App Components

User Interface

Overview

Layouts

Input Controls

Input Events

Menus

Action Bar

Settings

Dialogs

Notifications

Toasts

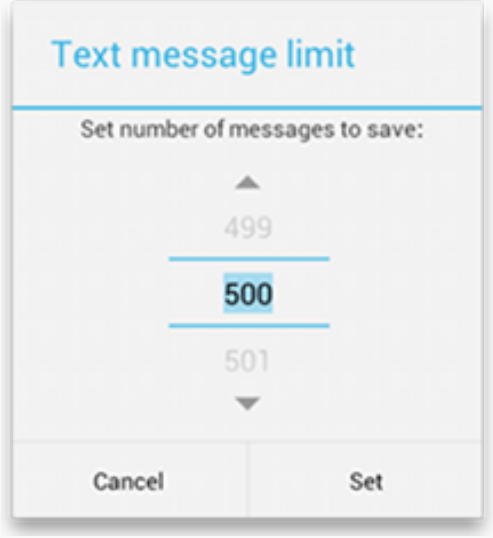
Search

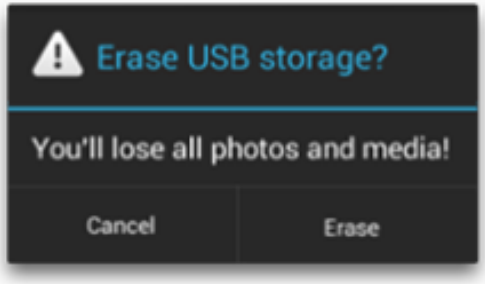
Drag and Drop

Dialogs

A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.

> **Dialog Design**
>
> For information about how to design your dialogs, including recommendations for language, read the Dialogs design guide.





The `Dialog` class is the base class for dialogs, but you should avoid

IN THIS DOCUMENT

Creating a Dialog Fragment
Building an Alert Dialog
Adding buttons
Adding a list
Creating a Custom Layout
Passing Events Back to the Dialog's Host
Showing a Dialog
Showing a Dialog Fullscreen or as an Embedded Fragment
Showing an activity as a dialog on large screens
Dismissing a Dialog

KEY CLASSES

`DialogFragment`
`AlertDialog`

SEE ALSO

instantiating `Dialog` directly. Instead, use one of the following subclasses:

`AlertDialog`
: A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.

`DatePickerDialog` or `TimePickerDialog`
: A dialog with a pre-defined UI that allows the user to select a date or time.

These classes define the style and structure for your dialog, but you should use a `DialogFragment` as a container for your dialog. The `DialogFragment` class provides all the controls you need to create your dialog and manage its appearance, instead of calling methods on the `Dialog` object.

Using `DialogFragment` to manage the dialog ensures that it correctly handles lifecycle events such as when the user presses the *Back* button or rotates the screen. The `DialogFragment` class also allows you to reuse the dialog's UI as an embeddable component in a larger UI, just like a traditional `Fragment` (such as when you want the dialog UI to appear differently on large and small screens).

The following sections in this guide describe how to use a `DialogFragment` in combination with an `AlertDialog` object. If you'd like to create a date or time picker, you should instead read the Pickers guide.

> **Note:** Because the `DialogFragment` class was originally added with Android 3.0 (API level 11), this document describes how to use the `DialogFragment` class that's provided with the Support Library. By adding this library to your app, you can use `DialogFragment` and a variety of other APIs on devices running Android 1.6 or higher. If the minimum version your app supports is API level 11 or higher, then you can use the framework version of `DialogFragment`, but be aware that the links in this document are for the support library APIs. When using the support library, be sure that you import `android.support.v4.app.DialogFragment` class and *not* `android.app.DialogFragment`.

Dialogs design guide

Pickers (Date/Time dialogs)

Avoid ProgressDialog

Android includes another dialog class called `ProgressDialog` that shows a dialog with a progress bar. However, if you need to indicate loading or indeterminate progress, you should instead follow the design guidelines for Progress & Activity and use a `ProgressBar` in your layout.

Creating a Dialog Fragment

You can accomplish a wide variety of dialog designs—including custom layouts and those described in the Dialogs design guide—by extending `DialogFragment` and creating a `AlertDialog` in the `onCreateDialog()` callback method.

For example, here's a basic `AlertDialog` that's managed within a `DialogFragment`:

```
public class FireMissilesDialogFragment extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the Builder class for convenient dialog construction
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
               .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener()
                   public void onClick(DialogInterface dialog, int id) {
                       // FIRE ZE MISSILES!
                   }
               })
               .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener
                   public void onClick(DialogInterface dialog, int id) {
                       // User cancelled the dialog
                   }
               });
        // Create the AlertDialog object and return it
        return builder.create();
    }
}
```

Now, when you create an instance of this class and call

`show()` on that object, the dialog appears as shown in figure 1.

Figure 1. A dialog with a message and two action buttons.

The next section describes more about using the `AlertDialog.Builder` APIs to create the dialog.

Depending on how complex your dialog is, you can implement a variety of other callback methods in the `DialogFragment`, including all the basic fragment lifecycle methods.

Building an Alert Dialog

The `AlertDialog` class allows you to build a variety of dialog designs and is often the only dialog class you'll need. As shown in figure 2, there are three regions of an alert dialog:

1. **Title**

   This is optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout. If you need to state a simple message or question (such as the dialog in figure 1), you don't need a title.

2. **Content area**

   This can display a message, a list, or other custom layout.

3. **Action buttons**

   There should be no more than three action buttons in a dialog.

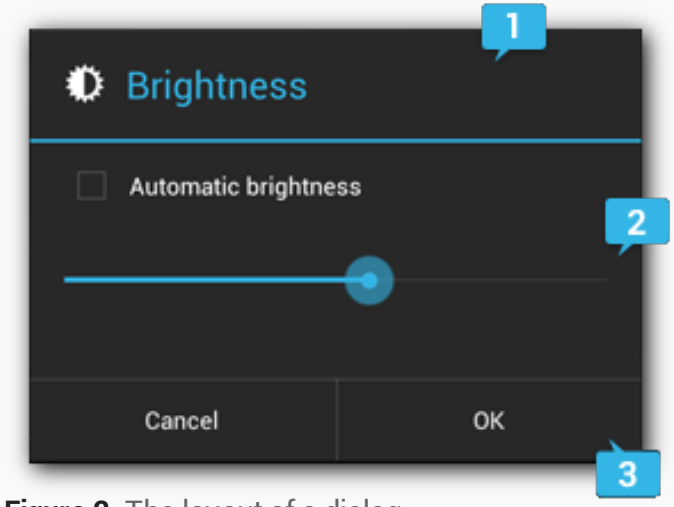


Figure 2. The layout of a dialog.

The `AlertDialog.Builder` class provides APIs that allow you to create an `AlertDialog` with these kinds of content, including a custom layout.

To build an `AlertDialog`:

```
// 1. Instantiate an AlertDialog.Builder with its constructor
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

// 2. Chain together various setter methods to set the dialog characteristics
builder.setMessage(R.string.dialog_message)
       .setTitle(R.string.dialog_title);

// 3. Get the AlertDialog from create()
AlertDialog dialog = builder.create();
```

The following topics show how to define various dialog attributes using the **`AlertDialog.Builder`** class.

Adding buttons

To add action buttons like those in figure 2, call the **`setPositiveButton()`** and **`setNegativeButton()`** methods:

```
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
// Add the buttons
builder.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
               // User clicked OK button
           }
       });
builder.setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
               // User cancelled the dialog
           }
       });
// Set other dialog properties
...
```

```
// Create the AlertDialog
AlertDialog dialog = builder.create();
```

The `set...Button()` methods require a title for the button (supplied by a string resource) and a `DialogInterface.OnClickListener` that defines the action to take when the user presses the button.

There are three different action buttons you can add:

Positive
: You should use this to accept and continue with the action (the "OK" action).

Negative
: You should use this to cancel the action.

Neutral
: You should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."

You can add only one of each button type to an `AlertDialog`. That is, you cannot have more than one "positive" button.

Adding a list

There are three kinds of lists available with the `AlertDialog` APIs:

- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)

To create a single-choice list like the one in figure 3, use the `setItems()` method:

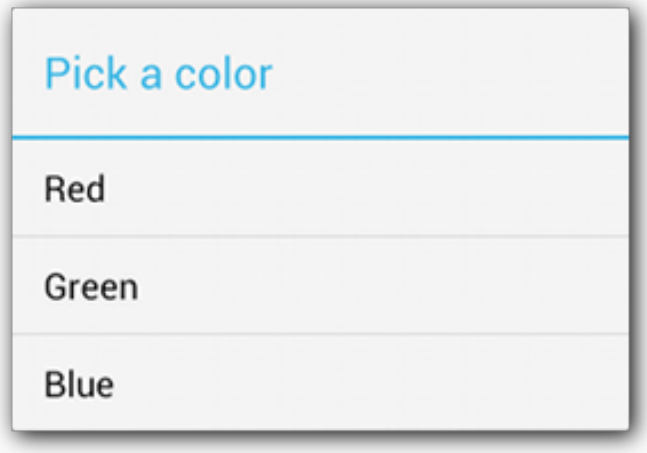


Figure 3. A dialog with a title and list.

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    builder.setTitle(R.string.pick_color);
           .setItems(R.array.colors_array, new DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int which) {
               // The 'which' argument contains the index position
               // of the selected item
           }
    });
    return builder.create();
}
```

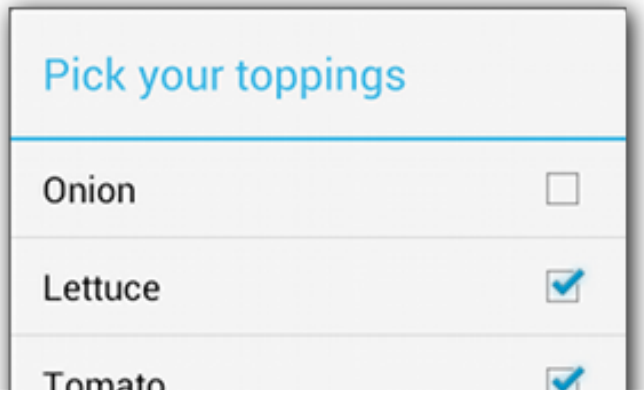
Because the list appears in the dialog's content area, the dialog cannot show both a message and a list and you should set a title for the dialog with **`setTitle()`**. To specify the items for the list, call **`setItems()`**, passing an array. Alternatively, you can specify a list using **`setAdapter()`**. This allows you to back the list with dynamic data (such as from a database) using a **`ListAdapter`**.

If you choose to back your list with a **`ListAdapter`**, always use a **`Loader`** so that the content loads asynchronously. This is described further in Building Layouts with an Adapter and the Loaders guide.

> **Note:** By default, touching a list item dismisses the dialog, unless you're using one of the following persistent choice lists.

Adding a persistent multiple-choice or single-choice list

To add a list of multiple-choice items (checkboxes) or single-choice items (radio buttons), use the **`setMultiChoiceItems()`** or **`setSingleChoiceItems()`** methods, respectively.



For example, here's how you can create a multiple-choice list like the one shown in figure 4 that saves the selected items in an `ArrayList`:

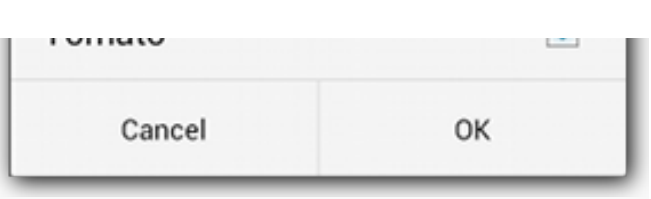


Figure 4. A list of multiple-choice items.

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    mSelectedItems = new ArrayList();  // Where we track the selected items
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    // Set the dialog title
    builder.setTitle(R.string.pick_toppings)
    // Specify the list array, the items to be selected by default (null for none),
    // and the listener through which to receive callbacks when items are selected
           .setMultiChoiceItems(R.array.toppings, null,
                      new DialogInterface.OnMultiChoiceClickListener() {
               @Override
               public void onClick(DialogInterface dialog, int which,
                       boolean isChecked) {
                   if (isChecked) {
                       // If the user checked the item, add it to the selected items
                       mSelectedItems.add(which);
                   } else if (mSelectedItems.contains(which)) {
                       // Else, if the item is already in the array, remove it
                       mSelectedItems.remove(Integer.valueOf(which));
                   }
               }
           })
    // Set the action buttons
           .setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {
               @Override
```

```
                public void onClick(DialogInterface dialog, int id) {
                    // User clicked OK, so save the mSelectedItems results somewhere
                    // or return them to the component that opened the dialog
                    ...
                }
            })
            .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int id) {
                    ...
                }
            });

    return builder.create();
}
```

Although both a traditional list and a list with radio buttons provide a "single choice" action, you should use `setSingleChoiceItems()` if you want to persist the user's choice. That is, if opening the dialog again later should indicate what the user's current choice is, then you create a list with radio buttons.

Creating a Custom Layout

If you want a custom layout in a dialog, create a layout and add it to an `AlertDialog` by calling `setView()` on your `AlertDialog.Builder` object.

By default, the custom layout fills the dialog window, but you can still use `AlertDialog.Builder` methods to add buttons and a title.

For example, here's the layout file for the dialog in Figure 5:

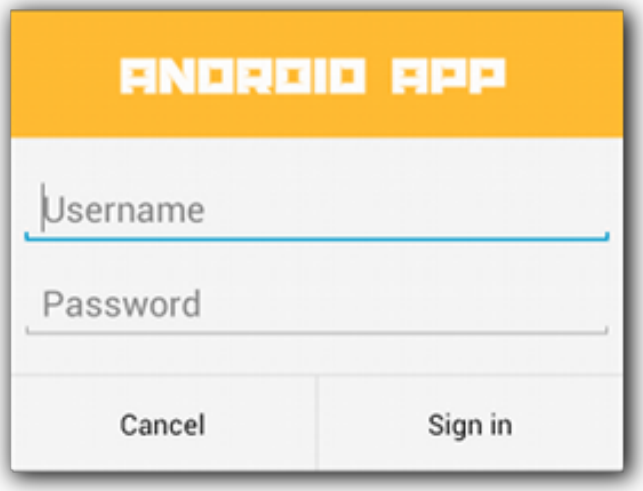


Figure 5. A custom dialog layout.

res/layout/dialog_signin.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
        android:src="@drawable/header_logo"
        android:layout_width="match_parent"
        android:layout_height="64dp"
        android:scaleType="center"
        android:background="#FFFFBB33"
        android:contentDescription="@string/app_name" />
    <EditText
        android:id="@+id/username"
        android:inputType="textEmailAddress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="4dp"
        android:hint="@string/username" />
    <EditText
        android:id="@+id/password"
        android:inputType="textPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="4dp"
        android:layout_marginLeft="4dp"
```

```
        android:layout_marginRight="4dp"
        android:layout_marginBottom="16dp"
        android:fontFamily="sans-serif"
        android:hint="@string/password"/>
</LinearLayout>
```

> **Tip:** By default, when you set an `EditText` element to use the `"textPassword"` input type, the font family is set to monospace, so you should change its font family to `"sans-serif"` so that both text fields use a matching font style.

To inflate the layout in your `DialogFragment`, get a `LayoutInflater` with `getLayoutInflater()` and call `inflate()`, where the first parameter is the layout resource ID and the second parameter is a parent view for the layout. You can then call `setView()` to place the layout in the dialog.

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    // Get the layout inflater
    LayoutInflater inflater = getActivity().getLayoutInflater();

    // Inflate and set the layout for the dialog
    // Pass null as the parent view because its going in the dialog layout
    builder.setView(inflater.inflate(R.layout.dialog_signin, null))
    // Add action buttons
           .setPositiveButton(R.string.signin, new DialogInterface.OnClickListener() {
               @Override
               public void onClick(DialogInterface dialog, int id) {
                   // sign in the user ...
               }
           })
           .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
```

```
                public void onClick(DialogInterface dialog, int id) {
                    LoginDialogFragment.this.getDialog().cancel();
                }
            });
    return builder.create();
}
```

> **Tip:** If you want a custom dialog, you can instead display an `Activity` as a dialog instead of using the `Dialog` APIs. Simply create an activity and set its theme to `Theme.Holo.Dialog` in the `<activity>` manifest element:
>
> ```
> <activity android:theme="@android:style/Theme.Holo.Dialog" >
> ```
>
> That's it. The activity now displays in a dialog window instead of fullscreen.

Passing Events Back to the Dialog's Host

When the user touches one of the dialog's action buttons or selects an item from its list, your `DialogFragment` might perform the necessary action itself, but often you'll want to deliver the event to the activity or fragment that opened the dialog. To do this, define an interface with a method for each type of click event. Then implement that interface in the host component that will receive the action events from the dialog.

For example, here's a `DialogFragment` that defines an interface through which it delivers the events back to the host activity:

```
public class NoticeDialogFragment extends DialogFragment {

    /* The activity that creates an instance of this dialog fragment must
     * implement this interface in order to receive event callbacks.
     * Each method passes the DialogFragment in case the host needs to query it. */
```

```
    public interface NoticeDialogListener {
        public void onDialogPositiveClick(DialogFragment dialog);
        public void onDialogNegativeClick(DialogFragment dialog);
    }

    // Use this instance of the interface to deliver action events
    NoticeDialogListener mListener;

    // Override the Fragment.onAttach() method to instantiate the NoticeDialogListener
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // Verify that the host activity implements the callback interface
        try {
            // Instantiate the NoticeDialogListener so we can send events to the host
            mListener = (NoticeDialogListener) activity;
        } catch (ClassCastException e) {
            // The activity doesn't implement the interface, throw exception
            throw new ClassCastException(activity.toString()
                    + " must implement NoticeDialogListener");
        }
    }
    ...
}
```

The activity hosting the dialog creates an instance of the dialog with the dialog fragment's constructor and receives the dialog's events through an implementation of the **`NoticeDialogListener`** interface:

```
public class MainActivity extends FragmentActivity
                          implements NoticeDialogFragment.NoticeDialogListener{
    ...
```

```
    public void showNoticeDialog() {
        // Create an instance of the dialog fragment and show it
        DialogFragment dialog = new NoticeDialogFragment();
        dialog.show(getSupportFragmentManager(), "NoticeDialogFragment");
    }

    // The dialog fragment receives a reference to this Activity through the
    // Fragment.onAttach() callback, which it uses to call the following methods
    // defined by the NoticeDialogFragment.NoticeDialogListener interface
    @Override
    public void onDialogPositiveClick(DialogFragment dialog) {
        // User touched the dialog's positive button
        ...
    }

    @Override
    public void onDialogNegativeClick(DialogFragment dialog) {
        // User touched the dialog's negative button
        ...
    }
}
```

Because the host activity implements the **`NoticeDialogListener`**—which is enforced by the **`onAttach()`** callback method shown above—the dialog fragment can use the interface callback methods to deliver click events to the activity:

```
public class NoticeDialogFragment extends DialogFragment {
    ...

    @Override
```

```
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Build the dialog and set up the button click handlers
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
               .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener()
                   public void onClick(DialogInterface dialog, int id) {
                       // Send the positive button event back to the host activity
                       mListener.onDialogPositiveClick(NoticeDialogFragment.this);
                   }
               })
               .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener
                   public void onClick(DialogInterface dialog, int id) {
                       // Send the negative button event back to the host activity
                       mListener.onDialogPositiveClick(NoticeDialogFragment.this);
                   }
               });
        return builder.create();
    }
}
```

Showing a Dialog

When you want to show your dialog, create an instance of your **`DialogFragment`** and call **`show()`**, passing the **`FragmentManager`** and a tag name for the dialog fragment.

You can get the **`FragmentManager`** by calling **`getSupportFragmentManager()`** from the **`FragmentActivity`** or **`getFragmentManager()`** from a **`Fragment`**. For example:

```
public void confirmFireMissiles() {
    DialogFragment newFragment = new FireMissilesDialogFragment();
```

```
    newFragment.show(getSupportFragmentManager(), "missiles");
}
```

The second argument, **`"missiles"`**, is a unique tag name that the system uses to save and restore the fragment state when necessary. The tag also allows you to get a handle to the fragment by calling **`findFragmentByTag()`**.

Showing a Dialog Fullscreen or as an Embedded Fragment

You might have a UI design in which you want a piece of the UI to appear as a dialog in some situations, but as a full screen or embedded fragment in others (perhaps depending on whether the device is a large screen or small screen). The **`DialogFragment`** class offers you this flexibility because it can still behave as an embeddable **`Fragment`**.

However, you cannot use **`AlertDialog.Builder`** or other **`Dialog`** objects to build the dialog in this case. If you want the **`DialogFragment`** to be embeddable, you must define the dialog's UI in a layout, then load the layout in the **`onCreateView()`** callback.

Here's an example **`DialogFragment`** that can appear as either a dialog or an embeddable fragment (using a layout named **`purchase_items.xml`**):

```
public class CustomDialogFragment extends DialogFragment {
    /** The system calls this to get the DialogFragment's layout, regardless
        of whether it's being displayed as a dialog or an embedded fragment. */
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        // Inflate the layout to use as dialog or embedded fragment
        return inflater.inflate(R.layout.purchase_items, container, false);
    }
```

```
    /** The system calls this only when creating the layout in a dialog. */
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // The only reason you might override this method when using onCreateView() is
        // to modify any dialog characteristics. For example, the dialog includes a
        // title by default, but your custom layout might not need it. So here you can
        // remove the dialog title, but you must call the superclass to get the Dialog
        Dialog dialog = super.onCreateDialog(savedInstanceState);
        dialog.requestWindowFeature(Window.FEATURE_NO_TITLE);
        return dialog;
    }
}
```

And here's some code that decides whether to show the fragment as a dialog or a fullscreen UI, based on the screen size:

```
public void showDialog() {
    FragmentManager fragmentManager = getSupportFragmentManager();
    CustomDialogFragment newFragment = new CustomDialogFragment();

    if (mIsLargeLayout) {
        // The device is using a large layout, so show the fragment as a dialog
        newFragment.show(fragmentManager, "dialog");
    } else {
        // The device is smaller, so show the fragment fullscreen
        FragmentTransaction transaction = fragmentManager.beginTransaction();
        // For a little polish, specify a transition animation
        transaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN);
        // To make it fullscreen, use the 'content' root view as the container
        // for the fragment, which is always the root view for the activity
        transaction.add(android.R.id.content, newFragment)
```

```
                    .addToBackStack(null).commit();
        }
    }
```

For more information about performing fragment transactions, see the Fragments guide.

In this example, the **`mIsLargeLayout`** boolean specifies whether the current device should use the app's large layout design (and thus show this fragment as a dialog, rather than fullscreen). The best way to set this kind of boolean is to declare a bool resource value with an alternative resource value for different screen sizes. For example, here are two versions of the bool resource for different screen sizes:

res/values/bools.xml

```
<!-- Default boolean values -->
<resources>
    <bool name="large_layout">false</bool>
</resources>
```

res/values-large/bools.xml

```
<!-- Large screen boolean values -->
<resources>
    <bool name="large_layout">true</bool>
</resources>
```

Then you can initialize the **`mIsLargeLayout`** value during the activity's **`onCreate()`** method:

```
boolean mIsLargeLayout;

@Override
public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mIsLargeLayout = getResources().getBoolean(R.bool.large_layout);
    }
```

Showing an activity as a dialog on large screens

Instead of showing a dialog as a fullscreen UI when on small screens, you can accomplish the same result by showing an **`Activity`** as a dialog when on large screens. Which approach you choose depends on your app design, but showing an activity as a dialog is often useful when your app is already designed for small screens and you'd like to improve the experience on tablets by showing a short-lived activity as a dialog.

To show an activity as a dialog only when on large screens, apply the **`Theme.Holo.DialogWhenLarge`** theme to the **`<activity>`** manifest element:

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

For more information about styling your activities with themes, see the Styles and Themes guide.

Dismissing a Dialog

When the user touches any of the action buttons created with an **`AlertDialog.Builder`**, the system dismisses the dialog for you.

The system also dismisses the dialog when the user touches an item in a dialog list, except when the list uses radio buttons or checkboxes. Otherwise, you can manually dismiss your dialog by calling **`dismiss()`** on your **`DialogFragment`**.

In case you need to perform certain actions when the dialog goes away, you can implement the **`onDismiss()`** method in your **`DialogFragment`**.

You can also *cancel* a dialog. This is a special event that indicates the user explicitly left the dialog without completing the task. This occurs if the user presses the *Back* button, touches the screen outside the dialog area, or if you explicitly call `cancel()` on the `Dialog` (such as in response to a "Cancel" button in the dialog).

As shown in the example above, you can respond to the cancel event by implementing `onCancel()` in your `DialogFragment` class.

> **Note:** The system calls `onDismiss()` upon each event that invokes the `onCancel()` callback. However, if you call `Dialog.dismiss()` or `DialogFragment.dismiss()`, the system calls `onDismiss()` *but not* `onCancel()`. So you should generally call `dismiss()` when the user presses the *positive* button in your dialog in order to remove the dialog from view.

PREVIOUS NEXT