
    popup.setOnMenuItemClickListener(this);
    popup.inflate(R.menu.actions);
    popup.show();
}

@Override
public boolean onMenuItemClick(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.archive:
            archive(item);
            return true;
        case R.id.delete:
            delete(item);
            return true;
        default:
            return false;
    }
}
```

Creating 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/menu_save"
          android:icon="@drawable/menu_save"
          android:title="@string/menu_save" />
    <!-- menu group -->
    <group android:id="@+id/group delete">
        <item android:id="@+id/menu_archive"
              android:title="@string/menu_archive" />
        <item android:id="@+id/menu_delete"
              android:title="@string/menu_delete" />
    </group>
</menu>
```

The items that are in the group appear at the same level as the first item—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. The system will also never separate grouped items. For example, if you declare `android:showAsAction="ifRoom"` for each item, they will either both appear in the action bar or both appear in the action overflow.

Using checkable menu items

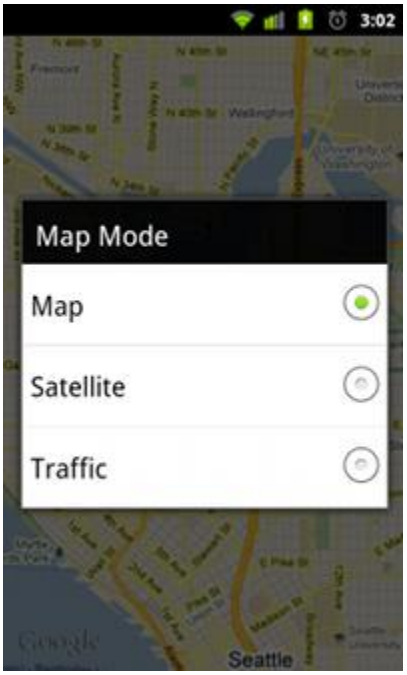

Figure 5. Screenshot of a submenu with checkable items.

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 5 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:

```
<?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.

Adding Menu Items Based on an Intent

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="@string/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.