Working with Dialogs

Android application user interfaces need to be elegant and easy to use. One important technique developers can use is to implement dialogs to inform the user or allow the user to perform actions such as edits without redrawing the main screen. In this text, we discuss how to incorporate dialogs into your applications.

Choosing Your Dialog Implementation

The Android platform is growing and changing quickly. New revisions of the Android SDK are released on a frequent basis. This means that developers are always struggling to keep up with the latest that Android has to offer. Right now the Android platform is in a period of transition from a traditional smartphone platform to a "smart-device" platform that will support a much wider variety of devices, such as tablets, TVs, and toasters. To this end, one of the most important additions to the platform is the concept of the fragment. One area of application design that has received an overhaul during this transition is the way in which dialogs are implemented.

So what does this mean for developers? It means that there are now two methods for incorporating dialogs into your application—the legacy method and the method recommended for developers moving forward:

- Using the legacy method, which has existed since the first Android SDK was released, an `Activity` class manages its dialogs in a dialog pool. Dialogs are created, initialized, updated, and destroyed using `Activity` class callback methods. Dialogs are not shared among activities. This type of dialog implementation works for all versions of the Android platform; however, many of the methods used in this type of solution have been deprecated as of API Level 13 (Android 3.2). If your application `Activity` classes are not using, and do not plan to use, the `Fragment` APIs, then this method may be the most straightforward to implement, despite being deprecated and perhaps not receiving future support, bug fixes, or exhaustive testing.

The source code that accompanies this chapter is available for download on the publisher website: www.informit.com/title/9780321813831.

- Using the new fragment-based method, which was introduced in API Level 11 (Android 3.0), dialogs are managed using the `FragmentManager` class (`android.app.FragmentManager`). Dialogs become a special type of `Fragment` that must still be used within the scope of an `Activity` class, but its lifecycle is managed like any other `Fragment`. This type of dialog implementation works with the newest versions of the Android platform, but is not backward compatible with older devices unless you incorporate the latest Android Support Package into your application to gain access to these new classes for use with older Android SDKs. However, this is likely the best choice for moving forward with the Android platform.

Note

Unlike with some other platforms, which routinely remove deprecated methods after a few releases, deprecated methods within the Android SDK can normally be used safely for the foreseeable future, as necessary. That said, developers should understand the ramifications of using deprecated methods and techniques, which may include difficulty in upgrading application functionality to use the latest SDK features later on, slower performance as newer features are streamlined and legacy ones are left as is, and the possibility of the application "showing its age." Deprecated methods are also unlikely to receive any sort of fixes or updates.

We will cover both methods in this chapter, given that most applications on the Android Market today are still using the deprecated method. If you're maintaining legacy applications, you'll need to understand this method. However, if you are developing new applications or updating existing applications to use the latest technologies the Android SDK has to offer, we highly recommend implementing the newer `Fragment`-based method and using the Android Support Package to support older versions of the Android platform.

Exploring the Different Types of Dialogs

Regardless of which way you implement them, a number of different dialog types are available within the Android SDK. Each type has a special function that most users should be somewhat familiar with. The dialog types available as part of the Android SDK include the following:

- **`Dialog`**: The basic class for all `Dialog` types. A basic `Dialog` (`android.app.Dialog`) is shown in the top left of Figure 1.
- **`AlertDialog`**: A `Dialog` with one, two, or three `Button` controls. An `AlertDialog` (`android.app.AlertDialog`) is shown in the top center of Figure 1.
- **`CharacterPickerDialog`**: A `Dialog` for choosing an accented character associated with a base character. A `CharacterPickerDialog`

(`android.text.method.CharacterPickerDialog`) is shown in the top right of Figure 1.

- **`DatePickerDialog`**: A `Dialog` with a `DatePicker` control. A `DatePickerDialog` (`android.app.DatePickerDialog`) is shown in the bottom left of Figure 1.
- **`ProgressDialog`**: A `Dialog` with a determinate or indeterminate `ProgressBar` control. An indeterminate `ProgressDialog` (`android.app.ProgressDialog`) is shown in the bottom center of Figure 1.
- **`TimePickerDialog`**: A `Dialog` with a `TimePicker` control. A `TimePickerDialog` (`android.app.TimePickerDialog`) is shown in the bottom right of Figure 1.

If none of the existing `Dialog` types is adequate, you can also create custom `Dialog` windows, with your specific layout requirements.

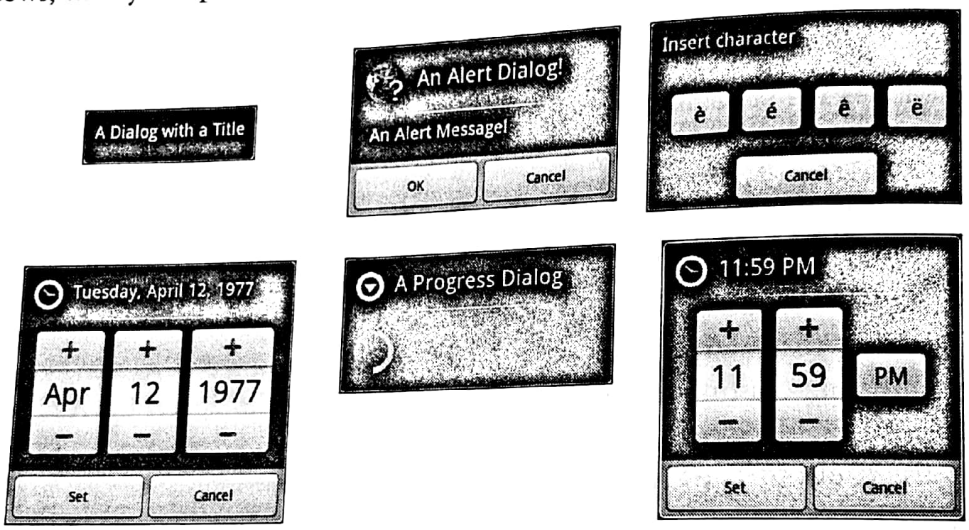

Figure 1 The different dialog types available in Android.

Working with Dialogs: The Legacy Method

An `Activity` can use dialogs to organize information and react to user-driven events. For example, an activity might display a dialog informing the user of a problem or asking the user to confirm an action such as deleting a data record. Using dialogs for simple tasks helps keep the number of application activities manageable.

Tip

Many of the code examples provided in this section are taken from the SimpleDialogs application. The source code for the SimpleDialogs application is provided for download on the book's website.

Tracing the Lifecycle of a Dialog

Each Dialog must be defined within the Activity in which it is used. A Dialog may be launched once, or used repeatedly. Understanding how an Activity manages the Dialog lifecycle is important to implementing a Dialog correctly. Let's look at the key methods that an Activity must use to manage a Dialog:

- The showDialog() method is used to display a Dialog.
- The dismissDialog() method is used to stop showing a Dialog. The Dialog is kept around in the Activity's Dialog pool. If the Dialog is shown again using showDialog(), the cached version is displayed once more.
- The removeDialog() method is used to remove a Dialog from the Activity object's Dialog pool. The Dialog is no longer kept around for future use. If you call showDialog() again, the Dialog must be re-created.

Adding the Dialog to an Activity involves several steps:

1. Define a unique identifier for the Dialog within the Activity.
2. Implement the onCreateDialog() method of the Activity to return a Dialog of the appropriate type, when supplied the unique identifier.
3. Implement the onPrepareDialog() method of the Activity to initialize the Dialog as appropriate.
4. Launch the Dialog using the showDialog() method with the unique identifier.

Defining a Dialog

A Dialog used and managed by an Activity must be defined in advance. Each Dialog has a special identifier (an integer). When the showDialog() method is called, you pass in this identifier. At this point, the onCreateDialog() method is called and must return a Dialog of the appropriate type.

It is up to the developer to override the onCreateDialog() method of the Activity and return the appropriate Dialog for a given identifier. If an Activity has multiple Dialog windows, the onCreateDialog() method generally contains a switch statement to return the appropriate Dialog based on the incoming parameter—the Dialog identifier.

Initializing a Dialog

Because a Dialog is often kept around by the Activity in its Dialog pool, it might be important to reinitialize a Dialog each time it is shown, instead of just when it is created the first time. For this purpose, you can override the onPrepareDialog() method of the Activity.

Whereas the onCreateDialog() method may only be called once for initial Dialog creation, the onPrepareDialog() method is called each time the showDialog() method is called, giving the Activity a chance to modify the Dialog before it is shown to the user.

Launching a Dialog

You can display any dialog defined within an `Activity` by calling the `showDialog()` method of the `Activity` class and passing it a valid `Dialog` object identifier—one that will be recognized by the `onCreateDialog()` method.

Dismissing a Dialog

Most types of dialogs have automatic dismissal circumstances. However, if you want to force a `Dialog` to be dismissed, simply call the `dismissDialog()` method and pass in the `Dialog` identifier.

Removing a Dialog from Use

Dismissing a `Dialog` does not destroy it. If the `Dialog` is shown again, its cached contents are redisplayed. If you want to force an `Activity` to remove a `Dialog` from its pool and not use it again, you can call the `removeDialog()` method, passing in the valid `Dialog` identifier. Although not generally needed, a single-use, but resource heavy, dialog may provide some benefit when being removed.

Here's an example of a simple class called `SimpleDialogsActivity` that illustrates how to implement a simple `Dialog` control that is launched when a `Button` called `Button_AlertDialog` (defined in a layout resource) is clicked:

```
public class SimpleDialogsActivity extends Activity {

    static final int ALERT_DIALOG_ID = 1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Handle Alert Dialog Button
        Button launchAlertDialog = (Button) findViewById(
          R.id.Button_AlertDialog);
        launchAlertDialog.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialog(ALERT_DIALOG_ID);
            }
        });
    }

@Override
protected Dialog onCreateDialog(int id) {
    switch (id) {

        case ALERT_DIALOG_ID:
```

```
            AlertDialog.Builder alertDialog = new
                  AlertDialog.Builder(this);
            alertDialog.setTitle("Alert Dialog");
                alertDialog.setMessage("You have been alerted.");
                alertDialog.setIcon(android.R.drawable.btn_star);
            alertDialog.setPositiveButton(android.R.string.ok,
                new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog,
                int which) {
                    Toast.makeText(getApplicationContext(),
                        "Clicked OK!", Toast.LENGTH_SHORT).show();
                    return;
                }
            });
                return alertDialog.create();
        }
    return null;
    }

    @Override
    protected void onPrepareDialog(int id, Dialog dialog) {
        super.onPrepareDialog(id, dialog);
        switch (id) {
        case ALERT_DIALOG_ID:
            // No extra configuration needed
            return;
        }
    }
}
```

The full implementation of this `AlertDialog`, as well as many other types of dialogs, can be found in the sample code provided on the book's website.

Working with Custom Dialogs

When the dialog types do not suit your purpose exactly, you can create a custom `Dialog`. One easy way to create a custom `Dialog` is to begin with an `AlertDialog` and use an `AlertDialog.Builder` class to override its default layout. In order to create a custom `Dialog` this way, the following steps must be performed:

1. Design a custom layout resource to display in the `AlertDialog`.
2. Define the custom `Dialog` identifier in the `Activity`.
3. Update the `Activity`'s `onCreateDialog()` method to build and return the appropriate custom `AlertDialog`. You should use a `LayoutInflater` to inflate the custom layout resource for the `Dialog`.
4. Launch the `Dialog` using the `showDialog()` method.

Working with Dialogs: The Fragment Method

Moving forward, most `Activity` classes should be "fragment aware." This means that you'll want to decouple your `Dialog` management from the `Activity` and move it into the realm of fragments. There is a special subclass of `Fragment` called a `DialogFragment` (`android.app.DialogFragment`) that can be used for this purpose.

Tip

Many of the code examples provided in this section are taken from the SimpleFragDialog application. The source code for the SimpleFragDialog application is provided for download on the book's website.

Let's look at a quick example of how you might implement a simple `AlertDialog` that behaves much like the legacy `AlertDialog` discussed earlier in this text. In order to show an advantage of using the new fragment-based dialog technique, we pass some data to the dialog that demonstrates multiple instances of the `DialogFragment` class running within a single `Activity`. To begin, you need to implement your own `DialogFragment` class. This class simply needs to be able to return an instance of the object that is fully configured and needs to implement the `onCreateDialog` method, which returns the fully configured `AlertDialog`, much as it did using the legacy method. The following code is a full implementation of a simple `DialogFragment` that manages an `AlertDialog`:

```
public class MyAlertDialogFragment extends DialogFragment {

    public static MyAlertDialogFragment
        newInstance(String fragmentNumber) {
        MyAlertDialogFragment newInstance = new MyAlertDialogFragment();
        Bundle args = new Bundle();
        args.putString("fragnum", fragmentNumber);
        newInstance.setArguments(args);
        return newInstance;
    }

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {

        final String fragNum = getArguments().getString("fragnum");

        AlertDialog.Builder alertDialog = new AlertDialog.Builder(
            getActivity());
        alertDialog.setTitle("Alert Dialog");
        alertDialog.setMessage("This alert brought to you by "
            + fragNum );
        alertDialog.setIcon(android.R.drawable.btn_star);
```

```
        alertDialog.setPositiveButton(android.R.string.ok,
            new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                ((SimpleFragDialogActivity) getActivity())
                    .doPositiveClick(fragNum);
                return;
            }
        });
        return alertDialog.create();
    }
}
```

Now that you have defined your `DialogFragment`, you can use it within your `Activity` much as you would any fragment—by using the `FragmentManager`. The following `Activity` class, called `SimpleFragDialogActivity`, has a layout resource that contains two `Button` controls, each of which triggers a new instance of the `MyAlertDialogFragment` to be generated and shown. The `show()` method of the `DialogFragment` is used to display the dialog, adding the fragment to the `FragmentManager` and passing in a little bit of information to configure the specific instance of the `DialogFragment` and its internal `AlertDialog`.

```
public class SimpleFragDialogActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Handle Alert Dialog Button
        Button launchAlertDialog = (Button) findViewById(
            R.id.Button_AlertDialog);
        launchAlertDialog.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialogFragment("Fragment Instance One");
            }
        });

        // Handle Alert Dialog 2 Button
        Button launchAlertDialog2 = (Button) findViewById(
            R.id.Button_AlertDialog2);
        launchAlertDialog2.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialogFragment("Fragment Instance Two");
            }
        });
    }
```

```
    void showDialogFragment(String strFragmentNumber) {
        DialogFragment newFragment = MyAlertDialogFragment
            .newInstance(strFragmentNumber);
        newFragment.show(getFragmentManager(), strFragmentNumber);
    }

    public void doPositiveClick(String strFragmentNumber) {
        Toast.makeText(getApplicationContext(),
            "Clicked OK! (" + strFragmentNumber + ")",
            Toast.LENGTH_SHORT).show();
    }
}
```

Figure 2 shows a `DialogFragment` being displayed to the user.

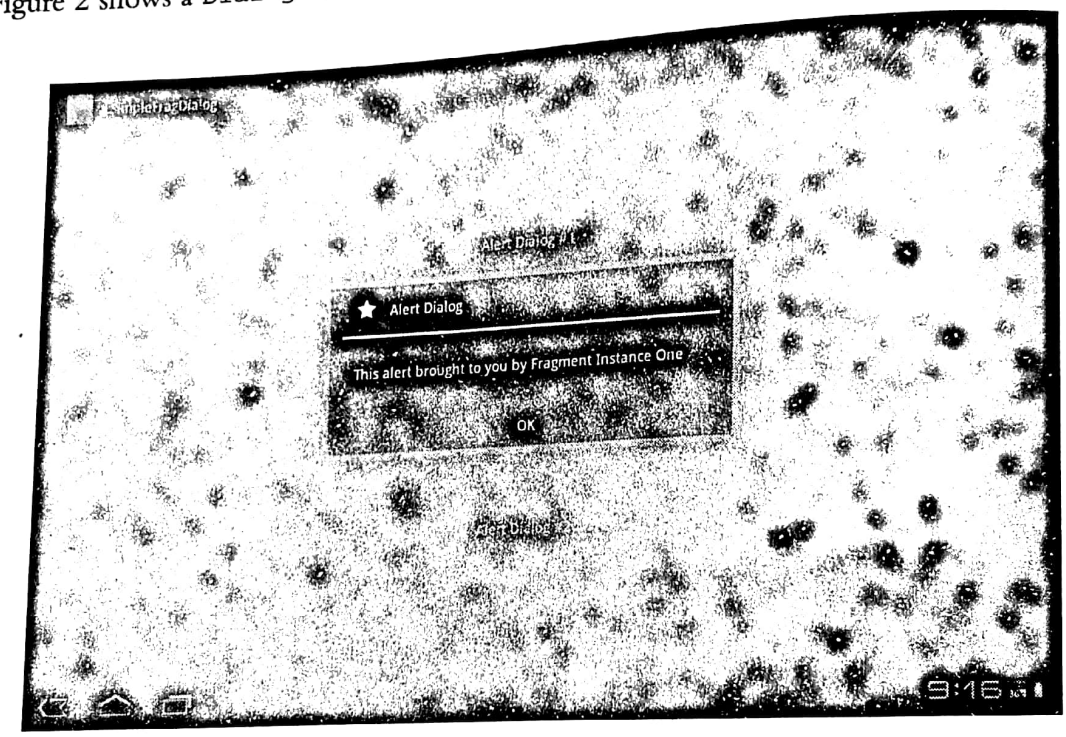

Figure 2 Using `DialogFragment` instances in an `Activity`.

We've shown you how to create `Dialog` controls that are managed by a `Fragment`, but when you start thinking of dialogs as fragments, you will see that this makes your dialogs much more powerful. For example, your `Dialog` controls can now be used by different activities because their lifecycle is managed inside the `Fragment`, not the `Activity`.

Also, `DialogFragment` instances can be traditional pop-ups (as shown in the example provided) or they can be embedded like any other `Fragment`. Why might you want to embed a `Dialog`? Consider the following example: You've created a picture gallery application and implemented a custom dialog that displays a larger-sized image when

you click a thumbnail. On small-screened devices, you might want this to be a pop-up `Dialog`, but on a tablet or TV, you might have the screen space to show the larger-sized graphic off to the right or below the thumbnails. This would be a good opportunity to take advantage of code reuse and simply embed your `Dialog`.

Summary

Dialogs are useful controls for keeping your Android application user interfaces clean and user friendly. Many types of `Dialog` controls are defined in the Android SDK, and you can create custom `Dialog` controls if none of the canned controls suit your purposes.

Developers should be aware that there are two different approaches to implementing `Dialog` controls within applications. With the legacy method, the `Activity` class manages its `Dialog` controls using a number of straightforward callbacks. This method is backward-compatible without any issues, but it does not work well with the new `Fragment`-oriented user interface design paradigms used in Honeycomb and beyond. The second method involves using a special type of `Fragment` called a `DialogFragment`. This method decouples your dialogs from the `Activity` and instead treats them like fragments. They can also be used by other `Fragment` instances as well.

References and More Information

Android SDK Reference regarding the application `Dialog` class:
http://d.android.com/reference/android/app/Dialog.html

Android SDK Reference regarding the application `AlertDialog` class:
http://d.android.com/reference/android/app/AlertDialog.html

Android SDK Reference regarding the application `DatePickerDialog` class:
http://d.android.com/reference/android/app/DatePickerDialog.html

Android SDK Reference regarding the application `TimePickerDialog` class:
http://d.android.com/reference/android/app/TimePickerDialog.html

Android SDK Reference regarding the application `ProgressDialog` class:
http://d.android.com/reference/android/app/ProgressDialog.html

Android SDK Reference regarding the application `CharacterPickerDialog` class:
http://d.android.com/reference/android/text/method/CharacterPickerDialog.html

Android SDK Reference regarding the application `DialogFragment` class:
http://d.android.com/reference/android/app/DialogFragment.html

Android Dev Guide: "Creating Dialogs":
http://d.android.com/guide/topics/ui/dialogs.html

Android DialogFragment Reference: "Selecting Between Dialog and Embedding":
http://d.android.com/reference/android/app/DialogFragment.html#DialogOrEmbed