Action Bar

The action bar is a window feature that identifies the application and user location, and provides user actions and navigation modes. You should use the action bar in most activities that need to prominently present user actions or global navigation, because the action bar offers users a consistent interface across applications and the system gracefully adapts the action bar's appearance for different screen configurations. You can control the behaviors and visibility of the action bar with the `ActionBar` `(/reference/android/app/ActionBar.html)` APIs, which were added in Android 3.0 (API level 11).

The primary goals of the action bar are to:

- Provide a dedicated space for identifying the application brand and user location.

  This is accomplished with the app icon or logo on the left side and the activity title. You might choose to remove the activity title, however, if the current view is identified by a navigation label, such as the currently selected tab.

- Provide consistent navigation and view refinement across different applications.

  The action bar provides built-in tab navigation for switching between fragments (/guide/components/fragments.html). It also offers a drop-down list you can use as an alternative navigation mode or to refine the current view (such as to sort a list by different criteria).

- Make key actions for the activity (such as "search", "create", "share", etc.) prominent and accessible to the user in a predictable way.

  You can provide instant access to key user actions by placing items from the options menu (/guide/topics/ui/menus.html#OptionsMenu) directly in the action bar, as "action items." Action items can also provide an "action view," which provides an embedded widget for even more immediate action behaviors. Menu items that are not promoted to an action item are available in the overflow menu, revealed by either the device *Menu* button (when available) or by an "overflow menu" button in the action bar (when the device does not include a *Menu* button).

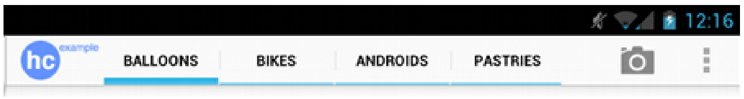

Figure 1. Action bar from the Honeycomb Gallery (/resources/samples/HoneycombGallery/index.html) app (on a landscape handset), showing the logo on the left, navigation tabs, and an action item on the right (plus the overflow menu button).

> **Note:** If you're looking for information about the contextual action bar for displaying contextual action items, see the Menu (/guide/topics/ui/menus.html#context-menu) guide.

> Action Bar Design
>
> For design guidelines, read Android Design's Action Bar (/design/patterns/actionbar.html) guide.

IN THIS DOCUMENT

- Adding the Action Bar
  - Removing the action bar
- Adding Action Items
  - Choosing your action items
  - Using split action bar
- Using the App Icon for Navigation
  - Navigating up
- Adding an Action View
  - Handling collapsible action views
- Adding an Action Provider
  - Using the ShareActionProvider
  - Creating a custom action provider
- Adding Navigation Tabs
- Adding Drop-down Navigation
- Styling the Action Bar
  - General appearance
  - Action items
  - Navigation tabs
  - Drop-down lists
  - Advanced styling

KEY CLASSES

`ActionBar`
`Menu`
`ActionProvider`

RELATED SAMPLES

Honeycomb Gallery
Action Bar Compatibility
API Demos

SEE ALSO

Android Design: Action Bar
Menus
Supporting Tablets and Handsets

Adding the Action Bar

Remaining backward-compatible

Beginning with Android 3.0 (API level 11), the action bar is included in all activities that use the Theme.Holo (/reference/android/R.style.html#Theme_Holo) theme (or one of its descendants), which is the default theme when either the targetSdkVersion (/guide/topics/manifest/uses-sdk-element.html#target) or minSdkVersion (/guide/topics/manifest/uses-sdk-element.html#min) attribute is set to "11" or greater. For example:

```
<manifest ... >
    <uses-sdk android:minSdkVersion="4"
              android:targetSdkVersion="11" />
    ...
</manifest>
```

In this example, the application requires a minimum version of API Level 4 (Android 1.6), but it also targets API level 11 (Android 3.0). This way, when the application runs on Android 3.0 or greater, the system applies the holographic theme to each activity, and thus, each activity includes the action bar.

If you want to provide an action bar in your application *and* remain compatible with versions of Android older than 3.0, you need to create the action bar in your activity's layout (because the ActionBar (/reference/android/app/ActionBar.html) class is not available on older versions).

To help you, the Action Bar Compatibility (/resources/samples/ActionBarCompat/index.html) sample app provides an API layer and action bar layout that allows your app to use some of the ActionBar (/reference/android/app/ActionBar.html) APIs and also support older versions of Android by replacing the traditional title bar with a custom action bar layout.

If you want to use ActionBar (/reference/android/app/ActionBar.html) APIs, such as to add navigation modes and modify action bar styles, you should set the minSdkVersion (/guide/topics/manifest/uses-sdk-element.html#min) to "11" or greater. If you want your app to support older versions of Android, there are ways to use a limited set of ActionBar (/reference/android/app/ActionBar.html) APIs on devices that support API level 11 or higher, while still running on older versions. See the sidebox for information about remaining backward-compatible.

Removing the action bar

If you don't want the action bar for a particular activity, set the activity theme to Theme.Holo.NoActionBar (/reference/android/R.style.html#Theme_Holo_NoActionBar). For example:

```
<activity android:theme="@android:style/Theme.Holo.NoActionBar">
```

You can also hide the action bar at runtime by calling hide() (/reference/android/app/ActionBar.html#hide()). For example:

```
ActionBar actionBar = getActionBar();
actionBar.hide();
```

When the action bar hides, the system adjusts your activity layout to fill all the screen space now available. You can bring the action bar back with show() (/reference/android/app/ActionBar.html#show()).

Beware that hiding and removing the action bar causes your activity to re-layout in order to account for the space consumed by the action bar. If your activity regularly hides and shows the action bar (such as in the Android Gallery app), you might want to use overlay mode. Overlay mode draws the action bar on top of your activity layout rather than in its own area of the screen. This way, your layout remains fixed when the action bar hides and re-appears. To enable overlay mode, create a theme for your activity and set android:windowActionBarOverlay (/reference/android/R.attr.html#windowActionBarOverlay) to true. For more information, see the section about Styling the Action Bar (#Style).

Tip: If you have a custom activity theme in which you'd like to remove the action bar, set the android:windowActionBar (/reference/android/R.styleable.html#Theme_windowActionBar) style property to false. However, if you remove the action bar using a theme, then the window will not allow the action bar at all, so you cannot add it later—calling getActionBar() (/reference/android/app/Activity.html#getActionBar()) will return null.

Adding Action Items

Sometimes you might want to give users immediate access to an item from the options menu (/guide/topics/ui/menus.html#OptionsMenu). To do this, you can declare that the menu item should appear in the action bar as an "action item." An action item can include an icon and/or a text title. If a menu item does not appear as an action item, then the system places it in the overflow menu. The overflow menu is revealed either by the device *Menu* button (if provided by the device) or an additional button in the action bar (if the device does not provide the *Menu* button).

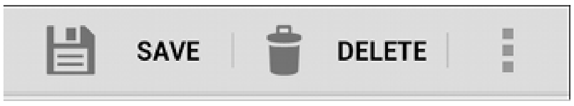

Figure 2. Two action items with icon and text titles, and the overflow menu button.

When the activity first starts, the system populates the action bar and overflow menu by calling `onCreateOptionsMenu()` (/reference/android/app/Activity.html#onCreateOptionsMenu(android.view.Menu)) for your activity. As discussed in the Menus (/guide/topics/ui/menus.html) developer guide, it's in this callback method that you should inflate an XML menu resource (/guide/topics/resources/menu-resource.html) that defines the menu items. For example:

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

In the XML file, you can request a menu item to appear as an action item by declaring `android:showAsAction="ifRoom"` for the `<item>` element. This way, the menu item appears in the action bar for quick access only *if there is room* available. If there's not enough room, the item appears in the overflow menu.

If your menu item supplies both a title and an icon—with the `android:title` and `android:icon` attributes—then the action item shows only the icon by default. If you want to display the text title, add `"withText"` to the `android:showAsAction` attribute. For example:

```
<?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/ic_menu_save"
          android:title="@string/menu_save"
          android:showAsAction="ifRoom|withText" />
</menu>
```

Note: The `"withText"` value is a *hint* to the action bar that the text title should appear. The action bar will show the title when possible, but might not if an icon is available and the action bar is constrained for space.

When the user selects an action item, your activity receives a call to `onOptionsItemSelected()` (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)), passing the ID supplied by the `android:id` attribute—the same callback received for all items in the options menu.

It's important that you always define `android:title` for each menu item—even if you don't declare that the title appear with the action item—for three reasons:

- If there's not enough room in the action bar for the action item, the menu item appears in the overflow menu and only the title appears.
- Screen readers for sight-impaired users read the menu item's title.
- If the action item appears with only the icon, a user can long-press the item to reveal a tool-tip that displays the action item's title.

The `android:icon` is always optional, but recommended. For icon design recommendations, see the Action Bar Icon (/guide/practices/ui_guidelines/icon_design_action_bar.html) design guidelines.

> **Note:** If you added the menu item from a fragment, via the `Fragment` (/reference/android/app/Fragment.html) class's `onCreateOptionsMenu` (/reference/android/app/Fragment.html#onCreateOptionsMenu(android.view.Menu, android.view.MenuInflater)) callback, then the system calls the respective `onOptionsItemSelected()` (/reference/android/app/Fragment.html#onOptionsItemSelected(android.view.MenuItem)) method for that fragment when the user selects one of the fragment's items. However the activity gets a chance to handle the event first, so the system calls `onOptionsItemSelected()` (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) on the activity before calling the same callback for the fragment.

You can also declare an item to *"always"* appear as an action item, instead of being placed in the overflow menu when space is limited. In most cases, you **should not** force an item to appear in the action bar by using the `"always"` value. However, you might need an item to always appear when it provides an action view (#ActionView) that does not offer a default action for the overflow menu. Beware that too many action items can create a cluttered UI and cause layout problems on devices with a narrow screen. It's best to instead use `"ifRoom"` to request that an item appear in the action bar, but allow the system to move it into the overflow menu when there's not enough room.

For more information about creating the options menu that defines your action items, see the Menus (/guide/topics/ui/menus.html#options-menu) developer guide.

Choosing your action items

You should carefully choose which items from your options menu should appear as action items by assessing a few key traits. In general, each action item should be *at least one* of the following:

1. **Frequently used**: It's an action that your users need seven out of ten visits or they use it several times in a row.

   Example frequent actions: "New message" in the Messaging app and "Search" on Google Play.

2. **Important**: It's an action that you need users to easily discover or, if it's not frequently used, it's important that it be effortless to perform in the few cases that users do need it.

   Example important actions: "Add network" in Wi-Fi settings and "Switch to camera" in the Gallery app.

3. **Typical**: It's an action that is typically provided in the action bar in similar apps, so your users expect to find it in yours.

   Example typical actions: "Refresh" in an email or social app, and "New contact" in the People app.

If you believe that more than four of your menu items can be justified as action items, then you should carefully consider their relative level of importance and try to set no more than four as action items (and do so using the `"ifRoom"` value to allow the system to put some back in the overflow menu when space is limited on smaller screens). Even if space is available on a wide screen, you should not create a long stream of action items that clutter the UI and appear like a desktop toolbar, so keep the number of action items to a minimum.

Menu items vs. other app controls

As a general rule, all items in the options menu (/guide/topics/ui/menus.html#OptionsMenu) (let alone action items) should have a global impact on the app, rather than affect only a small portion of the interface. For example, if you have a multi-pane layout and one pane shows a video while another lists all videos, the video player controls should appear within the pane containing the video (not in the action bar), while the action bar might provide action items to share the video or save the video to a favorites list.

So, even before deciding whether a menu item should appear as an action item, be sure that the item has a global scope for the current activity. If it doesn't, then you should place it as a button in the appropriate context of the activity layout.

Additionally, the following actions should never appear as action items: Settings, Help, Feedback, or similar. Always keep them in the overflow menu.

> **Note:** Remember that not all devices provide a dedicated hardware button for Search, so if it's an important feature in your app, it should always appear as an action item (and usually as the first item, especially if you offer it with an action view (#ActionView)).

Using split action bar

When your application is running on Android 4.0 (API level 14) and higher, there's an extra mode available for the action bar called "split action bar." When you enable split action bar, a separate bar appears at the bottom of the screen to display all action items when the activity is running on a narrow screen (such as a portrait-oriented handset). Splitting the action bar to separate the action items ensures that a reasonable amount of space is available to display all your action items on a narrow screen, while leaving room for navigation and title elements at the top.

To enable split action bar, simply add `uiOptions="splitActionBarWhenNarrow"` to your `<activity>` (/guide/topics/manifest/activity-element.html) or `<application>` (/guide/topics/manifest/application-element.html) manifest element.

Be aware that Android adjusts the action bar's appearance in a variety of ways, based on the current screen size. Using split action bar is just one option that you can enable to allow the action bar to further optimize the user experience for different screen sizes. In doing so, you may also allow the action bar to collapse navigation tabs into the main action bar. That is, if you use navigation tabs (#Tabs) in your action bar, once the action items are separated on a narrow screen, the navigation tabs may be able to fit into the main action bar rather than be separated into the "stacked action bar." Specifically, if you've disabled the action bar icon and title (with `setDisplayShowHomeEnabled(false)` `(/reference/android/app/ActionBar.html#setDisplayShowHomeEnabled(boolean))` and `setDisplayShowTitleEnabled(false)` `(/reference/android/app/ActionBar.html#setDisplayShowTitleEnabled(boolean))`), then the navigation tabs collapse into the main action bar, as shown by the second device in figure 3.

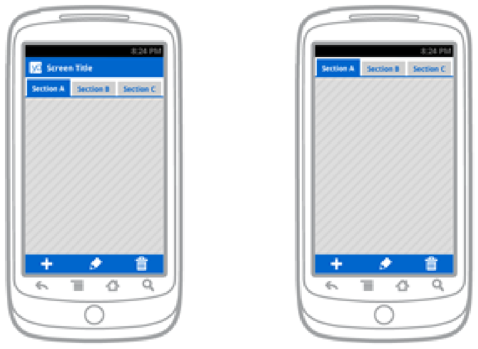

Figure 3. Mock-ups of split action bar with navigation tabs on the left; with the app icon and title disabled on the right.

Note: Although the `android:uiOptions` `(/reference/android/R.attr.html#uiOptions)` attribute was added in Android 4.0 (API level 14), you can safely include it in your application even if your `minSdkVersion` (/guide/topics/manifest/uses-sdk-element.html#min) is set to a value lower than `"14"` to remain compatible with older versions of Android. When running on older versions, the system simply ignores the XML attribute because it doesn't understand it. The only condition to including it in your manifest is that you must compile your application against a platform version that supports API level 14 or higher. Just be sure that you don't openly use other APIs in your application code that aren't supported by the version declared by your `minSdkVersion` (/guide/topics/manifest/uses-sdk-element.html#min) attribute—only XML attributes are safely ignored by older platforms.

Using the App Icon for Navigation

By default, your application icon appears in the action bar on the left side. If you'd like, you can enable the icon to behave as an action item. In response to user action on the icon, your application should do one of two things:

- Go to the application "home" activity, or
- Navigate "up" the application's structural hierarchy

When the user touches the icon, the system calls your activity's

Using a logo instead of icon

By default, the system uses your application icon in the action bar, as specified by the `android:icon` (/guide/topics/manifest/application-element.html#icon) attribute in the `<application>`

onOptionsItemSelected()

(/guide/topics/manifest/application-element.html) or <activity> (/guide/topics/manifest/activity-element.html) element. However, if you also specify the android:logo (/guide/topics/manifest/application-element.html#logo) attribute, then the action bar uses the logo image instead of the icon.

A logo should usually be wider than the icon, but should not include unnecessary text. You should generally use a logo only when it represents your brand in a traditional format that users recognize. A good example is the YouTube app's logo—the logo represents the expected user brand, whereas the app's icon is a modified version that conforms to the square requirement.

(/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) method with the android.R.id.home ID. In response, you should either start the home activity or take the user one step up in your application's structural hierarchy.

If you respond to the application icon by returning to the home activity, you should include the FLAG_ACTIVITY_CLEAR_TOP (/reference/android/content/Intent.html#FLAG_ACTIVITY_CLEAR_TOP) flag in the Intent (/reference/android/content/Intent.html). With this flag, if the activity you're starting already exists in the current task, then all activities on top of it are destroyed and it is brought to the front. Adding this flag is often important because going "home" is an action that's equivalent to "going back" and you should usually not create a new instance of the home activity. Otherwise, you might end up with a long stack of activities in the current task with multiple instances of the home activity.

For example, here's an implementation of onOptionsItemSelected() (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) that returns to the application's "home" activity:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            // app icon in action bar clicked; go home
            Intent intent = new Intent(this, HomeActivity.class);
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

In case the user can enter the current activity from another application, you might also want to add the FLAG_ACTIVITY_NEW_TASK (/reference/android/content/Intent.html#FLAG_ACTIVITY_NEW_TASK) flag. This flag ensures that, when the user navigates either "home" or "up", the new activity is **not** added to the current task, but instead started in a task that belongs to your application. For example, if the user starts an activity in your application through an intent invoked by another application, then selects the action bar icon to navigate home or up, the FLAG_ACTIVITY_CLEAR_TOP (/reference/android/content/Intent.html#FLAG_ACTIVITY_CLEAR_TOP) flag starts the activity in a task that belongs to your application (not the current task). The system either starts a new task with your new activity as the root activity or, if an existing task exists in the background with an instance of that activity, then that task is brought forward and the target activity receives onNewIntent() (/reference/android/app/Activity.html#onNewIntent(android.content.Intent)). So if your activity accepts intents from other applications (it declares any generic intent filters), you should usually add the FLAG_ACTIVITY_NEW_TASK (/reference/android/content/Intent.html#FLAG_ACTIVITY_NEW_TASK) flag to the intent:

```
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_NEW_TASK);
```

For more information about these flags and other back stack behaviors, read the Tasks and Back Stack (/guide/components/tasks-and-back-stack.html) developer guide.

> **Note:** If you're using the icon to navigate to the home activity, beware that beginning with Android 4.0 (API level 14), you must explicitly enable the icon as an action item by calling setHomeButtonEnabled(true) (/reference/android/app/ActionBar.html#setHomeButtonEnabled(boolean)) (in previous versions, the icon was enabled as an action item by default).

Navigating up

As a supplement to traditional "back" navigation—which takes the user to the previous screen in the task history—you can enable the action bar icon to offer "up" navigation, which should take the user one step up in your application's structural hierarchy. For instance, if the current screen is somewhere deep in the hierarchy of the application, touching the app icon should navigate upward one level, to the parent of the current screen.

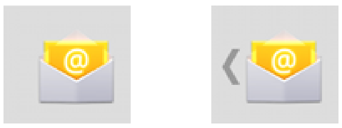

Figure 4. The Email app's standard icon (left) and the "navigate up" icon (right). The system automatically adds the "up" indicator.

For example, figure 5 illustrates how the BACK button behaves when the user navigates from one application to an activity belonging to a different application (specifically, when composing an email to a person selected from the People app).

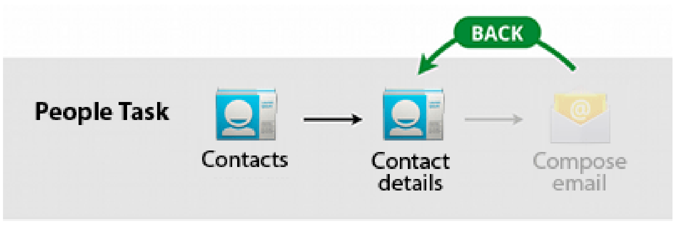

Figure 5. The BACK button behavior after entering the Email app from the People (or Contacts) app.

However, if the user wants to stay within the email application after composing the email, up navigation allows the user to navigate upward in the email application, rather than go back to the previous activity. Figure 6 illustrates this scenario, in which the user again comes into the email application, but presses the action bar icon to navigate up, rather than back.

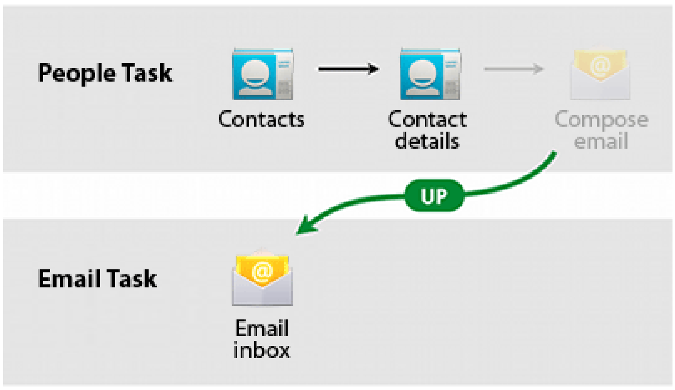

Figure 6. Example behavior for UP navigation after entering the Email app from the People app.

> Navigation Design

> For more about how *Up* and *Back* navigation differ, read Android Design's Navigation (/design/patterns/navigation.html) guide.

To enable the icon for up navigation (which displays the "up" indicator next to the icon), call `setDisplayHomeAsUpEnabled(true)` (/reference/android/app/ActionBar.html#setDisplayHomeAsUpEnabled(boolean)) on your `ActionBar` (/reference/android/app/ActionBar.html):

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
    ActionBar actionBar = getActionBar();
    actionBar.setDisplayHomeAsUpEnabled(true);
    ...
}
```

When the user touches the icon, the system calls your activity's `onOptionsItemSelected()` (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) method with the `android.R.id.home` ID, as shown in the above section about Using the App Icon for Navigation (#Home).

Remember to use the `FLAG_ACTIVITY_CLEAR_TOP` (/reference/android/content/Intent.html#FLAG_ACTIVITY_CLEAR_TOP) flag in the `Intent` (/reference/android/content/Intent.html), so that you don't create a new instance of the parent activity if one already exists. For instance, if you don't use the `FLAG_ACTIVITY_CLEAR_TOP` (/reference/android/content/Intent.html#FLAG_ACTIVITY_CLEAR_TOP) flag, then after navigating up, the BACK button will actually take the user "forward", with respect to the application structure, which would be strange.

> **Note:** If there are many paths that the user could have taken to reach the current activity within your application, the up icon should navigate backward along the path the user actually followed to get to the current activity.

Adding an Action View

An action view is a widget that appears in the action bar as a substitute for an action item's button. For example, if you have an item in the options menu for "Search," you can add an action view that replaces the button with a `SearchView` (/reference/android/widget/SearchView.html) widget, as shown in figure 7.

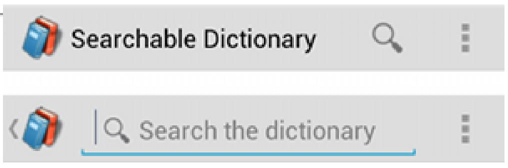

Figure 7. An action bar with a collapsed action view for Search (top), then expanded action view with the `SearchView` (/reference/android/widget/SearchView.html) widget (bottom).

To declare an action view for an item in your menu resource (/guide/topics/resources/menu-resource.html), use either the `android:actionLayout` or `android:actionViewClass` attribute to specify either a layout resource or widget class to use, respectively. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_search"
          android:title="@string/menu_search"
          android:icon="@drawable/ic_menu_search"
          android:showAsAction="ifRoom|collapseActionView"
          android:actionViewClass="android.widget.SearchView" />
</menu>
```

Notice that the `android:showAsAction` attribute also includes `"collapseActionView"`. This is optional and declares that the action view should be collapsed into a button. When the user selects the button, the action view expands. Otherwise, the action view is visible by default and might consume valuable action bar space even when the user is not using it. For more information, see the next section about Handling collapsible action views (#ActionViewCollapsing).

If you need to add some event hooks to your action view, you can do so during the `onCreateOptionsMenu()` `(/reference/android/app/Activity.html#onCreateOptionsMenu(android.view.Menu))` callback. You can acquire elements in an action view by calling `findItem()` `(/reference/android/view/Menu.html#findItem(int))` with the ID of the menu item, then call `getActionView()` `(/reference/android/view/MenuItem.html#getActionView())`. For example, the search widget from the above sample is acquired like this:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);
    SearchView searchView = (SearchView) menu.findItem(R.id.menu_search).getActionViev
    // Configure the search info and add any event listeners
    ...
    return super.onCreateOptionsMenu(menu);
}
```

For more information about using the search widget, see Creating a Search Interface (/guide/topics/search/search-dialog.html).

Handling collapsible action views

Action views allow you to provide fast access to rich actions without changing activities or fragments, or replacing the action bar. However, it might not be appropriate to make an action view visible by default. To preserve the action bar space (especially when running on smaller screens), you can collapse your action view into an action item button. When the user selects the button, the action view appears in the action bar. When collapsed, the system might place the item into the overflow menu if you've defined `android:showAsAction` with `"ifRoom"`, but the action view still appears in the action bar when the user selects the item. You can make your action view collapsible by adding `"collapseActionView"` to the `android:showAsAction` attribute, as shown in the XML above.

Supporting Android 3.0 with an action view

The `"collapseActionView"` option was added with Android 4.0 (API level 14). However, if your application supports older versions, you should still declare `"collapseActionView"` in order to better support smaller screens. Devices running Android 4.0 and higher will show the action view collapsed, while older versions work as designed otherwise.

Adding this value requires that you set your build target to Android 4.0 or higher in order to compile. Older versions of Android ignore the `"collapseActionView"` value because they don't understand it. Just be sure not to use other APIs in your source code that are not supported in the version declared by your `minSdkVersion` (/guide/topics/manifest/uses-sdk-element.html#min), unless you add the appropriate version check at runtime.

Because the system will expand the action view when the user selects the item, you *do not* need to respond to the item in the `onOptionsItemSelected` `(/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem))` callback. The system still calls `onOptionsItemSelected()` `(/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem))` when the user selects it, but the system will always expand the action view unless you return `true` (indicating you've handled the event instead).

The system also collapses your action view when the user selects the "up" icon in the action bar or presses the

BACK button.

If necessary, you can expand or collapse the action view in your own code by calling expandActionView() (/reference/android/view/MenuItem.html#expandActionView()) and collapseActionView() (/reference/android/view/MenuItem.html#collapseActionView()) on the MenuItem (/reference/android/view/MenuItem.html).

> **Note:** Although collapsing your action view is optional, we recommend that you always collapse your action view if it includes SearchView (/reference/android/widget/SearchView.html). Also be aware that some devices provide a dedicated SEARCH button and you should expand your search action view if the user presses the SEARCH button. Simply override your activity's onKeyUp() (/reference/android/app/Activity.html#onKeyUp(int, android.view.KeyEvent)) callback method, listen for the KEYCODE_SEARCH (/reference/android/view/KeyEvent.html#KEYCODE_SEARCH) event, then call expandActionView() (/reference/android/view/MenuItem.html#expandActionView()).

If you need to update your activity based on the visibility of your action view, you can receive callbacks when it's expanded and collapsed by defining an OnActionExpandListener (/reference/android/view/MenuItem.OnActionExpandListener.html) and registering it with setOnActionExpandListener() (/reference/android/view/MenuItem.html#setOnActionExpandListener(android.view.MenuItem.OnActionExpandListener)). For example:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);
    MenuItem menuItem = menu.findItem(R.id.actionItem);
    ...

    menuItem.setOnActionExpandListener(new OnActionExpandListener() {
        @Override
        public boolean onMenuItemActionCollapse(MenuItem item) {
            // Do something when collapsed
            return true;  // Return true to collapse action view
        }

        @Override
        public boolean onMenuItemActionExpand(MenuItem item) {
            // Do something when expanded
            return true;  // Return true to expand action view
        }
    });
}
```

Adding an Action Provider

Similar to an action view (#ActionView), an action provider (defined by the ActionProvider (/reference/android/view/ActionProvider.html) class) replaces an action item with a customized layout, but it also takes control of all the item's behaviors. When you declare an action provider for a menu item in the action bar, it not only controls the appearance of the item in the action bar with a custom layout, but also handles the default event for the menu item when it appears in the overflow menu. It can also provide a submenu from either the action bar or the overflow menu.

For example, the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) is an extension of ActionProvider (/reference/android/view/ActionProvider.html) that facilitates a “share” action by showing a list of available share targets from the action bar. Instead of using a traditional action item that invokes the ACTION_SEND (/reference/android/content/Intent.html#ACTION_SEND) intent, you can declare an instance of ShareActionProvider (/reference/android/widget/ShareActionProvider.html) to handle an action item. This action provider presents an action view with a drop-down list of applications that handle the ACTION_SEND (/reference/android/content/Intent.html#ACTION_SEND) intent, even when the menu item appears in the overflow menu. Hence, when you use an action provider such as this one, you don't have to handle user events on the menu item.

To declare an action provider for an action item, define the android:actionProviderClass attribute for the appropriate the <item> element in your menu resource (/guide/topics/resources/menu-resource.html), using the fully-qualified class name of the action provider. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/an
    <item android:id="@+id/menu_share"
          android:title="@string/share"
          android:showAsAction="ifRoom"
          android:actionProviderClass="android.widget.Shar
    ...
</menu>
```

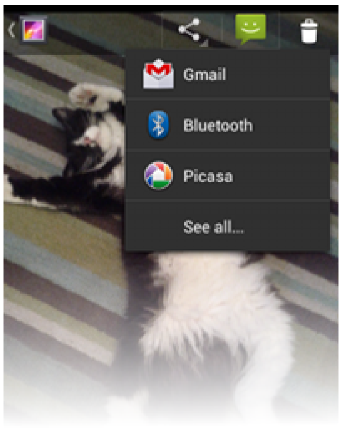

Figure 8. Screenshot from the Gallery app, with the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) submenu expanded to show share targets.

In this example, the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) is used as the action provider. At this point, the action provider officially takes control of the menu item and handles both its appearance and behavior in the action bar and its behavior in the overflow menu. You must still provide a text title for the item to be used in the overflow menu.

Although the action provider can perform the default action for the menu item when it appears in the overflow menu, your activity (or fragment) can override that behavior by also handling the click event from the onOptionsItemSelected() (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) callback method. If you do not handle the event in that callback, then the action provider receives the onPerformDefaultAction() (/reference/android/view/ActionProvider.html#onPerformDefaultAction()) callback to handle the event. However, if the action provider provides a submenu, then your activity will not receive the onOptionsItemSelected() (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) callback, because the submenu is shown instead of invoking the default menu item behavior when selected.

Using the ShareActionProvider

If you want to provide a "share" action in your action bar by leveraging other applications installed on the device (for example, to share a photo using a messaging or social app), then using ShareActionProvider (/reference/android/widget/ShareActionProvider.html) is an effective way to do so, rather than adding an action item that invokes the ACTION_SEND (/reference/android/content/Intent.html#ACTION_SEND) intent. When you use ShareActionProvider (/reference/android/widget/ShareActionProvider.html) for an action item, it presents an action view with a drop-down list of applications that handle the ACTION_SEND (/reference/android/content/Intent.html#ACTION_SEND) intent (as shown in figure 8).

All the logic for creating the submenu, populating it with share targets, and handling click events (including when the item appears in the overflow menu) is implemented by the ShareActionProvider (/reference/android/widget/ShareActionProvider.html)—the only code you need to write is to declare the action provider for the menu item and specify the share intent.

By default, the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) retains a ranking for each share target based on how often the user selects each one. The share targets used more frequently appear at the top of the drop-down list and the target used most often appears directly in the action bar as the default share target. By default, the ranking information is saved in a private file with a name specified by DEFAULT_SHARE_HISTORY_FILE_NAME (/reference/android/widget/ShareActionProvider.html#DEFAULT_SHARE_HISTORY_FILE_NAME). If you use the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) or an extension of it for only one type of action, then you should continue to use this default history file and there's nothing you need to do. However, if you use ShareActionProvider (/reference/android/widget/ShareActionProvider.html) or an extension of it for multiple actions with semantically different meanings, then each ShareActionProvider (/reference/android/widget/ShareActionProvider.html) should specify its own history file in order to maintain its own history. To specify a different history file for the ShareActionProvider (/reference/android/widget/ShareActionProvider.html), call setShareHistoryFileName() (/reference/android/widget/ShareActionProvider.html#setShareHistoryFileName(java.lang.String)) and provide an XML file name (for example, "custom_share_history.xml").

> **Note:** Although the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) ranks share targets based on frequency of use, the behavior is extensible and extensions of ShareActionProvider (/reference/android/widget/ShareActionProvider.html) can perform different behaviors and ranking based on the history file (if appropriate).

To add ShareActionProvider (/reference/android/widget/ShareActionProvider.html), simply define the android:actionProviderClass attribute with "android.widget.ShareActionProvider", as shown in the XML example above. The only thing left to do is define the Intent (/reference/android/content/Intent.html) you want to use for sharing. To do so, you must call getActionProvider() (/reference/android/view/MenuItem.html#getActionProvider()) to retrieve the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) that's associated with a MenuItem (/reference/android/view/MenuItem.html), then call setShareIntent() (/reference/android/widget/ShareActionProvider.html#setShareIntent(android.content.Intent)).

If the format for the share intent depends on the selected item or other variables that change during the activity lifecycle, you should save the ShareActionProvider (/reference/android/widget/ShareActionProvider.html) in a member field and update it by calling setShareIntent() (/reference/android/widget/ShareActionProvider.html#setShareIntent(android.content.Intent)) as necessary. For example:

```
private ShareActionProvider mShareActionProvider;
...

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    mShareActionProvider = (ShareActionProvider) menu.findItem(R.id.menu_share).getAct

    // If you use more than one ShareActionProvider, each for a different action,
    // use the following line to specify a unique history file for each one.
    // mShareActionProvider.setShareHistoryFileName("custom_share_history.xml");

    // Set the default share intent
    mShareActionProvider.setShareIntent(getDefaultShareIntent());

    return true;
}
// When you need to update the share intent somewhere else in the app, call
// mShareActionProvider.setShareIntent()
```

The ShareActionProvider (/reference/android/widget/ShareActionProvider.html) now handles all user interaction with the item and you *do not* need to handle click events from the onOptionsItemSelected() (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) callback method.

For a sample using the share action provider, see ActionBarShareActionProviderActivity (/resources/samples/ApiDemos/src/com/example/android/apis/app/ActionBarShareActionProviderActivity.html).

Creating a custom action provider

When you want to create an action view that has dynamic behaviors and a default action in the overflow menu, extending ActionProvider (/reference/android/view/ActionProvider.html) to define those behaviors is a good solution. Creating your own action provider offers you an organized and reusable component, rather than handling the various action item transformations and behaviors in your fragment or activity code. As shown in the previous section, Android provides one implementation of `ActionProvider` (/reference/android/view/ActionProvider.html) for share actions: the `ShareActionProvider` (/reference/android/widget/ShareActionProvider.html).

To create your own, simply extend the `ActionProvider` (/reference/android/view/ActionProvider.html) class and implement its callback methods as appropriate. Most importantly, you should implement the following:

`ActionProvider()`
: This constructor passes you the application `Context`, which you should save in a member field to use in the other callback methods.

`onCreateActionView()`
: This is where you define the action view for the item. Use the `Context` acquired from the constructor to instantiate a `LayoutInflater` and inflate your action view layout from an XML resource, then hook up event listeners. For example:

```
public View onCreateActionView() {
    // Inflate the action view to be shown on the action bar.
    LayoutInflater layoutInflater = LayoutInflater.from(mContext);
    View view = layoutInflater.inflate(R.layout.action_provider, null);
    ImageButton button = (ImageButton) view.findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // Do something...
        }
    });
    return view;
}
```

`onPerformDefaultAction()`
: The system calls this when the menu item is selected from the overflow menu and the action provider should perform a default action for the menu item.

  However, if your action provider provides a submenu, through the `onPrepareSubMenu()` (/reference/android/view/ActionProvider.html#onPrepareSubMenu(android.view.SubMenu)) callback, then the submenu appears even when the menu item is in the overflow menu. Thus, `onPerformDefaultAction()` (/reference/android/view/ActionProvider.html#onPerformDefaultAction()) is never called when there is a submenu.

> **Note:** An activity or a fragment that implements `onOptionsItemSelected()` (/reference/android/app/Activity.html#onOptionsItemSelected(android.view.MenuItem)) can override the action provider's default behavior by handling the item-selected event (and returning true), in which case, the system does not call `onPerformDefaultAction()` (/reference/android/view/ActionProvider.html#onPerformDefaultAction()).

For an example extension of `ActionProvider` (/reference/android/view/ActionProvider.html), see ActionBarSettingsActionProviderActivity (/resources/samples/ApiDemos/src/com/example/android/apis/app/ActionBarSettingsActionProviderActivity.html).

Adding Navigation Tabs

When you want to provide navigation tabs in an activity, using the action bar's tabs is a great option (instead of using `TabWidget`

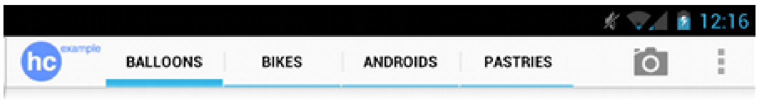

Figure 9. Screenshot of action bar tabs from the Honeycomb Gallery

(/resources/samples/HoneycombGallery/index.html) app.

(/reference/android/widget/TabWidget.html)), because the system adapts the action bar tabs for different screen sizes—placing them in the main action bar when the screen is sufficiently wide, or in a separate bar (known as the "stacked action bar") when the screen is too narrow, as shown in figures 9 and 10.

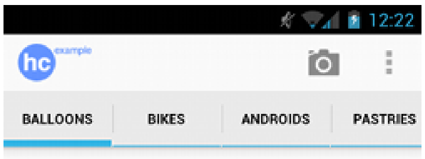

Figure 10. Screenshot of tabs in the stacked action bar on a narrow screen.

To switch between fragments using the tabs, you must perform a fragment transaction each time a tab is selected. If you're not familiar with how to change fragments using `FragmentTransaction` (/reference/android/app/FragmentTransaction.html), first read the Fragments (/guide/components/fragments.html) developer guide.

To get started, your layout must include a `ViewGroup` (/reference/android/view/ViewGroup.html) in which you place each `Fragment` (/reference/android/app/Fragment.html) associated with a tab. Be sure the `ViewGroup` (/reference/android/view/ViewGroup.html) has a resource ID so you can reference it from your tab-swapping code. Alternatively, if the tab content will fill the activity layout (excluding the action bar), then your activity doesn't need a layout at all (you don't even need to call `setContentView()` (/reference/android/app/Activity.html#setContentView(android.view.View))). Instead, you can place each fragment in the default root `ViewGroup` (/reference/android/view/ViewGroup.html), which you can refer to with the `android.R.id.content` ID (you can see this ID used in the sample code below, during fragment transactions).

Once you determine where the fragments appear in the layout, the basic procedure to add tabs is:

1. Implement the `ActionBar.TabListener` interface. Callbacks in this interface respond to user events on the tabs so you can swap fragments.
2. For each tab you want to add, instantiate an `ActionBar.Tab` and set the `ActionBar.TabListener` by calling `setTabListener()`. Also set the tab's title and/or icon with `setText()` and/or `setIcon()`.
3. Add each tab to the action bar by calling `addTab()`.

When looking at the `ActionBar.TabListener` (/reference/android/app/ActionBar.TabListener.html) interface, notice that the callback methods provide only the `ActionBar.Tab` (/reference/android/app/ActionBar.Tab.html) that was selected and a `FragmentTransaction` (/reference/android/app/FragmentTransaction.html) for you to perform fragment transactions—it doesn't say anything about what fragment you should swap in or out. Thus, you must define your own association between each `ActionBar.Tab` (/reference/android/app/ActionBar.Tab.html) and the appropriate `Fragment` (/reference/android/app/Fragment.html) that it represents (in order to perform the appropriate fragment transaction). There are several ways you can define the association, depending on your design. In the example below, the `ActionBar.TabListener` (/reference/android/app/ActionBar.TabListener.html) implementation provides a constructor such that each new tab uses its own instance of the listener. Each instance of the listener defines several fields that are necessary to later perform a transaction on the appropriate fragment.

For example, here's how you might implement the `ActionBar.TabListener` (/reference/android/app/ActionBar.TabListener.html) such that each tab uses its own instance of the listener:

```
public static class TabListener<T extends Fragment> implements ActionBar.TabListener {
    private Fragment mFragment;
    private final Activity mActivity;
    private final String mTag;
    private final Class<T> mClass;

    /** Constructor used each time a new tab is created.
      * @param activity  The host Activity, used to instantiate the fragment
      * @param tag  The identifier tag for the fragment
      * @param clz  The fragment's Class, used to instantiate the fragment
      */
    public TabListener(Activity activity, String tag, Class<T> clz) {
        mActivity = activity;
        mTag = tag;
        mClass = clz;
    }

    /* The following are each of the ActionBar.TabListener callbacks */

    public void onTabSelected(Tab tab, FragmentTransaction ft) {
        // Check if the fragment is already initialized
        if (mFragment == null) {
            // If not, instantiate and add it to the activity
            mFragment = Fragment.instantiate(mActivity, mClass.getName());
            ft.add(android.R.id.content, mFragment, mTag);
        } else {
            // If it exists, simply attach it in order to show it
            ft.attach(mFragment);
        }
    }

    public void onTabUnselected(Tab tab, FragmentTransaction ft) {
        if (mFragment != null) {
            // Detach the fragment, because another one is being attached
            ft.detach(mFragment);
        }
    }

    public void onTabReselected(Tab tab, FragmentTransaction ft) {
        // User selected the already selected tab. Usually do nothing.
    }
}
```

Caution: You **must not** call commit() (/reference/android/app/FragmentTransaction.html#commit()) for the fragment transaction in each of these callbacks—the system calls it for you and it may throw an exception if you call it yourself. You also **cannot** add these fragment transactions to the back stack.

In this example, the listener simply attaches (attach() (/reference/android/app/FragmentTransaction.html#attach(android.app.Fragment))) a fragment to the activity layout—or if not instantiated, creates the fragment and adds (add() (/reference/android/app/FragmentTransaction.html#add(android.app.Fragment, java.lang.String))) it to the layout (as a child of the `android.R.id.content` view group)—when the respective tab is selected, and detaches (detach() (/reference/android/app/FragmentTransaction.html#detach(android.app.Fragment))) it when the tab is unselected.

The ActionBar.TabListener (/reference/android/app/ActionBar.TabListener.html) implementation is the bulk of the work. All that remains is to create each ActionBar.Tab (/reference/android/app/ActionBar.Tab.html) and add it to the ActionBar (/reference/android/app/ActionBar.html). Additionally, you must call setNavigationMode(NAVIGATION_MODE_TABS) (/reference/android/app/ActionBar.html#setNavigationMode(int)) to make the tabs visible. You might also want to disable the activity title by calling setDisplayShowTitleEnabled(false) (/reference/android/app/ActionBar.html#setDisplayShowTitleEnabled(boolean)) if the tab titles actually indicate the current view.

For example, the following code adds two tabs using the listener defined above:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Notice that setContentView() is not used, because we use the root
    // android.R.id.content as the container for each fragment

    // setup action bar for tabs
    ActionBar actionBar = getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
    actionBar.setDisplayShowTitleEnabled(false);

    Tab tab = actionBar.newTab()
            .setText(R.string.artist)
            .setTabListener(new TabListener<ArtistFragment>(
                    this, "artist", ArtistFragment.class));
    actionBar.addTab(tab);

    tab = actionBar.newTab()
        .setText(R.string.album)
        .setTabListener(new TabListener<AlbumFragment>(
                this, "album", AlbumFragment.class));
    actionBar.addTab(tab);
}
```

> **Note:** The above implementation for `ActionBar.TabListener` (/reference/android/app/ActionBar.TabListener.html) is one of several possible techniques. You can see more of this style in the API Demos (/resources/samples/ApiDemos/src/com/example/android/apis/app/FragmentTabs.html) app.

If your activity stops, you should retain the currently selected tab with the saved instance state (/guide/components/activities.html#SavingActivityState) so you can open the appropriate tab when the user returns. When it's time to save the state, you can query the currently selected tab with `getSelectedNavigationIndex()` (/reference/android/app/ActionBar.html#getSelectedNavigationIndex()). This returns the index position of the selected tab.

> **Caution:** It's important that you save the state of each fragment as necessary, so that when users switch fragments with the tabs and then return to a previous fragment, it looks the way it did when they left. For information about saving the state of your fragment, see the Fragments (/guide/components/fragments.html) developer guide.

> **Note:** In some cases, the Android system will show your action bar tabs as a drop-down list in order to ensure the best fit in the action bar.

Adding Drop-down Navigation

As another mode of navigation (or filtering) within your activity, the action bar offers a built in drop-down list. For example, the drop-down list can offer different modes by which content in the activity is sorted.

The basic procedure to enable drop-down navigation is:

1. Create a `SpinnerAdapter` that provides the list of selectable items for the drop-down and the layout to use when drawing each item in the list.
2. Implement `ActionBar.OnNavigationListener` to define the behavior that occurs when the user selects an item from the list.
3. Enable navigation mode for the action bar with `setNavigationMode()`. For example:

   ```
   ActionBar actionBar = getActionBar();
   actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
   ```

   > **Note:** You should perform this during your activity's `onCreate()` (/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method.

4. Set the callback for the drop-down list with `setListNavigationCallbacks()`. For example:

```
actionBar.setListNavigationCallbacks(mSpinnerAdapter, mNavigationCallback);
```

This method takes your `SpinnerAdapter` (/reference/android/widget/SpinnerAdapter.html) and `ActionBar.OnNavigationListener` (/reference/android/app/ActionBar.OnNavigationListener.html).

That's the basic setup. However, implementing the `SpinnerAdapter` (/reference/android/widget/SpinnerAdapter.html) and `ActionBar.OnNavigationListener` (/reference/android/app/ActionBar.OnNavigationListener.html) is where most of the work is done. There are many ways you can implement these to define the functionality for your drop-down navigation and implementing various types of `SpinnerAdapter` (/reference/android/widget/SpinnerAdapter.html) is beyond the scope of this document (you should refer to the `SpinnerAdapter` (/reference/android/widget/SpinnerAdapter.html) class reference for more information). However, below is a simple example for a `SpinnerAdapter` (/reference/android/widget/SpinnerAdapter.html) and `ActionBar.OnNavigationListener` (/reference/android/app/ActionBar.OnNavigationListener.html) to get you started (click the title to reveal the sample).

▸ **Example SpinnerAdapter and OnNavigationListener**

Styling the Action Bar

If you've implemented a custom design for the widgets in your application, you might also want to redesign some of the action bar to match your app design. To do so, you need to use Android's style and theme (/guide/topics/ui/themes.html) framework to restyle the action bar using special style properties.

> **Note:** In order for background images to change appearance depending on the current button state (selected, pressed, unselected), the drawable resource you use must be a state list drawable (/guide/topics/resources/drawable-resource.html#StateList).

> **Caution:** For all background drawables you provide, be sure to use Nine-Patch drawables (/guide/topics/graphics/2d-graphics.html#nine-patch) to allow stretching. The Nine-Patch image should be *smaller* than 40px tall and 30px wide (for the mdpi asset).

General appearance

`android:windowActionBarOverlay`

Declares whether the action bar should overlay the activity layout rather than offset the activity's layout position (for example, the Gallery app uses overlay mode). This is `false` by default.

Normally, the action bar requires its own space on the screen and your activity layout fills in what's left over. When the action bar is in overlay mode, your activity layout uses all the available space and the system draws the action bar on top. Overlay mode can be useful if you want your content to keep a fixed size and position when the action bar is hidden and shown. You might also like to use it purely as a visual effect, because you can use a semi-transparent background for the action bar so the user can still see some of your activity layout behind the action bar.

> **Note:** The `Holo` (/reference/android/R.style.html#Theme_Holo) theme families draw the action bar with a semi-transparent background by default. However, you can modify it with your own styles and the `DeviceDefault` (/reference/android/R.style.html#Theme_DeviceDefault) theme on different devices might use an opaque background by default.

When overlay mode is enabled, your activity layout has no awareness of the action bar laying on top of it. So, you must be careful not to place any important information or UI components in the area overlayed by the action bar. If appropriate, you can refer to the platform's value for `actionBarSize` (/reference/android/R.attr.html#actionBarSize) to determine the height of the action bar, by referencing it in your XML layout. For example:

```
<SomeView
```

```
    ...
    android:layout_marginTop="?android:attr/actionBarSize" />
```

You can also retrieve the action bar height at runtime with getHeight() (/reference/android/app/ActionBar.html#getHeight()). This reflects the height of the action bar at the time it's called, which might not include the stacked action bar (due to navigation tabs) if called during early activity lifecycle methods. To see how you can determine the total height at runtime, including the stacked action bar, see the TitlesFragment (/resources/samples/HoneycombGallery/src/com/example/android/hcgallery/TitlesFragment.html) class in the Honeycomb Gallery (/resources/samples/HoneycombGallery/index.html) sample app.

Action items

`android:actionButtonStyle`
: Defines a style resource for the action item buttons.

`android:actionBarItemBackground`
: Defines a drawable resource for each action item's background. (Added in API level 14.)

`android:itemBackground`
: Defines a drawable resource for each overflow menu item's background.

`android:actionBarDivider`
: Defines a drawable resource for the divider between action items. (Added in API level 14.)

`android:actionMenuTextColor`
: Defines a color for text that appears in an action item.

`android:actionMenuTextAppearance`
: Defines a style resource for text that appears in an action item.

`android:actionBarWidgetTheme`
: Defines a theme resource for widgets that are inflated into the action bar as action views. (Added in API level 14.)

Navigation tabs

`android:actionBarTabStyle`
: Defines a style resource for tabs in the action bar.

`android:actionBarTabBarStyle`
: Defines a style resource for the thin bar that appears below the navigation tabs.

`android:actionBarTabTextStyle`
: Defines a style resource for text in the navigation tabs.

Drop-down lists

`android:actionDropDownStyle`
: Defines a style for the drop-down navigation (such as the background and text styles).

For example, here's a file that defines a few custom styles for the action bar:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActivityTheme" parent="@android:style/Theme.Holo">
        <item name="android:actionBarTabTextStyle">@style/CustomTabTextStyle</item>
        <item name="android:actionBarDivider">@drawable/ab_divider</item>
        <item name="android:actionBarItemBackground">@drawable/ab_item_background</ite
    </style>

    <!-- style for the action bar tab text -->
    <style name="CustomTabTextStyle" parent="@android:style/TextAppearance.Holo">
        <item name="android:textColor">#2456c2</item>
    </style>
</resources>
```

> **Note:** Be certain that your theme declares a parent theme in the `<style>` tag, from which it inherits all styles not explicitly declared by your theme. When modifying the action bar, using a parent theme is important so that you can simply override the action bar styles you want to change without re-implementing the styles you want to leave alone (such as text appearance or padding in action items).

You can apply your custom theme to the entire application or to individual activities in your manifest file like this:

```
<application android:theme="@style/CustomActivityTheme" ... />
```

For more information about using style and theme resources in your application, read Styles and Themes (/guide/topics/ui/themes.html).

Advanced styling

If you need more advanced styling for the action bar than is available with the properties above, you can include `android:actionBarStyle` (/reference/android/R.attr.html#actionBarStyle) and `android:actionBarSplitStyle` (/reference/android/R.attr.html#actionBarSplitStyle) in your activity's theme. Each of these specifies another style that can define various properties for the action bar, including different backgrounds with `android:background` (/reference/android/R.attr.html#background), `android:backgroundSplit` (/reference/android/R.attr.html#backgroundSplit), and `android:backgroundStacked` (/reference/android/R.attr.html#backgroundStacked). If you override these action bar styles, be sure that you define a parent action bar style such as `Widget.Holo.ActionBar` (/reference/android/R.style.html#Widget_Holo_ActionBar).

For example, if you want to change the action bar's background, you can use the following styles:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActivityTheme" parent="@android:style/Theme.Holo">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <!-- other activity and action bar styles here -->
    </style>

    <!-- style for the action bar backgrounds -->
    <style name="MyActionBar" parent="@android:style/Widget.Holo.ActionBar">
        <item name="android:background">@drawable/ab_background</item>
        <item name="android:backgroundStacked">@drawable/ab_background</item>
        <item name="android:backgroundSplit">@drawable/ab_split_background</item>
    </style>
</resources>
```