```
    package="com.paad.contactpicker">
    <application android:icon="@drawable/icon">
      <activity android:name=".ContactPicker"
                android:label="@string/app_name">
        <intent-filter>
          <action android:name="android.intent.action.PICK"/>
          <category android:name="android.intent.category.DEFAULT"/>
          <data android:path="contacts" android:scheme="content"/>
        </intent-filter>
      </activity>
      <activity android:name=".ContentPickerTester"
                android:label="Contact Picker Test">
        <intent-filter>
          <action android:name="android.intent.action.MAIN"/>
          <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
      </activity>
    </application>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
</manifest>
```

When your Activity is running, press the button. The contact picker Activity should be shown as in Figure 5-1.

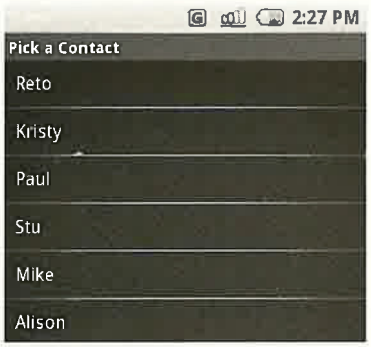


Figure 5-1

Once you select a contact, the parent Activity should return to the foreground with the selected contact name displayed, as shown in Figure 5-2.

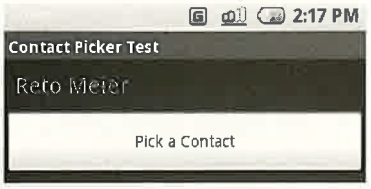


Figure 5-2

THE UNIVERSITY OF MICHIGAN LIBRARIES

REGIONS 1005
PART 2 OF 3

Using Intent Filters for Plug-ins and Extensibility

So far you've learned how to explicitly create implicit Intents, but that's only half the story. Android lets future packages provide new functionality for existing applications, using Intent Filters to populate menus dynamically at run time.

This provides a plug-in model for your Activities that lets them take advantage of functionality you haven't yet conceived of through new application components, without your having to modify or recompile your projects.

The `addIntentOptions` method available from the `Menu` class lets you specify an Intent that describes the data that is acted upon by this menu. Android resolves this Intent and returns every action specified in the Intent Filters that matches the specified data. A new Menu Item is created for each, with the text populated from the matching Intent Filters' labels.

The elegance of this concept is best explained by example. Say the data your application displays are a list of places. At the moment, the menu actions available might include "view" and "show directions to." Jump a few years ahead, and you've created an application that interfaces with your car, allowing your phone to handle driving. Thanks to the runtime menu generation, by including a new Intent Filter — with a `DRIVE_CAR` action — within the new Activity's node, Android will automagically add this action as a new Menu Item in your earlier application.

Runtime menu population provides the ability to retrofit functionality when you create new components capable of performing actions on a given type of data. Many of Android's native applications use this functionality, giving you the ability to provide additional actions to native Activities.

Supplying Anonymous Actions to Applications

To make actions available for other Activities, publish them using `intent-filter` tags within their manifest nodes.

The Intent Filter describes the `action` it performs and the `data` upon which it can be performed. The latter will be used during the Intent resolution process to determine when this action should be available. The `category` tag must be either or both `ALTERNATIVE` and `SELECTED_ALTERNATIVE`. The text used by Menu Items is specified by the `android:label` attribute.

The following XML shows an example of an Intent Filter used to advertise an Activity's ability to nuke moon bases from orbit.

```
<activity android:name=".NostromoController">
  <intent-filter android:label="Nuke From Orbit">
    <action android:name="com.pad.nostromo.NUKE_FROM_ORBIT"/>
    <data android:mimeType="vnd.moonbase.cursor.item/*"/>
    <category android:name="android.intent.category.ALTERNATIVE"/>
    <category
      android:name="android.intent.category.SELECTED_ALTERNATIVE"
    />
  </intent-filter>
</activity>
```

The Content Provider and other code needed for this example to run aren't provided; in the following sections, you'll see how to write the code that will make this action available dynamically from another Activity's menu.

Incorporating Anonymous Actions in Your Activity's Menu

To add menu options to your menus at run time, you use the `addIntentOptions` method on the menu object in question, passing in an Intent that specifies the data for which you want to provide actions. Generally, this will be handled within your Activity's `onCreateOptionsMenu` or `onCreateContextMenu handlers`.

The Intent you create will be used to resolve components with Intent Filters that supply actions for the data you specify. The Intent is being used to find actions, so don't assign it one; it should only specify the data on which to perform actions. You should also specify the category of the action, either `CATEGORY_ALTERNATIVE` or `CATEGORY_SELECTED_ALTERNATIVE`.

The skeleton code for creating an Intent for menu-action resolution is shown below:

```
Intent intent = new Intent();
intent.setData(MyProvider.CONTENT_URI);
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

Pass this Intent into `addIntentOptions` on the menu you wish to populate, as well as any option flags, the name of the calling class, the menu group to use, and menu ID values. You can also specify an array of Intents you'd like to use to create additional Menu Items.

The following code snippet gives an idea of how to dynamically populate an Activity menu that would include the "moonbase nuker" action from the previous section:

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

  // Create the intent used to resolve which actions
  // should appear in the menu.
  Intent intent = new Intent();
  intent.setData(MoonBaseProvider.CONTENT_URI);
  intent.addCategory(Intent.CATEGORY_SELECTED_ALTERNATIVE);

  // Normal menu options to let you set a group and ID
  // values for the menu items you're adding.
  int menuGroup = 0;
  int menuItemId = 0;
  int menuItemOrder = Menu.NONE;

  // Provide the name of the component that's calling
  // the action -- generally the current Activity.
  ComponentName caller = getComponentName();

  // Define intents that should be added first.
  Intent[] specificIntents = null;
  // The menu items created from the previous Intents
  // will populate this array.
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
        MenuItem[] outSpecificItems = null;

        // Set any optional flags.
        int flags = Menu.FLAG_APPEND_TO_GROUP;

        // Populate the menu
        menu.addIntentOptions(menuGroup,
                              menuItemId,
                              menuItemOrder,
                              caller,
                              specificIntents,
                              intent,
                              flags,
                              outSpecificItems);

        return true;
    }
```

Using Intents to Broadcast Events

As a system-level message-passing mechanism, Intents are capable of sending structured messages across process boundaries.

So far you've looked at using Intents to start new application components, but they can also be used to broadcast messages anonymously *between* components with the `sendBroadcast` method. You can implement Broadcast Receivers to listen for, and respond to, these broadcast Intents within your applications.

Broadcast Intents are used to notify listeners of system or application events, extending the event-driven programming model between applications.

Broadcasting Intents helps make your application more open; by broadcasting an event using an Intent, you let yourself and third-party developers react to events without having to modify your original application. Within your applications, you can listen for Broadcast Intents to replace or enhance native (or third-party) applications or react to system changes and application events.

For example, by listening for the incoming call broadcast, you can modify the ringtone or volume based on the caller.

Android uses Broadcast Intents extensively to broadcast system events like battery-charging levels, network connections, and incoming calls.

Broadcasting Events with Intents

Broadcasting Intents is actually quite simple. Within your application component, construct the Intent you want to broadcast, and use the `sendBroadcast` method to send it.

Set the action, data, and category of your Intent in a way that lets Broadcast Receivers accurately determine their interest. In this scenario, the Intent *action string* is used to identify the event being broadcast,

so it should be a unique string that identifies the event. By convention, action strings are constructed using the same form as Java packages, as shown in the following snippet:

```
public static final String NEW_LIFEFORM_DETECTED =
  "com.paad.action.NEW_LIFEFORM";
```

If you wish to include data within the Intent, you can specify a URI using the Intent's `data` property. You can also include *extras* to add additional primitive values. Considered in terms of an event-driven paradigm, the extras Bundle equates to optional parameters within an event handler.

The skeleton code below shows the basic creation of a Broadcast Intent using the action defined previously, with additional event information stored as extras.

```
Intent intent = new Intent(NEW_LIFEFORM_DETECTED);
intent.putExtra("lifeformName", lifeformType);
intent.putExtra("longitude", currentLongitude);
intent.putExtra("latitude", currentLatitude);

sendBroadcast(intent);
```

Listening for Broadcasts with Broadcast Receivers

Broadcast Receivers are used to listen for Broadcast Intents. To enable a Broadcast Receiver, it needs to be registered, either in code or within the application manifest. When registering a Broadcast Receiver, you must use an Intent Filter to specify which Intents it is listening for.

To create a new Broadcast Receiver, extend the `BroadcastReceiver` class and override the `onReceive` event handler as shown in the skeleton code below:

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class MyBroadcastReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    //TODO: React to the Intent received.
  }

}
```

The `onReceive` method will be executed when a Broadcast Intent is received that matches the Intent Filter used to register the receiver. The `onReceive` handler must complete within 5 seconds, or the *Application Unresponsive* dialog will be displayed.

Applications with registered Broadcast Receivers do not have to be running when the Intent is broadcast for the receivers to execute. They will be started automatically when a matching Intent is broadcast. This is excellent for resource management as it lets you create event-driven applications that can be closed or killed, safe in the knowledge that they will still respond to broadcast events.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Typically, Broadcast Receivers will update content, launch Services, update Activity UI, or notify the user using the Notification Manager. The 5-second execution limit ensures that major processing cannot, as it should not, be done within the Broadcast Receiver directly.

The following example shows how to implement a Broadcast Receiver. In the following sections, you will learn how to register it in code or in your application manifest.

```
public class LifeformDetectedBroadcastReceiver extends BroadcastReceiver {

  public static final String BURN =
    "com.paad.alien.action.BURN_IT_WITH_FIRE";

  @Override
  public void onReceive(Context context, Intent intent) {
    // Get the lifeform details from the intent.
    Uri data = intent.getData();
    String type = intent.getStringExtra("type");
    double lat = intent.getDoubleExtra("latitude", 0);
    double lng = intent.getDoubleExtra("longitude", 0);
    Location loc = new Location("gps");
    loc.setLatitude(lat);
    loc.setLongitude(lng);

    if (type.equals("alien")) {
      Intent startIntent = new Intent(BURN, data);
      startIntent.putExtra("latitude", lat);
      startIntent.putExtra("longitude", lng);

      context.startActivity(startIntent);
    }
  }
}
```

Registering Broadcast Receivers in Your Application Manifest

To include a Broadcast Receiver in the application manifest, add a `receiver` tag within the `application` node specifying the class name of the Broadcast Receiver to register. The receiver node needs to include an `intent-filter` tag that specifies the action string being listened for, as shown in the XML snippet below:

```
<receiver android:name=".LifeformDetectedBroadcastReceiver">
  <intent-filter>
    <action android:name="com.paad.action.NEW_LIFEFORM"/>
  </intent-filter>
</receiver>
```

Broadcast Receivers registered this way are always active.

Registering Broadcast Receivers in Code

You can control the registration of Broadcast Receivers in code. This is typically done when the receiver is being used to update UI elements in an Activity. In this case, it's good practice to unregister Broadcast Receivers when the Activity isn't visible (or active).

The following code snippet shows how to register a Broadcast Receiver using an `IntentFilter`:

```
// Create and register the broadcast receiver.
IntentFilter filter = new IntentFilter(NEW_LIFEFORM_DETECTED);
LifeformDetectedBroadcastReceiver r = new LifeformDetectedBroadcastReceiver();
registerReceiver(r, filter);
```

To unregister a Broadcast Receiver, use the `unregisterReceiver` method on your application context, passing in a Broadcast Receiver instance, as shown below:

```
unregisterReceiver(r);
```

Further examples can also be found in Chapter 8 when you learn to create your own background Services and use Intents to broadcast events back to your Activities.

Native Android Broadcast Actions

Android broadcasts Intents for many of the system Services. You can use these messages to add functionality to your own projects based on system events such as time-zone changes, data-connection status, incoming SMS messages, or phone calls.

The following list introduces some of the native actions exposed as constants in the Intents class; these actions are used primarily to track device status changes:

- `ACTION_BOOT_COMPLETED` Fired once when the device has completed its start-up sequence. Requires the `RECEIVE_BOOT_COMPLETED` permission.
- `ACTION_CAMERA_BUTTON` Fired when the Camera button is clicked.
- `ACTION_DATE_CHANGED` and `ACTION_TIME_CHANGED` These actions are broadcast if the date or time on the device is manually changed (as opposed to them changing through the natural progress of time).
- `ACTION_GTALK_SERVICE_CONNECTED` and `ACTION_GTALK_SERVICE_DISCONNECTED` These two actions are broadcast each time a GTalk connection is made or lost. More specific handling of GTalk messaging is discussed in more detail in Chapter 9.
- `ACTION_MEDIA_BUTTON` Fired when the Media button is clicked.
- `ACTION_MEDIA_EJECT` If the user chooses to eject the external storage media, this event is fired first. If your application is reading or writing to the external media storage, you should listen for this event in order to save and close any open file handles.
- `ACTION_MEDIA_MOUNTED` and `ACTION_MEDIA_UNMOUNTED` These two events are broadcast whenever new external storage media are successfully added or removed from the device.
- `ACTION_SCREEN_OFF` and `ACTION_SCREEN_ON` Broadcasts when the screen turns off or on.
- `ACTION_TIMEZONE_CHANGED` This action is broadcast whenever the phone's current time zone changes. The Intent includes a `time-zone` extra that returns the ID of the new `java.util.TimeZone`.

A comprehensive list of the broadcast actions used and transmitted natively by Android to notify applications of system state changes is available at `http://code.google.com/android/reference/android/content/Intent.html`.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Android also uses Broadcast Intents to monitor application-specific events like incoming SMS messages. The actions and Intents associated with these events are discussed in more detail in later chapters when you learn more about the associated Services.

Introducing Adapters

Adapters are bridging classes that bind data to user-interface Views. The adapter is responsible for creating the child views used to represent each item and providing access to the underlying data.

User-interface controls that support Adapter binding must extend the `AdapterView` abstract class. It's possible to create your own `AdapterView`-derived controls and create new Adapter classes to bind them.

Introducing Some Android-Supplied Adapters

In many cases, you won't have to create your own Adapter from scratch. Android supplies a set of Adapters that pump data into the native user-interface widgets.

Because Adapters are responsible both for supplying the data and selecting the Views that represent each item, Adaptors can radically modify the appearance and functionality of the controls they're bound to.

The following list highlights two of the most useful and versatile native adapters:

- `ArrayAdapter` The `ArrayAdapter` is a generic class that binds Adapter Views to an array of objects. By default, the `ArrayAdapter` binds the `toString` value of each object to a `TextView` control defined within a layout. Alternative constructors allow you to use more complex layouts, or you can extend the class to use alternatives to Text View (such as populating an `ImageView` or nested layout) by overriding the `getView` method.
- `SimpleCursorAdapter` The `SimpleCursorAdapter` binds Views to cursors returned from Content Provider queries. You specify an XML layout definition and then bind the value within each column in the result set, to a View in that layout.

The following sections will delve into these Adapter classes in more detail. The examples provided bind data to List Views, although the same logic will work just as well for other AdapterView classes such as `Spinners` and `Galleries`.

Using Adapters for Data Binding

To apply an Adapter to an AdapterView-derived class, you call the View's `setAdapter` method, passing in an Adapter instance, as shown in the snippet below:

```
ArrayList<String> myStringArray = new ArrayList<String>();
ArrayAdapter<String> myAdapterInstance;

int layoutID = android.R.layout.simple_list_item_1;
myAdapterInstance = new ArrayAdapter<String>(this, layoutID,
```

```
                                                    myStringArray);

myListView.setAdapter(myAdapterInstance);
```

This snippet shows the most simplistic case, where the array being bound is a string and the List View items are displayed using a single Text View control.

The first of the following examples demonstrates how to bind an array of complex objects to a List View using a custom layout. The second shows how to use a Simple Cursor Adapter to bind a query result to a custom layout within a List View.

Customizing the To-Do List `ArrayAdapter`

This example extends the To-Do List project, storing each item as a `ToDoItem` object that includes the date each item was created.

You will extend `ArrayAdapter` to bind a collection of `ToDoItem` objects to the `ListView` and customize the layout used to display each List View item.

1. Return to the To-Do List project. Create a new `ToDoItem` class that stores the task and its creation date. Override the `toString` method to return a summary of the item data.

```
package com.paad.todolist;

import java.text.SimpleDateFormat;
import java.util.Date;

public class ToDoItem {

  String task;
  Date created;

  public String getTask() {
    return task;
  }

  public Date getCreated() {
    return created;
  }

  public ToDoItem(String _task) {
    this(_task, new Date(java.lang.System.currentTimeMillis()));
  }

  public ToDoItem(String _task, Date _created) {
    task = _task;
    created = _created;
  }

  @Override
  public String toString() {
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yy");
    String dateString = sdf.format(created);
    return "(" + dateString +  ") " + task;
  }
}
```

2. Open the `ToDoList` Activity, and modify the `ArrayList` and `ArrayAdapter` variable types to store `ToDoItem` objects rather than Strings. You'll then need to modify the `onCreate` method to update the corresponding variable initialization. You'll also need to update the `onKeyListener` handler to support the `ToDoItem` objects.

```
private ArrayList<ToDoItem> todoItems;
private ListView myListView;
private EditText myEditText;
private ArrayAdapter<ToDoItem> aa;

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  // Inflate your view
  setContentView(R.layout.main);

  // Get references to UI widgets
  myListView = (ListView)findViewById(R.id.myListView);
  myEditText = (EditText)findViewById(R.id.myEditText);

  todoItems = new ArrayList<ToDoItem>();
  int resID = R.layout.todolist_item;
  aa = new ArrayAdapter<ToDoItem>(this, resID, todoItems);
  myListView.setAdapter(aa);

  myEditText.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent event) {
      if (event.getAction() == KeyEvent.ACTION_DOWN)
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
          ToDoItem newItem;
          newItem = new ToDoItem(myEditText.getText().toString());
          todoItems.add(0, newItem);
          myEditText.setText("");
          aa.notifyDataSetChanged();
          cancelAdd();
          return true;
        }
      return false;
    }
  });

  registerForContextMenu(myListView);
}
```

3. If you run the Activity, it will now display each to-do item, as shown in Figure 5-3.

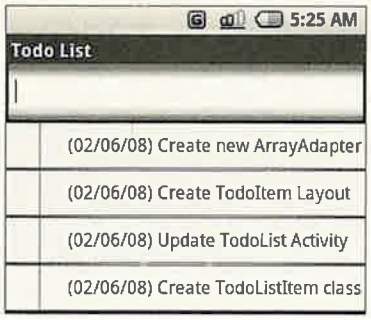


Figure 5-3

4. Now you can create a custom layout to display each to-do item. Start by modifying the custom layout you created in Chapter 4 to include a second `TextView`. It will be used to show the creation date of each to-do item.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:background="@color/notepad_paper">
  <TextView
    android:id="@+id/rowDate"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:scrollbars="vertical"
    android:fadingEdge="vertical"
    android:textColor="@color/notepad_text"
    android:layout_alignParentRight="true"
  />
  <TextView
    android:id="@+id/row"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:scrollbars="vertical"
    android:fadingEdge="vertical"
    android:textColor="@color/notepad_text"
    android:layout_alignParentLeft="@+id/rowDate"
  />
</RelativeLayout>
```

5. Create a new class (`ToDoItemAdapter`) that extends an `ArrayAdapter` with a `ToDoItem`-specific variation. Override `getView` to assign the task and date properties in the `ToDoItem` object to the Views in the layout you created in Step 4.

```
import java.text.SimpleDateFormat;
import android.content.Context;
import java.util.*;
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
import android.view.*;
import android.widget.*;

public class ToDoItemAdapter extends ArrayAdapter<ToDoItem> {

  int resource;

  public ToDoItemAdapter(Context _context,
                         int _resource,
                         List<ToDoItem> _items) {
    super(_context, _resource, _items);
    resource = _resource;
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent)
  {
    LinearLayout todoView;

    ToDoItem item = getItem(position);

    String taskString = item.getTask();
    Date createdDate = item.getCreated();
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yy");
    String dateString = sdf.format(createdDate);

    if (convertView == null) {
      todoView = new LinearLayout(getContext());
      String inflater = Context.LAYOUT_INFLATER_SERVICE;
      LayoutInflater vi;
      vi = (LayoutInflater)getContext().getSystemService(inflater);
      vi.inflate(resource, todoView, true);
    } else {
      todoView = (LinearLayout) convertView;
    }

    TextView dateView = (TextView)todoView.findViewById(R.id.rowDate);
    TextView taskView = (TextView)todoView.findViewById(R.id.row);

    dateView.setText(dateString);
    taskView.setText(taskString);

    return todoView;
  }
}
```

6. Finally, replace the `ArrayAdapter` declaration with a `ToDoItemAdapter`.

```
private ToDoItemAdapter aa;
```

Within `onCreate`, replace the `ArrayAdapter<String>` instantiation with the new `ToDoItemAdapter`.

```
aa = new ToDoItemAdapter(this, resID, todoItems);
```

7. If you run your Activity, it should appear as shown in the screenshot in Figure 5-4.

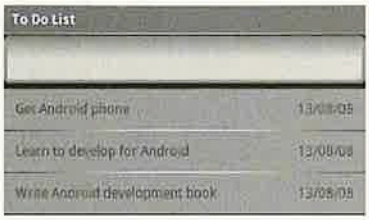


Figure 5-4

Using the `SimpleCursorAdapter`

The `SimpleCursorAdapter` lets you bind columns from a `Cursor` to a List View using a custom layout definition.

The `SimpleCursorAdapter` is constructed by passing in the current context, a layout resource, a Cursor, and two arrays: one that contains the names of the columns to be used and a second (equally sized) array that has resource IDs for the Views to use to display the corresponding column's data value.

The following skeleton code shows how to construct a `SimpleCursorAdapter` to display contact information:

```
String uriString = "content://contacts/people/";
Cursor myCursor = managedQuery(Uri.parse(uriString), null, null, null, null);

String[] fromColumns = new String[] {People.NUMBER, People.NAME};

int[] toLayoutIDs = new int[] { R.id.nameTextView, R.id.numberTextView};

SimpleCursorAdapter myAdapter;
myAdapter = new SimpleCursorAdapter(this,
                                    R.layout.simplecursorlayout,
                                    myCursor,
                                    fromColumns,
                                    toLayoutIDs);

myListView.setAdapter(myAdapter);
```

The Simple Cursor Adapter was used previously in this chapter when creating the Contact Picker example. You'll learn more about Content Providers and Cursors in Chapter 6, where you'll also find more `SimpleCursorAdapter` examples.

Using Internet Resources

With Internet connectivity and WebKit browser, you might well ask if there's any reason to create native Internet-based applications when you could make a web-based version instead.

THE UNIVERSITY OF MICHIGAN LIBRARIES

There are several benefits to creating thick- and thin-client native applications rather than relying on entirely web-based solutions:

- **Bandwidth** Static resources like images, layouts, and sounds can be expensive data consumers on devices with limited and often expensive bandwidth restraints. By creating a native application, you can limit the bandwidth requirements to only data updates.
- **Caching** Mobile Internet access has not yet reached the point of ubiquity. With a browser-based solution, a patchy Internet connection can result in intermittent application availability. A native application can cache data to provide as much functionality as possible without a live connection.
- **Native Features** Android devices are more than a simple platform for running a browser; they include location-based services, camera hardware, and accelerometers. By creating a native application, you can combine the data available online with the hardware features available on the device to provide a richer user experience.

Modern mobile devices offer various alternatives for accessing the Internet. Looked at broadly, Android provides three connection techniques for Internet connectivity. Each is offered transparently to the application layer.

- **GPRS, EDGE, and 3G** Mobile Internet access is available through carriers that offer mobile data plans.
- **Wi-Fi** Wi-Fi receivers and mobile hotspots are becoming increasingly more common.

Connecting to an Internet Resource

While the details of working with specific web services aren't covered within this book, it's useful to know the general principles of connecting to the Internet and getting an input stream from a remote data source.

Before you can access Internet resources, you need to add an `INTERNET` `uses-permission` node to your application manifest, as shown in the following XML snippet:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

The following skeleton code shows the basic pattern for opening an Internet data stream:

```
String myFeed = getString(R.string.my_feed);
try {
  URL url = new URL(myFeed);

  URLConnection connection = url.openConnection();
  HttpURLConnection httpConnection = (HttpURLConnection)connection;

  int responseCode = httpConnection.getResponseCode();
  if (responseCode == HttpURLConnection.HTTP_OK) {
    InputStream in = httpConnection.getInputStream();
```

```
      [ ... Process the input stream as required ... ]
    }
  }
  catch (MalformedURLException e) { }
  catch (IOException e) { }
```

Android includes several classes to help you handle network communications. They are available in the `java.net.*` and `android.net.*` packages.

Later in this chapter, there is a fully worked example that shows how to obtain and process an Internet feed to get a list of earthquakes felt in the last 24 hours.

Chapter 9 includes further details on Internet-based communications using the GTalk Service. Chapter 10 features more information on managing specific Internet connections, including monitoring connection status and configuring Wi-Fi access point connections.

Leveraging Internet Resources

Android offers several ways to leverage Internet resources.

At one extreme, you can use the `WebView` widget to include a WebKit-based browser control within an Activity. At the other extreme, you can use client-side APIs such as Google's GData APIs to interact directly with server processes. Somewhere in between, you can process remote XML feeds to extract and process data using a Java-based XML parser such as `SAX` or `javax`.

Detailed instructions for parsing XML and interacting with specific web services are outside the scope of this book. That said, the Earthquake example, included later in this chapter, gives a fully worked example of parsing an XML feed using the `javax` classes.

If you're using Internet resources in your application, remember that your users' data connections depend on the communications technology available to them. EDGE and GSM connections are notoriously low bandwidth, while a Wi-Fi connection may be unreliable in a mobile setting.

Optimize the user experience by limiting the quantity of data being transmitted, and ensure that your application is robust enough to handle network outages and bandwidth limitations.

Introducing Dialogs

Dialog boxes are a common UI metaphor in desktop and web applications. They're used to help users answer questions, make selections, confirm actions, and read warning or error messages. An *Android Dialog* is a floating window that partially obscures the Activity that launched it.

As you can see in Figure 5-5, Dialog boxes are not full screen and can be partially transparent. They generally obscure the Activities behind them using a blur or dim filter.

THE UNIVERSITY OF MICHIGAN LIBRARIES



Figure 5-5

There are three ways to implement a Dialog box in Android:

- **Using a Dialog-Class Descendent** As well as the general-purpose `AlertDialog` class, Android includes several specialist classes that extend `Dialog`. Each is designed to provide specific Dialog-box functionality. `Dialog`-class-based screens are constructed entirely within their calling Activity, so they don't need to be registered in the manifest, and their life cycle is controlled entirely by the calling Activity.
- **Dialog-Themed Activities** You can apply the Dialog theme to a regular Activity to give it the appearance of a Dialog box.
- **Toasts** Toasts are special non-modal transient message boxes, often used by Broadcast Receivers and background services to notify users of events. You learn more about Toasts in Chapter 8.

Introducing the `Dialog` Class

The `Dialog` class implements a simple floating window that is constructed entirely within an Activity.

To use the base `Dialog` class, you create a new instance and set the title and layout as shown below:

```
Dialog d = new Dialog(MyActivity.this);

// Have the new window tint and blur the window it
// obscures.
Window window = d.getWindow();
window.setFlags(WindowManager.LayoutParams.FLAG_BLUR_BEHIND,
                WindowManager.LayoutParams.FLAG_BLUR_BEHIND);

d.setTitle("Dialog Title");
d.setContentView(R.layout.dialog_view);

TextView text = (TextView)d.findViewById(R.id.dialogTextView);
text.setText("This is the text in my dialog");
```

Once it's configured to your liking, use the `show` method to display it.

```
d.show();
```

The `AlertDialog` Class

The `AlertDialog` class is one of the most versatile Dialog implementations. It offers various options that let you construct screens for some of the most common Dialog-box use cases, including:

- Presenting a message to the user offering one to three options in the form of alternative buttons. This functionality is probably familiar to you if you've done any desktop programming, where the buttons presented are usually a selection of OK, Cancel, Yes, or No.
- Offering a list of options in the form of check buttons or radio buttons.
- Providing a text entry box for user input.

To construct the Alert Dialog user interface, create a new `AlertDialog.Builder` object, as shown below:

```
AlertDialog.Builder ad = new AlertDialog.Builder(context);
```

You can then assign values for the title and message to display, and optionally assign values to be used for any buttons, selection items, and text input boxes you wish to display. That includes setting event listeners to handle user interaction.

The following code gives an example of a new Alert Dialog used to display a message and offer two button options to continue. Clicking on either button will automatically close the Dialog after executing the attached Click Listener.

```
Context context = MyActivity.this;
String title = "It is Pitch Black";
String message = "You are likely to be eaten by a grue.";
String button1String = "Go Back";
String button2String = "Move Forward";

AlertDialog.Builder ad = new AlertDialog.Builder(context);
ad.setTitle(title);
ad.setMessage(message);
ad.setPositiveButton(button1String,
                     new OnClickListener() {
                       public void onClick(DialogInterface dialog,
                                           int arg1) {
                         eatenByGrue();
                       }
                     });
ad.setNegativeButton(button2String,
                     new OnClickListener(){
                       public void onClick(DialogInterface dialog,
                                           int arg1) {
                         // do nothing
                       }
                     });
ad.setCancelable(true);
ad.setOnCancelListener(new OnCancelListener() {
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
    public void onCancel(DialogInterface dialog) {
      eatenByGrue();
    }
  });
```

To display an Alert Dialog that you've created call `show`.

```
ad.show();
```

Alternatively, you can override the `onCreateDialog` and `onPrepareDialog` methods within your Activity to create single-instance Dialogs that persist their state. This technique is examined later in this chapter.

Specialist Input Dialogs

One of the major uses of Dialog boxes is to provide an interface for user input. Android includes several specialist Dialog boxes that encapsulate controls designed to facilitate common user input requests. They include the following:

- **DatePickerDialog** Lets users select a date from a `DatePicker` View. The constructor includes a callback listener to alert your calling Activity when the date has been set.
- **TimePickerDialog** Similar to the DatePickerDialog, this Dialog lets users select a time from a `TimePicker` View.
- **ProgressDialog** A Dialog that displays a progress bar beneath a message textbox. Perfect for keeping the user informed of the ongoing progress of a time-consuming operation.

Using and Managing Dialogs

Rather than creating new instances of a Dialog each time it's required, Android provides the `OnCreateDialog` and `onPrepareDialog` event handlers within the Activity class to persist and manage Dialog-box instances.

By overriding the `onCreateDialog` class, you can specify Dialogs that will be created on demand when `showDialog` is used to display a specific Dialog. As shown in this code snippet, the overridden method includes a switch statement that lets you determine which Dialog is required:

```
static final private int TIME_DIALOG = 1;

@Override
public Dialog onCreateDialog(int id) {
  switch(id) {
    case (TIME_DIALOG) :
      AlertDialog.Builder timeDialog = new AlertDialog.Builder(this);
      timeDialog.setTitle("The Current Time Is...");
      timeDialog.setMessage("Now");
      return timeDialog.create();
  }
  return null;
}
```

After the initial creation, each time a `showDialog` is called, it will trigger the `onPrepareDialog` handler. By overriding this method, you can modify a Dialog immediately before it is displayed. This lets you contextualize any of the display values, as shown in the following snippet, which assigns the current time to the Dialog created above:

```
@Override
public void onPrepareDialog(int id, Dialog dialog) {
  switch(id) {
    case (TIME_DIALOG) :
      SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
      Date currentTime;
      currentTime = new Date(java.lang.System.currentTimeMillis());
      String dateString = sdf.format(currentTime);
      AlertDialog timeDialog = (AlertDialog)dialog;
      timeDialog.setMessage(dateString);

      break;
  }
}
```

Once you've overridden these methods, you can display the Dialogs by calling `showDialog`, as shown below. Pass in the identifier for the Dialog you wish to display, and Android will create (if necessary) and prepare the Dialog before displaying it:

```
showDialog(TIME_DIALOG);
```

As well as improving resource use, this technique lets your Activity handle the persistence of state information within Dialogs. Any selection or data input (such as item selection and text entry) will be persisted between displays of each Dialog instance.

Using Activities as Dialogs

Dialogs offer a simple and lightweight technique for displaying screens, but there will still be times when you need more control over the content and life cycle of your Dialog box.

The solution is to implement it as a full Activity. By creating an Activity, you lose the lightweight nature of the `Dialog` class, but you gain the ability to implement any screen you want and full access to the Activity life-cycle event handlers.

So, when is an Activity a Dialog? The easiest way to make an Activity look like a Dialog is to apply the `android:style/Theme.Dialog` theme when you add it to your manifest, as shown in the following XML snippet:

```
<activity android:name="MyDialogActivity"
          android:theme="@android:style/Theme.Dialog">
</activity>
```

This will cause your Activity to behave like a Dialog, floating on top of, and partially obscuring, the Activity beneath it.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Creating an Earthquake Viewer

In the following example, you'll create a tool that uses a USGS earthquake feed to display a list of recent earthquakes.

You will return to this Earthquake application several times, first in Chapter 6 to save and share the earthquake data with a Content Provider, and again in Chapters 7 and 8 to add mapping support and to move the earthquake updates into a background Service.

In this example, you will create a list-based Activity that connects to an earthquake feed and displays the location, magnitude, and time of the earthquakes it contains. You'll use an Alert Dialog to provide a detail window that includes a linkified Text View with a link to the USGS web site.

1. Start by creating an Earthquake project featuring an `Earthquake` Activity. Modify the main.xml layout resource to include a List View control — be sure to name it so you can reference it from the Activity code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <ListView
    android:id="@+id/earthquakeListView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

2. Create a new public `Quake` class. This class will be used to store the details (date, details, location, magnitude, and link) of each earthquake. Override the `toString` method to provide the string that will be used for each quake in the List View.

```
package com.paad.earthquake;

import java.util.Date;
import java.text.SimpleDateFormat;
import android.location.Location;

public class Quake {
  private Date date;
  private String details;
  private Location location;
  private double magnitude;
  private String link;

  public Date getDate() { return date; }
  public String getDetails() { return details; }
  public Location getLocation() { return location; }
  public double getMagnitude() { return magnitude; }
  public String getLink() { return link; }

  public Quake(Date _d, String _det, Location _loc, double _mag,
               String _link) {
    date = _d;
```

```
    details = _det;
    location = _loc;
    magnitude = _mag;
    link = _link;
  }

  @Override
  public String toString() {
    SimpleDateFormat sdf = new SimpleDateFormat("HH.mm");
    String dateString = sdf.format(date);
    return dateString + ": " + magnitude + " " + details;
  }

}
```

3. In the `Earthquake` Activity, override the `onCreate` method to store an ArrayList of `Quake` objects, and bind that to the ListView using an `ArrayAdapter`.

```
package com.paad.earthquake;

import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.GregorianCalendar;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
import android.app.Activity;
import android.app.Dialog;
import android.location.Location;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.view.WindowManager;
import android.view.MenuItem;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.AdapterView.OnItemClickListener;

public class Earthquake extends Activity {

  ListView earthquakeListView;
```

```
    ArrayAdapter<Quake> aa;

    ArrayList<Quake> earthquakes = new ArrayList<Quake>();

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

      earthquakeListView =
        (ListView)this.findViewById(R.id.earthquakeListView);

      int layoutID = android.R.layout.simple_list_item_1;
      aa = new ArrayAdapter<Quake>(this, layoutID , earthquakes);
      earthquakeListView.setAdapter(aa);
    }
  }
```

4. Next, you should start processing the earthquake feed. For this example, the feed used is the 1-day USGS feed for earthquakes with a magnitude greater than 2.5.

Add the location of your feed as an external string resource. This lets you potentially specify a different feed based on a user's location.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Earthquake</string>
  <string name="quake_feed">
    http://earthquake.usgs.gov/eqcenter/catalogs/1day-M2.5.xml
  </string>
</resources>
```

5. Before your application can access the Internet, it needs to be granted permission for Internet access. Add the `uses-permission` to the manifest.

```
<uses-permission xmlns:android="http://schemas.android.com/apk/res/android"
  android:name="android.permission.INTERNET">
</uses-permission>
```

6. Returning to the Earthquake Activity, create a new `refreshEarthquakes` method that connects to, and parses, the earthquake feed. Extract each earthquake, and parse the details to obtain the date, magnitude, link, and location. As you finish parsing each earthquake, pass it in to a new `addNewQuake` method.

The XML parsing is presented here without further comment.

```
private void refreshEarthquakes() {
  // Get the XML
  URL url;
  try {
    String quakeFeed = getString(R.string.quake_feed);
    url = new URL(quakeFeed);

    URLConnection connection;
```

```
connection = url.openConnection();

HttpURLConnection httpConnection = (HttpURLConnection)connection;
int responseCode = httpConnection.getResponseCode();

if (responseCode == HttpURLConnection.HTTP_OK) {
  InputStream in = httpConnection.getInputStream();

  DocumentBuilderFactory dbf;
  dbf = DocumentBuilderFactory.newInstance();
  DocumentBuilder db = dbf.newDocumentBuilder();

  // Parse the earthquake feed.
  Document dom = db.parse(in);
  Element docEle = dom.getDocumentElement();

  // Clear the old earthquakes
  earthquakes.clear();

  // Get a list of each earthquake entry.
  NodeList nl = docEle.getElementsByTagName("entry");
  if (nl != null && nl.getLength() > 0) {
    for (int i = 0 ; i < nl.getLength(); i++) {
      Element entry = (Element)nl.item(i);
      Element title =
       (Element)entry.getElementsByTagName("title").item(0);
      Element g =
       (Element)entry.getElementsByTagName("georss:point").item(0);
      Element when =
       (Element)entry.getElementsByTagName("updated").item(0);
      Element link =
       (Element)entry.getElementsByTagName("link").item(0);

      String details = title.getFirstChild().getNodeValue();
      String hostname = "http://earthquake.usgs.gov";
      String linkString = hostname + link.getAttribute("href");

      String point = g.getFirstChild().getNodeValue();
      String dt = when.getFirstChild().getNodeValue();
      SimpleDateFormat sdf;
      sdf = new SimpleDateFormat("yyyy-MM-dd'T'hh:mm:ss'Z'");
      Date qdate = new GregorianCalendar(0,0,0).getTime();
      try {
        qdate = sdf.parse(dt);
      } catch (ParseException e) {
        e.printStackTrace();
      }

      String[] location = point.split(" ");
      Location l = new Location("dummyGPS");
      l.setLatitude(Double.parseDouble(location[0]));
      l.setLongitude(Double.parseDouble(location[1]));

      String magnitudeString = details.split(" ")[1];
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
            int end =  magnitudeString.length()-1;
            double magnitude;
            magnitude = Double.parseDouble(magnitudeString.substring(0,
                                                                    end));

            details = details.split(",")[1].trim();

            Quake quake = new Quake(qdate, details, l,
                                    magnitude, linkString);

            // Process a newly found earthquake
            addNewQuake(quake);
          }
        }
      }
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ParserConfigurationException e) {
      e.printStackTrace();
    } catch (SAXException e) {
      e.printStackTrace();
    }
    finally {
    }
  }

  private void addNewQuake(Quake _quake) {
    // TODO: Add the earthquakes to the array list.
  }
```

7. Update the **addNewQuake** method so that it takes each newly processed quake and adds it to the Earthquake **ArrayList**. It should also notify the Array Adapter that the underlying data have changed.

```
private void addNewQuake(Quake _quake) {
  // Add the new quake to our list of earthquakes.
  earthquakes.add(_quake);

  // Notify the array adapter of a change.
  aa.notifyDataSetChanged();
}
```

8. Modify your `onCreate` method to call `refreshEarthquakes` on start-up.

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

  earthquakeListView =
```

```
        (ListView)this.findViewById(R.id.earthquakeListView);

    int layoutID = android.R.layout.simple_list_item_1;
    aa = new ArrayAdapter<Quake>(this, layoutID , earthquakes);
    earthquakeListView.setAdapter(aa);

    refreshEarthquakes();
}
```

The Internet lookup is currently happening on the main UI thread. This is bad form as the application will become unresponsive if the lookup takes longer than a few seconds. In Chapter 8, you'll learn how to move expensive or time-consuming operations like this onto the background thread.

9. If you run your project, you should see a List View that features the earthquakes from the last 24 hours with a magnitude greater than 2.5, as shown in the screenshot in Figure 5-6.



Figure 5-6

10. There are only two more steps to make this a more useful application. First, create a new menu item to let users refresh the earthquake feed on demand.

10.1. Start by adding a new external string for the menu option.

```
<string name="menu_update">Refresh Earthquakes</string>
```

10.2 Then override the Activity's `onCreateOptionsMenu` and `onOptionsItemSelected` methods to display and handle the *refresh earthquakes* menu item.

```
static final private int MENU_UPDATE = Menu.FIRST;

@Override
```

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

  menu.add(0, MENU_UPDATE, Menu.NONE, R.string.menu_update);

  return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  super.onOptionsItemSelected(item);

  switch (item.getItemId()) {
    case (MENU_UPDATE): {
      refreshEarthquakes();
      return true;
    }
  }
  return false;
}
```

11. Now add some interaction. Let users find more details by opening a Dialog box when they select an earthquake from the list.

11.1. Creating a new quake_details.xml layout resource for the Dialog box you'll display on an item click.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:padding="10sp">
  <TextView
    android:id="@+id/quakeDetailsTextView"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textSize="14sp"
  />
</LinearLayout>
```

11.2. Then modify your `onCreate` method to add an `ItemClickListener` to the List View that displays a Dialog box whenever an earthquake item is selected.

```
static final private int QUAKE_DIALOG = 1;
Quake selectedQuake;

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

  earthquakeListView =
```

```
    (ListView)this.findViewById(R.id.earthquakeListView);

  earthquakeListView.setOnItemClickListener(new OnItemClickListener() {

    public void onItemClick(AdapterView _av, View _v, int _index,
                            long arg3) {
      selectedQuake = earthquakes.get(_index);
      showDialog(QUAKE_DIALOG);
    }
  });

  int layoutID = android.R.layout.simple_list_item_1;
  aa = new ArrayAdapter<Quake>(this, layoutID , earthquakes);
  earthquakeListView.setAdapter(aa);

  refreshEarthquakes();
}
```

11.3. Now override the `onCreateDialog` and `onPrepareDialog` methods to create and populate the Earthquake Details dialog.

```
@Override
public Dialog onCreateDialog(int id) {
  switch(id) {
    case (QUAKE_DIALOG) :
      LayoutInflater li = LayoutInflater.from(this);
      View quakeDetailsView = li.inflate(R.layout.quake_details, null);

      AlertDialog.Builder quakeDialog = new AlertDialog.Builder(this);
      quakeDialog.setTitle("Quake Time");
      quakeDialog.setView(quakeDetailsView);
      return quakeDialog.create();
  }
  return null;
}

@Override
public void onPrepareDialog(int id, Dialog dialog) {
  switch(id) {
    case (QUAKE_DIALOG) :
      SimpleDateFormat sdf;
      sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
      String dateString = sdf.format(selectedQuake.getDate());
      String quakeText = "Mangitude " + selectedQuake.getMagnitude() +
                         "\n" + selectedQuake.getDetails()  + "\n" +
                         selectedQuake.getLink();

      AlertDialog quakeDialog = (AlertDialog)dialog;
      quakeDialog.setTitle(dateString);
      TextView tv =
        (TextView)quakeDialog.findViewById(R.id.quakeDetailsTextView);
      tv.setText(quakeText);

      break;
  }
}
```


THE UNIVERSITY OF CHICAGO LIBRARIES

11.4. The final step is to linkify the Dialog to make the link to the USGS a hyperlink. Adjust the Dialog box's XML layout resource definition to include an `autolink` attribute.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:padding="10sp">
  <TextView
    android:id="@+id/quakeDetailsTextView"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textSize="14sp"
    android:autoLink="all"
  />
</LinearLayout>
```

Launch your activity again. When you click on a particular earthquake, a Dialog box will appear, partially obscuring the list, as shown in Figure 5-7.



Figure 5-7

Summary

The focus of this chapter has been on binding your application components.

Intents provide a versatile messaging system that lets you pass intentions between your application and others, to perform actions and signal events. You learned how to use implicit and explicit Intents to start new Activities, and how to populate an Activity menu dynamically through runtime resolution of Activity Intent Filters.

You were introduced to Broadcast Intents and saw how they can be used to send messages throughout the device, particularly to support an event-driven model based on system- and application-specific events.

You learned how to use sub-Activities to pass data between Activities and how to use Dialogs to display information and facilitate user input.

Adapters were introduced and used to bind underlying data to visual components. In particular, you saw how to use an Array Adapter and Simple Cursor Adapter to bind a List View to Array Lists and Cursors.

Finally, you learned the basics behind connecting to the Internet and using remote feeds as data sources for your native client applications.

You also learned:

- To use Linkify to add implicit View Intents to TextViews at run time.
- Which native Android actions are available for you to extend, replace, or embrace.
- How to use Intent Filters to let your own Activities become handlers for completing action requests from your own or other applications.
- How to listen for Broadcast Intents using Broadcast Receivers.
- How to use an Activity as a Dialog box.

In the next chapter, you will learn how to persist information within your applications. Android provides several mechanisms for saving application data, including files, simple preferences, and fully featured relational databases (using the SQLite database library).

THE UNIVERSITY OF MICHIGAN LIBRARIES

6

Data Storage, Retrieval, and Sharing

In this chapter, you'll be introduced to three of the most versatile data persistence techniques in Android — preferences, local files, and SQLite databases — before looking at Content Providers.

Saving and loading data is an essential requirement for most applications. At a minimum, Activities should save their User Interface (UI) state each time they move out of the foreground. This ensures that the same UI state is presented when it's next seen, even if the process has been killed and restarted before that happens.

It's also likely that you'll need to save preferences, to let users customize the application, and persist data entered or recorded. Just as important is the ability to load data from files, databases, and Content Providers — your own, and those shared by native and third-party applications.

Android's nondeterministic Activity and Application lifetimes make persisting UI state and application data between sessions particularly important. Android offers several alternatives for saving application data, each optimized to fulfill a particular need.

Preferences are a simple, lightweight key/value pair mechanism for saving primitive application data, most commonly a user's application preferences. Android also provides access to the local filesystem, both through specialized methods and the normal `Java.IO` classes.

For a more robust persistence layer, Android provides the SQLite database library. The SQLite database offers a powerful native SQL database over which you have total control.

THE UNIVERSITY OF MICHIGAN LIBRARY

Content Providers offer a generic interface to any data source. They effectively decouple the underlying data storage technique from the application layer.

By default, access to all files, databases, and preferences is restricted to the application that created them. Content Providers offer a managed way for your applications to share private data with other applications. As a result, your applications can use the Content Providers offered by others, including native providers.

Android Techniques for Saving Data

The data persistence techniques in Android provide options for balancing speed, efficiency, and robustness:

- **Shared Preferences** When storing the UI state, user preferences, or application settings, you want a lightweight mechanism to store a known set of values. Shared Preferences let you save groups of key/value pairs of primitive data as named preferences.
- **Files** It's not pretty, but sometimes writing to, and reading from, files directly is the only way to go. Android lets you create and load files on the device's internal or external media.
- **SQLite Databases** When managed, structured data is the best approach, Android offers the SQLite relational database library. Every application can create its own databases over which it has total control.
- **Content Providers** Rather than a storage mechanism in their own right, Content Providers let you expose a well-defined interface for using and sharing private data. You can control access to Content Providers using the standard permission system.

Saving Simple Application Data

There are two lightweight techniques for saving simple application data for Android applications — Shared Preferences and a pair of event handlers used for saving Activity instance details. Both mechanisms use a name/value pair (NVP) mechanism to store simple primitive values.

Using `SharedPreferences`, you can create named maps of key/value pairs within your application that can be shared between application components running in the same Context.

Shared Preferences support the primitive types Boolean, string, float, long, and integer, making them an ideal way to quickly store default values, class instance variables, the current UI state, and user preferences. They are most commonly used to persist data across user sessions and to share settings between application components.

Alternatively, Activities offer the `onSaveInstanceState` handler. It's designed specifically to persist the UI state when the Activity becomes eligible for termination by a resource-hungry run time.

The handler works like the Shared Preference mechanism. It offers a `Bundle` parameter that represents a key/value map of primitive types that can be used to save the Activity's instance values. This Bundle

is then made available as a parameter passed in to the `onCreate` and `onRestoreInstanceState` method handlers.

This UI state Bundle is used to record the values needed for an Activity to provide an identical UI following unexpected restarts.

Creating and Saving Preferences

To create or modify a Shared Preference, call `getSharedPreferences` on the application Context, passing in the name of the Shared Preferences to change. Shared Preferences are shared across an application's components but aren't available to other applications.

To modify a Shared Preference, use the `SharedPreferences.Editor` class. Get the Editor object by calling `edit` on the `SharedPreferences` object you want to change. To save edits, call `commit` on the Editor, as shown in the code snippet below.

```
public static final String MYPREFS = "mySharedPreferences";

protected void savePreferences(){
  // Create or retrieve the shared preference object.
  int mode = Activity.MODE_PRIVATE;
  SharedPreferences mySharedPreferences = getSharedPreferences(MYPREFS,
                                                               mode);
  // Retrieve an editor to modify the shared preferences.
  SharedPreferences.Editor editor = mySharedPreferences.edit();

  // Store new primitive types in the shared preferences object.
  editor.putBoolean("isTrue", true);
  editor.putFloat("lastFloat", 1f);
  editor.putInt("wholeNumber", 2);
  editor.putLong("aNumber", 31);
  editor.putString("textEntryValue", "Not Empty");

  // Commit the changes.
  editor.commit();
}
```

Retrieving Shared Preferences

Accessing saved Shared Preferences is also done with the `getSharedPreferences` method. Pass in the name of the Shared Preference you want to access, and use the type-safe `get<type>` methods to extract saved values.

Each getter takes a key and a default value (used when no value is available for that key), as shown in the skeleton code below:

```
public void loadPreferences() {
  // Get the stored preferences
  int mode = Activity.MODE_PRIVATE;
```

```
    SharedPreferences mySharedPreferences = getSharedPreferences(MYPREFS,
                                                                 mode);

    // Retrieve the saved values.
    boolean isTrue = mySharedPreferences.getBoolean("isTrue", false);
    float lastFloat = mySharedPreferences.getFloat("lastFloat", 0f);
    int wholeNumber = mySharedPreferences.getInt("wholeNumber", 1);
    long aNumber = mySharedPreferences.getLong("aNumber", 0);
    String stringPreference;
    stringPreference = mySharedPreferences.getString("textEntryValue",
                                                     "");
}
```

Saving the Activity State

If you want to save Activity information that doesn't need to be shared with other components (e.g., class instance variables), you can call `Activity.getPreferences()` without specifying a preferences name. Access to the Shared Preferences map returned is restricted to the calling Activity; each Activity supports a single unnamed SharedPreferences object.

The following skeleton code shows how to use the Activity's private Shared Preferences:

```
protected void saveActivityPreferences(){
  // Create or retrieve the activity preferences object.
  SharedPreferences activityPreferences =
    getPreferences(Activity.MODE_PRIVATE);

  // Retrieve an editor to modify the shared preferences.
  SharedPreferences.Editor editor = activityPreferences.edit();

  // Retrieve the View
  TextView myTextView = (TextView)findViewById(R.id.myTextView);

  // Store new primitive types in the shared preferences object.
  editor.putString("currentTextValue",
                   myTextView.getText().toString());

  // Commit changes.
  editor.commit();
}
```

Saving and Restoring Instance State

To save Activity instance variables, Android offers a specialized alternative to Shared Preferences.

By overriding an Activity's `onSaveInstanceState` event handler, you can use its `Bundle` parameter to save instance values. Store values using the same `get` and `put` methods as shown for Shared Preferences, before passing the modified `Bundle` into the superclass's handler, as shown in the following code snippet:

```
private static final String TEXTVIEW_STATE_KEY = "TEXTVIEW_STATE_KEY";
```

```
@Override
public void onSaveInstanceState(Bundle outState) {
  // Retrieve the View
  TextView myTextView = (TextView)findViewById(R.id.myTextView);

  // Save its state
  outState.putString(TEXTVIEW_STATE_KEY,
                     myTextView.getText().toString());
  super.onSaveInstanceState(outState);
}
```

This handler will be triggered whenever an Activity completes its Active life cycle, but only when it's not being explicitly finished. As a result, it's used to ensure a consistent Activity state between active life cycles of a single user session.

The saved `Bundle` is passed in to the `onRestoreInstanceState` and `onCreate` methods if the application is forced to restart during a session. The following snippet shows how to extract values from the `Bundle` and use them to update the Activity instance state:

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

  TextView myTextView = (TextView)findViewById(R.id.myTextView);

  String text = "";
  if (icicle != null && icicle.containsKey(TEXTVIEW_STATE_KEY))
    text = icicle.getString(TEXTVIEW_STATE_KEY);

  myTextView.setText(text);
}
```

It's important to remember that `onSaveInstanceState` *is called only when an Activity becomes inactive, but not when it is being closed by a call to finish or by the user pressing the Back button.*

Saving the To-Do List Activity State

Currently, each time the To-Do List example application is restarted, all the to-do items are lost and any text entered into the text entry box is cleared. In this example, you'll start to save the application state of the To-Do list application across sessions.

The instance state in the `ToDoList` Activity consists of three variables:

- Is a new item being added?
- What text exists in the new item entry textbox?
- What is the currently selected item?

Using the Activity's default Shared Preference, you can store each of these values and update the UI when the Activity is restarted.

THE UNIVERSITY OF MICHIGAN LIBRARIES

Later in this chapter, you'll learn how to use the SQLite database to persist the to-do items as well. This example is a first step that shows how to ensure a seamless experience by saving Activity instance details.

1. Start by adding static String variables to use as preference keys.

```
private static final String TEXT_ENTRY_KEY = "TEXT_ENTRY_KEY";
private static final String ADDING_ITEM_KEY = "ADDING_ITEM_KEY";
private static final String SELECTED_INDEX_KEY = "SELECTED_INDEX_KEY";
```

2. Next, override the `onPause` method. Get the Activity's private Shared Preference object, and get its Editor object.

   Using the keys you created in Step 1, store the instance values based on whether a new item is being added and any text in the "new item" Edit Box.

```
@Override
protected void onPause(){
  super.onPause();

  // Get the activity preferences object.
  SharedPreferences uiState = getPreferences(0);
  // Get the preferences editor.
  SharedPreferences.Editor editor = uiState.edit();

  // Add the UI state preference values.
  editor.putString(TEXT_ENTRY_KEY, myEditText.getText().toString());
  editor.putBoolean(ADDING_ITEM_KEY, addingNew);
  // Commit the preferences.
  editor.commit();
}
```

3. Write a `restoreUIState` method that applies the instance values you recorded in Step 2 when the application restarts.

   Modify the `onCreate` method to add a call to the `restoreUIState` method at the very end.

```
@Override
public void onCreate(Bundle icicle) {
  [ ... existing onCreate logic ... ]
  restoreUIState();
}

private void restoreUIState() {
  // Get the activity preferences object.
  SharedPreferences settings = getPreferences(Activity.MODE_PRIVATE);

  // Read the UI state values, specifying default values.
  String text = settings.getString(TEXT_ENTRY_KEY, "");
  Boolean adding = settings.getBoolean(ADDING_ITEM_KEY, false);

  // Restore the UI to the previous state.
  if (adding) {
    addNewItem();
    myEditText.setText(text);
  }
}
```

4. Record the index of the selected item using the `onSaveInstanceState / onRestoreInstanceState` mechanism. It's then only saved and applied if the application is killed without the user's explicit instruction.

```
@Override
public void onSaveInstanceState(Bundle outState) {
  outState.putInt(SELECTED_INDEX_KEY,
                  myListView.getSelectedItemPosition());

  super.onSaveInstanceState(outState);
}

@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
  int pos = -1;

  if (savedInstanceState != null)
    if (savedInstanceState.containsKey(SELECTED_INDEX_KEY))
      pos = savedInstanceState.getInt(SELECTED_INDEX_KEY, -1);

  myListView.setSelection(pos);
}
```

When you run the To-Do List application, you should now see the UI state persisted across sessions. That said, it still won't persist the to-do list items — you'll add this essential piece of functionality later in the chapter.

Creating a Preferences Page for the Earthquake Viewer

In Chapter 5, you created an earthquake monitor that showed a list of recent earthquakes based on an Internet feed.

In the following example, you'll create a Preferences page for this earthquake viewer that lets users configure application settings for a more personalized experience. You'll provide the option to toggle automatic updates, control the frequency of updates, and filter the minimum earthquake magnitude displayed.

Later in this chapter, you'll extend this example further by creating a Content Provider to save and share earthquake data with other applications.

1. Open the Earthquake project you created in Chapter 5.

   Add new String resources for the labels displayed in the "Preferences" screen. Also, add a String for the new Menu Item that will let users access the Preferences screen.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Earthquake</string>
  <string name="quake_feed">
    http://earthquake.usgs.gov/eqcenter/catalogs/1day-M2.5.xml
  </string>
  <string name="menu_update">Refresh Earthquakes</string>
```


THE UNIVERSITY OF MICHIGAN LIBRARIES

```
    <string name="auto_update_prompt">Auto Update?</string>
    <string name="update_freq_prompt">Update Frequency</string>
    <string name="min_quake_mag_prompt">Minimum Quake Magnitude</string>
    <string name="menu_preferences">Preferences</string>
</resources>
```

2. Create a new preferences.xml layout resource that lays out the UI for the `Preferences` Activity. Include a checkbox for indicating the "automatic update" toggle, and spinners to select the update rate and magnitude filter.

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/auto_update_prompt"
  />
  <CheckBox android:id="@+id/checkbox_auto_update"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/update_freq_prompt"
  />
  <Spinner
    android:id="@+id/spinner_update_freq"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/min_quake_mag_prompt"
  />
  <Spinner
    android:id="@+id/spinner_quake_mag"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <Button
```

```
        android:id="@+id/okButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@android:string/ok"
      />
      <Button
        android:id="@+id/cancelButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@android:string/cancel"
      />
    </LinearLayout>
</LinearLayout>
```

3. Create four new array resources in a new `res/values/arrays.xml` file. They will provide the values to use for the update frequency and minimum magnitude filter.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="update_freq_options">
    <item>Every Minute</item>
    <item>5 minutes</item>
    <item>10 minutes</item>
    <item>15 minutes</item>
    <item>Every Hour</item>
  </string-array>

  <array name="magnitude">
    <item>3</item>
    <item>5</item>
    <item>6</item>
    <item>7</item>
    <item>8</item>
  </array>

  <string-array name="magnitude_options">
    <item>3</item>
    <item>5</item>
    <item>6</item>
    <item>7</item>
    <item>8</item>
  </string-array>

  <array name="update_freq_values">
    <item>1</item>
    <item>5</item>
    <item>10</item>
    <item>15</item>
    <item>60</item>
  </array>
</resources>
```

4. Create the `Preferences` Activity. It will be used to display the application preferences.

   Override `onCreate` to inflate the layout you created in Step 2, and get references to the Checkbox and both Spinner controls. Then make a call to the `populateSpinners` stub.

```
package com.paad.earthquake;

import android.app.Activity;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.Spinner;

public class Preferences extends Activity {

  CheckBox autoUpdate;
  Spinner updateFreqSpinner;
  Spinner magnitudeSpinner;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.preferences);

    updateFreqSpinner =
      (Spinner)findViewById(R.id.spinner_update_freq);
    magnitudeSpinner = (Spinner)findViewById(R.id.spinner_quake_mag);
    autoUpdate = (CheckBox)findViewById(R.id.checkbox_auto_update);

    populateSpinners();
  }

  private void populateSpinners() {
  }
}
```

5. Fill in the `populateSpinners` method, using Array Adapters to bind each Spinner to its corresponding array.

```
private void populateSpinners() {
  // Populate the update frequency spinner
  ArrayAdapter<CharSequence> fAdapter;
  fAdapter = ArrayAdapter.createFromResource(this,
                                  R.array.update_freq_options,
                                  android.R.layout.simple_spinner_item);
  fAdapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);
```

```
    updateFreqSpinner.setAdapter(fAdapter);

    // Populate the minimum magnitude spinner
    ArrayAdapter<CharSequence> mAdapter;
    mAdapter = ArrayAdapter.createFromResource(this,
                                    R.array.magnitude_options,
                                    android.R.layout.simple_spinner_item);
    mAdapter.setDropDownViewResource(
      android.R.layout.simple_spinner_dropdown_item);

    magnitudeSpinner.setAdapter(mAdapter);
  }
```

6. Add public static String values to use to identify the named Shared Preference you're going to create, and the keys it will use to store each preference value. Update the `onCreate` method to retrieve the named preference and call `updateUIFromPreferences`. The `updateUIFromPreferences` method uses the `get<type>` methods on the Shared Preference object to retrieve each preference value and apply it to the current UI.

```
  public static final String USER_PREFERENCE = "USER_PREFERENCES";

  public static final String PREF_AUTO_UPDATE = "PREF_AUTO_UPDATE";
  public static final String PREF_MIN_MAG = "PREF_MIN_MAG";
  public static final String PREF_UPDATE_FREQ = "PREF_UPDATE_FREQ";

  SharedPreferences prefs;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.preferences);

    updateFreqSpinner = (Spinner)findViewById(R.id.spinner_update_freq);
    magnitudeSpinner = (Spinner)findViewById(R.id.spinner_quake_mag);
    autoUpdate = (CheckBox)findViewById(R.id.checkbox_auto_update);

    populateSpinners();

    prefs = getSharedPreferences(USER_PREFERENCE, Activity.MODE_PRIVATE);
    updateUIFromPreferences();
  }

  private void updateUIFromPreferences() {
    boolean autoUpChecked = prefs.getBoolean(PREF_AUTO_UPDATE, false);
    int updateFreqIndex = prefs.getInt(PREF_UPDATE_FREQ, 2);
    int minMagIndex = prefs.getInt(PREF_MIN_MAG, 0);

    updateFreqSpinner.setSelection(updateFreqIndex);
    magnitudeSpinner.setSelection(minMagIndex);
    autoUpdate.setChecked(autoUpChecked);
  }
```

7. Still in the `onCreate` method, add event handlers for the OK and Cancel buttons. Cancel should close the Activity, while OK should call `savePreferences` first.

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.preferences);

  updateFreqSpinner = (Spinner)findViewById(R.id.spinner_update_freq);
  magnitudeSpinner = (Spinner)findViewById(R.id.spinner_quake_mag);
  autoUpdate = (CheckBox)findViewById(R.id.checkbox_auto_update);

  populateSpinners();

  prefs = getSharedPreferences(USER_PREFERENCE, Activity.MODE_PRIVATE);
  updateUIFromPreferences();

  Button okButton = (Button) findViewById(R.id.okButton);
  okButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {
      savePreferences();
      Preferences.this.setResult(RESULT_OK);
      finish();
    }
  });

  Button cancelButton = (Button) findViewById(R.id.cancelButton);
  cancelButton.setOnClickListener(new View.OnClickListener() {

    public void onClick(View view) {
      Preferences.this.setResult(RESULT_CANCELED);
      finish();
    }
  });
}

private void savePreferences() {
}
```

8. Fill in the `savePreferences` method to record the current preferences, based on the UI selections, to the Shared Preference object.

```
private void savePreferences() {
  int updateIndex = updateFreqSpinner.getSelectedItemPosition();
  int minMagIndex = magnitudeSpinner.getSelectedItemPosition();
  boolean autoUpdateChecked = autoUpdate.isChecked();

  Editor editor = prefs.edit();
  editor.putBoolean(PREF_AUTO_UPDATE, autoUpdateChecked);
  editor.putInt(PREF_UPDATE_FREQ, updateIndex);
  editor.putInt(PREF_MIN_MAG, minMagIndex);
  editor.commit();
}
```

9. That completes the `Preferences` Activity. Make it accessible in the application by adding it to the application manifest.

```
<activity android:name=".Preferences"
          android:label="Earthquake Preferences">
</activity>
```

10. Now return to the `Earthquake` Activity, and add support for the new Shared Preferences file and a Menu Item to display the Preferences Activity.

Start by adding the new Menu Item. Extend the `onCreateOptionsMenu` method to include a new item that opens the Preferences Activity.

```
static final private int MENU_PREFERENCES = Menu.FIRST+1;

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

  menu.add(0, MENU_UPDATE, Menu.NONE, R.string.menu_update);
  menu.add(0, MENU_PREFERENCES, Menu.NONE, R.string.menu_preferences);

  return true;
}
```

11. Modify the `onOptionsItemSelected` method to display the Preferences Activity when the new Menu Item is selected. Create an explicit Intent, and pass it in to the `startActivityForResult` method. This will launch the Preferences screen and alert the `Earthquake` class when the preferences are saved through the `onActivityResult` handler.

```
private static final int SHOW_PREFERENCES = 1;

public boolean onOptionsItemSelected(MenuItem item) {
  super.onOptionsItemSelected(item);

  switch (item.getItemId()) {
    case (MENU_UPDATE): {
      refreshEarthquakes();
      return true;
    }
    case (MENU_PREFERENCES): {
      Intent i = new Intent(this, Preferences.class);
      startActivityForResult(i, SHOW_PREFERENCES);
      return true;
    }
  }
  return false;
}
```

12. Launch your application, and select **Preferences** from the Activity menu. The Preferences Activity should be displayed as shown in Figure 6-1.

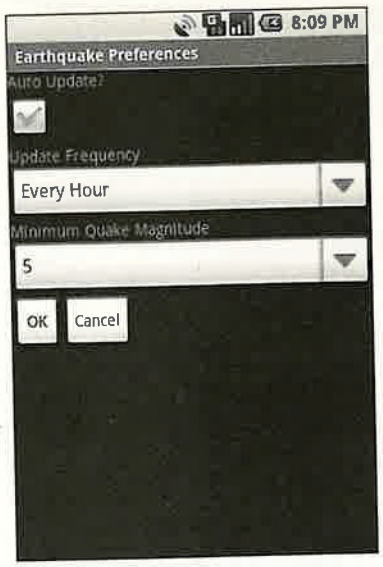


Figure 6-1

13. All that's left is to apply the preferences to the Earthquake functionality.

    Implementing the automatic updates will be left until Chapter 8, when you'll learn how to use Services and background threads. For now, you can put the framework in place and apply the magnitude filter.

14. Start by creating a new `updateFromPreferences` method that reads the Shared Preference values and creates instance variables for each of them.

```
int minimumMagnitude = 0;
boolean autoUpdate = false;
int updateFreq = 0;

private void updateFromPreferences() {
  SharedPreferences prefs =
    getSharedPreferences(Preferences.USER_PREFERENCE,
                         Activity.MODE_PRIVATE);

  int minMagIndex = prefs.getInt(Preferences.PREF_MIN_MAG, 0);
  if (minMagIndex < 0)
    minMagIndex = 0;

  int freqIndex = prefs.getInt(Preferences.PREF_UPDATE_FREQ, 0);
  if (freqIndex < 0)
    freqIndex = 0;

  autoUpdate = prefs.getBoolean(Preferences.PREF_AUTO_UPDATE, false);

  Resources r = getResources();
```

```
    // Get the option values from the arrays.
    int[] minMagValues = r.getIntArray(R.array.magnitude);
    int[] freqValues = r.getIntArray(R.array.update_freq_values);

    // Convert the values to ints.
    minimumMagnitude = minMagValues[minMagIndex];
    updateFreq = freqValues[freqIndex];
  }
```

15. Apply the magnitude filter by updating the `addNewQuake` method to check a new earthquake's magnitude before adding it to the list.

```
private void addNewQuake(Quake _quake) {
  if (_quake.getMagnitude() > minimumMagnitude) {
    // Add the new quake to our list of earthquakes.
    earthquakes.add(_quake);

    // Notify the array adapter of a change.
    aa.notifyDataSetChanged();
  }
}
```

16. Override the `onActivityResult` handler to call `updateFromPreferences` and refresh the earthquakes whenever the Preferences Activity saves changes.

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                             Intent data) {
  super.onActivityResult(requestCode, resultCode, data);

  if (requestCode == SHOW_PREFERENCES)
    if (resultCode == Activity.RESULT_OK) {
      updateFromPreferences();
      refreshEarthquakes();
    }
}
```

17. Finally, call `updateFromPreferences` in `onCreate` (before the call to `refreshEarthquakes`) to ensure that the preferences are applied when the Activity first starts.

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

  earthquakeListView =
   (ListView)this.findViewById(R.id.earthquakeListView);

  earthquakeListView.setOnItemClickListener(new OnItemClickListener() {

    public void onItemClick(AdapterView _av, View _v,
                            int _index, long arg3) {
      selectedQuake = earthquakes.get(_index);
      showDialog(QUAKE_DIALOG);
```

```
    }
  });

  int layoutID = android.R.layout.simple_list_item_1;
  aa = new ArrayAdapter<Quake>(this, layoutID, earthquakes);
  earthquakeListView.setAdapter(aa);

  updateFromPreferences();
  refreshEarthquakes();
}
```

Saving and Loading Files

It's good practice to use Shared Preferences or a database to store your application data, but there are still times when you'll want to use files directly rather than rely on Android's managed mechanisms.

As well as the standard Java I/O classes and methods, Android offers `openFileInput` and `openFileOuput` to simplify reading and writing streams from and to local files, as shown in the code snippet below:

```
String FILE_NAME = "tempfile.tmp";

// Create a new output file stream that's private to this application.
FileOutputStream fos = openFileOutput(FILE_NAME, Context.MODE_PRIVATE);
// Create a new file input stream.
FileInputStream fis = openFileInput(FILE_NAME);
```

These methods only support files in the current application folder; specifying path separators will cause an exception to be thrown.

If the filename you specify when creating a `FileOutputStream` does not exist, Android will create it for you. The default behavior for existing files is to overwrite them; to append an existing file, specify the mode as `Context.MODE_APPEND`.

By default, files created using the `openFileOutput` method are private to the calling application — a different application that tries to access these files will be denied access. The standard way to share a file between applications is to use a Content Provider. Alternatively, you can specify either `Context.MODE_WORLD_READABLE` or `Context.MODE_WORLD_WRITEABLE` when creating the output file to make them available in other applications, as shown in the following snippet:

```
String OUTPUT_FILE = "publicCopy.txt";
FileOutputStream fos = openFileOutput(OUTPUT_FILE, Context.MODE_WORLD_WRITEABLE);
```

Including Static Files as Resources

If your application requires external file resources, you can include them in your distribution package by placing them in the `res/raw` folder of your project hierarchy.

To access these Read Only file resources, call the `openRawResource` method from your application's `Resource` object to receive an `InputStream` based on the specified resource. Pass in the filename (without extension) as the variable name from the `R.raw` class, as shown in the skeleton code below:

```
Resources myResources = getResources();
InputStream myFile = myResources.openRawResource(R.raw.myfilename);
```

Adding raw files to your resources hierarchy is an excellent alternative for large, preexisting data sources (such as dictionaries) where it's not desirable (or even possible) to convert them into an Android database.

Android's resource mechanism lets you specify alternative resource files for different languages, locations, or hardware configurations. As a result, you could, for example, create an application that dynamically loads a dictionary resource based on the user's current settings.

File Management Tools

Android supplies some basic file management tools to help you deal with the filesystem. Many of these utilities are located within the standard `java.io.File` package.

Complete coverage of Java file management utilities is beyond the scope of this book, but Android does supply some specialized utilities for file management available from the application's Context.

- **`deleteFile`** Lets you remove files created by the current application.
- **`fileList`** Returns a `String` array that includes all the files created by the current application.

Databases in Android

Android provides full relational database capabilities through the SQLite library, without imposing any additional limitations.

Using SQLite, you can create independent, relational databases for each application. Use them to store and manage complex, structured application data.

All Android databases are stored in the `/data/data/<package_name>/databases` folder on your device (or emulator). By default, all databases are private, accessible only by the application that created them. To share a database across applications, use Content Providers, as shown later in this chapter.

Database design is a vast topic that deserves more thorough coverage than is possible within this book. However, it's worth highlighting that standard database best practices still apply. In particular, when creating databases for resource-constrained devices, it's important to reduce data redundancy using normalization.

The following sections focus on the practicalities of creating and managing SQLite databases in Android.

Introducing SQLite

SQLite is a relational database management system (RDBMS). It is well regarded, being:

- Open source
- Standards-compliant
- Lightweight
- Single-tier

It has been implemented as a compact C library that's included as part of the Android software stack.

By providing functionality through a library, rather than as a separate process, each database becomes an integrated part of the application that created it. This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.

SQLite has a reputation of being extremely reliable and is the database system of choice for many consumer electronic devices, including several MP3 players, the iPhone, and the iPod Touch.

Lightweight and powerful, SQLite differs from many conventional database engines by using a loosely typed approach to column definitions. Rather than requiring column values to conform to a single type, the values in each row for each column are individually typed. As a result, there's no strict type checking when assigning or extracting values from each column within a row.

For more comprehensive coverage of SQLite, including its particular strengths and limitations, check out the official site at `www.sqlite.org/`.

Cursors and Content Values

`ContentValues` objects are used to insert new rows into database tables (and Content Providers). Each Content Values object represents a single row, as a map of column names to values.

Queries in Android are returned as `Cursor` objects. Rather than extracting and returning a copy of the result values, `Cursors` act as pointers to a subset of the underlying data. `Cursors` are a managed way of controlling your position (row) in the result set of a database query.

The `Cursor` class includes several functions to navigate query results including, but not limited to, the following:

- `moveToFirst` Moves the cursor to the first row in the query result.
- `moveToNext` Moves the cursor to the next row.
- `moveToPrevious` Moves the cursor to the previous row.
- `getCount` Returns the number of rows in the result set.
- `getColumnIndexOrThrow` Returns an index for the column with the specified name (throwing an exception if no column exists with that name).
- `getColumnName` Returns the name of the specified column index.
- `getColumnNames` Returns a `String` array of all the column names in the current cursor.

- `moveToPosition` Moves the cursor to the specified row.
- `getPosition` Returns the current cursor position.

Android provides a mechanism to manage Cursor resources within your Activities. The `startManagingCursor` method integrates the Cursor's lifetime into the parent Activity's lifetime management. When you've finished with the Cursor, call `stopManagingCursor` to do just that.

Later in this chapter, you'll learn how to query a database and how to extract specific row/column values from the resulting `Cursor` objects.

Working with Android Databases

It's good practice to create a helper class to simplify your database interactions.

Consider creating a database adapter, which adds an abstraction layer that encapsulates database interactions. It should provide intuitive, strongly typed methods for adding, removing, and updating items. A database adapter should also handle queries and wrap creating, opening, and closing the database.

It's often also used as a convenient location from which to publish static database constants, including table names, column names, and column indexes.

The following snippet shows the skeleton code for a standard database adapter class. It includes an extension of the `SQLiteOpenHelper` class, used to simplify opening, creating, and upgrading the database.

```
import android.content.Context;
import android.database.*;
import android.database.sqlite.*;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.util.Log;

public class MyDBAdapter {
  private static final String DATABASE_NAME = "myDatabase.db";
  private static final String DATABASE_TABLE = "mainTable";
  private static final int DATABASE_VERSION = 1;

  // The index (key) column name for use in where clauses.
  public static final String KEY_ID="_id";

  // The name and column index of each column in your database.
  public static final String KEY_NAME="name";
  public static final int NAME_COLUMN = 1;
  // TODO: Create public field for each column in your table.

  // SQL Statement to create a new database.
  private static final String DATABASE_CREATE = "create table " +
    DATABASE_TABLE + " (" + KEY_ID +
    " integer primary key autoincrement, " +
    KEY_NAME + " text not null);";

  // Variable to hold the database instance
  private SQLiteDatabase db;

  // Context of the application using the database.
```

```
private final Context context;
// Database open/upgrade helper
private myDbHelper dbHelper;

public MyDBAdapter(Context _context) {
  context = _context;
  dbHelper = new myDbHelper(context, DATABASE_NAME, null,
                            DATABASE_VERSION);
}

public MyDBAdapter open() throws SQLException {
  db = dbHelper.getWritableDatabase();
  return this;
}

public void close() {
    db.close();
}

public long insertEntry(MyObject _myObject) {
  ContentValues contentValues = new ContentValues();
  // TODO fill in ContentValues to represent the new row
  return db.insert(DATABASE_TABLE, null, contentValues);
}

public boolean removeEntry(long _rowIndex) {
  return db.delete(DATABASE_TABLE, KEY_ID +
                   "=" + _rowIndex, null) > 0;
}

public Cursor getAllEntries () {
  return db.query(DATABASE_TABLE, new String[] {KEY_ID, KEY_NAME},
                  null, null, null, null, null);
}

public MyObject getEntry(long _rowIndex) {
  MyObject objectInstance = new MyObject();
  // TODO Return a cursor to a row from the database and
  // use the values to populate an instance of MyObject
  return objectInstance;
}

public int updateEntry(long _rowIndex, MyObject _myObject) {
  String where = KEY_ID + "=" + _rowIndex;
  ContentValues contentValues = new ContentValues();
  // TODO fill in the ContentValue based on the new object
  return db.update(DATABASE_TABLE, contentValues, where, null);
}

private static class myDbHelper extends SQLiteOpenHelper {

  public myDbHelper(Context context, String name,
                    CursorFactory factory, int version) {
```

```
      super(context, name, factory, version);
    }

    // Called when no database exists in
    // disk and the helper class needs
    // to create a new one.
    @Override
    public void onCreate(SQLiteDatabase _db) {
      _db.execSQL(DATABASE_CREATE);
    }

    // Called when there is a database version mismatch meaning that
    // the version of the database on disk needs to be upgraded to
    // the current version.
    @Override
    public void onUpgrade(SQLiteDatabase _db, int _oldVersion,
                          int _newVersion) {
      // Log the version upgrade.
      Log.w("TaskDBAdapter", "Upgrading from version " +
                             _oldVersion + " to " +
                             _newVersion +
                             ", which will destroy all old data");

      // Upgrade the existing database to conform to the new version.
      // Multiple previous versions can be handled by comparing
      // _oldVersion and _newVersion values.

      // The simplest case is to drop the old table and create a
      // new one.
      _db.execSQL("DROP TABLE IF EXISTS " + DATABASE_TABLE);
      // Create a new one.
      onCreate(_db);
    }
  }
}
```

Using the `SQLiteOpenHelper`

`SQLiteOpenHelper` is an abstract class that wraps up the best practice pattern for creating, opening, and upgrading databases. By implementing and using an `SQLiteOpenHelper`, you hide the logic used to decide if a database needs to be created or upgraded before it's opened.

The code snippet above shows how to extend the `SQLiteOpenHelper` class by overriding the constructor, `onCreate`, and `onUpgrade` methods to handle the creation of a new database and upgrading to a new version, respectively.

> *In the previous example,* `onUpgrade` *simply drops the existing table and replaces it with the new definition. In practice, a better solution is to migrate existing data into the new table.*

To use an implementation of the helper class, create a new instance, passing in the context, database name, current version, and a `CursorFactory` (if you're using one).

Call `getReadableDatabase` or `getWriteableDatabase` to open and return a readable/writable instance of the database.

A call to `getWriteableDatabase` can fail because of disk space or permission issues, so it's good practice to provide fallback to the `getReadableDatabase` method as shown below:

```
dbHelper = new myDbHelper(context, DATABASE_NAME, null, DATABASE_VERSION);

SQLiteDatabase db;
try {
  db = dbHelper.getWritableDatabase();
}
catch (SQLiteException ex){
  db = dbHelper.getReadableDatabase();
}
```

Behind the scenes, if the database doesn't exist, the helper executes its `onCreate` handler. If the database version has changed, the `onUpgrade` handler will fire. In both cases, the `get<read/write>ableDatabase` call will return the existing, newly created, or upgraded database as appropriate.

Opening and Creating Databases without the `SQLiteHelper`

You can create and open databases without using the `SQLiteHelper` class with the `openOrCreateDatabase` method on the application Context.

Setting up a database is a two-step process. First, call `openOrCreateDatabase` to create the new database. Then, call `execSQL` on the resulting database instance to run the SQL commands that will create your tables and their relationships. The general process is shown in the snippet below:

```
private static final String DATABASE_NAME = "myDatabase.db";
private static final String DATABASE_TABLE = "mainTable";

private static final String DATABASE_CREATE =
  "create table " + DATABASE_TABLE +
  " ( _id integer primary key autoincrement," +
  "column_one text not null);";

SQLiteDatabase myDatabase;

private void createDatabase() {
  myDatabase = openOrCreateDatabase(DATABASE_NAME,
                                    Context.MODE_PRIVATE, null);
  myDatabase.execSQL(DATABASE_CREATE);
}
```

Android Database Design Considerations

There are several considerations specific to Android that you should consider when designing your database:

- Files (such as bitmaps or audio files) are not usually stored within database tables. Instead, use a string to store a path to the file, preferably a fully qualified Content Provider URI.
- While not strictly a requirement, it's strongly recommended that all tables include an auto-increment key field, to function as a unique index value for each row. It's worth noting that if you plan to share your table using a Content Provider, this unique ID field is mandatory.

Querying Your Database

All database queries are returned as a `Cursor` to a result set. This lets Android manage resources more efficiently by retrieving and releasing row and column values on demand.

To execute a query on a database, use the `query` method on the database object, passing in:

- An optional Boolean that specifies if the result set should contain only unique values
- The name of the table to query
- A projection, as an array of `Strings`, that lists the columns to include in the result set
- A "where" clause that defines the rows to be returned. You can include `?` wildcards that will be replaced by the values stored in the selection argument parameter.
- An array of selection argument strings that will replace the `?`'s in the "where" clause
- A "group by" clause that defines how the resulting rows will be grouped
- A "having" filter that defines which row groups to include if you specified a "group by" clause
- A `String` that describes the order of the returned rows
- An optional `String` that defines a limit to the returned rows

The following skeleton code shows snippets for returning some, and all, of the rows in a particular table:

```
// Return all rows for columns one and three, no duplicates
String[] result_columns = new String[] {KEY_ID, KEY_COL1, KEY_COL3};

Cursor allRows = myDatabase.query(true, DATABASE_TABLE, result_columns,
                                  null, null, null, null, null, null);

// Return all columns for rows where column 3 equals a set value
// and the rows are ordered by column 5.
String where = KEY_COL3 + "=" + requiredValue;
String order = KEY_COL5;
Cursor myResult = myDatabase.query(DATABASE_TABLE, null, where,
                                   null, null, null, order);
```

In practice, it's often useful to abstract these query commands within an adapter class to simplify data access.

Extracting Results from a Cursor

To extract actual values from a result Cursor, first use the `moveTo<location>` methods described previously to position the Cursor at the correct row of the result set.

With the Cursor at the desired row, use the type-safe `get` methods (passing in a column index) to return the value stored at the current row for the specified column, as shown in the following snippet:

```
String columnValue = myResult.getString(columnIndex);
```

Database implementations should publish static constants that provide the column indexes using more easily recognizable variables based on the column names. They are generally exposed within a database adapter as described previously.

The following example shows how to iterate over a result cursor, extracting and summing a column of floats:

```
int GOLD_HOARDED_COLUMN = 2;
Cursor myGold = myDatabase.query("GoldHoards", null, null, null, null,
                                 null, null);
float totalHoard = 0f;

// Make sure there is at least one row.
if (myGold.moveToFirst()) {
  // Iterate over each cursor.
  do {
    float hoard = myGold.getFloat(GOLD_HOARDED_COLUMN);
    totalHoard += hoard;
  } while(myGold.moveToNext());
}

float averageHoard = totalHoard / myGold.getCount();
```

Because SQLite database columns are loosely typed, you can cast individual values into valid types as required. For example, values stored as `floats` can be read back as `Strings`.

Adding, Updating, and Removing Rows

The `SQLiteDatabase` class exposes specialized `insert`, `delete`, and `update` methods to encapsulate the SQL statements required to perform these actions. Nonetheless, the `execSQL` method lets you execute any valid SQL on your database tables should you want to execute these operations manually.

Any time you modify the underlying database values, you should call `refreshQuery` on any Cursors that currently have a view on the table.

Inserting New Rows

To create a new row, construct a `ContentValues` object, and use its `put` methods to supply values for each column. Insert the new row by passing the Content Values object into the `insert` method called on the target database object — along with the table name — as shown in the snippet below:

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Assign values for each row.
newValues.put(COLUMN_NAME, newValue);
[ ... Repeat for each column ... ]

// Insert the row into your table
myDatabase.insert(DATABASE_TABLE, null, newValues);
```

Updating a Row on the Database

Updating rows is also done using Content Values.

Create a new `ContentValues` object, using the `put` methods to assign new values to each column you want to update. Call `update` on the database object, passing in the table name, the updated Content Values object, and a `where` statement that returns the row(s) to update.

The update process is demonstrated in the snippet below:

```
// Define the updated row content.
ContentValues updatedValues = new ContentValues();

// Assign values for each row.
updatedValues.put(COLUMN_NAME, newValue);
[ ... Repeat for each column ... ]

String where = KEY_ID + "=" + rowId;

// Update the row with the specified index with the new values.
myDatabase.update(DATABASE_TABLE, updatedValues, where, null);
```

Deleting Rows

To delete a row, simply call `delete` on your database object, specifying the table name and a `where` clause that returns the rows you want to delete, as shown in the code below:

```
myDatabase.delete(DATABASE_TABLE, KEY_ID + "=" + rowId, null);
```

Saving Your To-Do List

Previously in this chapter, you enhanced the To-Do List example to persist the Activity's UI state across sessions. That was only half the job; in the following example, you'll create a private database to save the to-do items:

1. Start by creating a new `ToDoDBAdapter` class. It will be used to manage your database interactions.

   Create private variables to store the `SQLiteDatabase` object and the Context of the calling application. Add a constructor that takes the owner application's Context, and include static class variables for the name and version of the database and a name for the to-do item table.

```
package com.paad.todolist;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class ToDoDBAdapter {
  private static final String DATABASE_NAME = "todoList.db";
  private static final String DATABASE_TABLE = "todoItems";
  private static final int DATABASE_VERSION = 1;

  private SQLiteDatabase db;
```

```
    private final Context context;

    public ToDoDBAdapter(Context _context) {
        this.context = _context;
    }
}
```

2. Create public convenience variables that define the column names and indexes; this will make it easier to find the correct columns when extracting values from query result Cursors.

```
public static final String KEY_ID = "_id";

public static final String KEY_TASK = "task";
public static final int TASK_COLUMN = 1;

public static final String KEY_CREATION_DATE = "creation_date";
public static final int CREATION_DATE_COLUMN = 2;
```

3. Create a new `taskDBOpenHelper` class within the `ToDoDBAdapter` that extends `SQLiteOpenHelper`. It will be used to simplify version management of your database.

   Within it, overwrite the `onCreate` and `onUpgrade` methods to handle the database creation and upgrade logic.

```
private static class toDoDBOpenHelper extends SQLiteOpenHelper {

  public toDoDBOpenHelper(Context context, String name,
                          CursorFactory factory, int version) {
    super(context, name, factory, version);
  }

  // SQL Statement to create a new database.
  private static final String DATABASE_CREATE = "create table " +
    DATABASE_TABLE + " (" + KEY_ID +
    " integer primary key autoincrement, " +
    KEY_TASK + " text not null, " + KEY_CREATION_DATE + " long);";

  @Override
  public void onCreate(SQLiteDatabase _db) {
    _db.execSQL(DATABASE_CREATE);
  }

  @Override
  public void onUpgrade(SQLiteDatabase _db, int _oldVersion,
                        int _newVersion) {
    Log.w("TaskDBAdapter", "Upgrading from version " +
                           _oldVersion + " to " +
                           _newVersion +
                           ", which will destroy all old data");

    // Drop the old table.
    _db.execSQL("DROP TABLE IF EXISTS " + DATABASE_TABLE);
    // Create a new one.
    onCreate(_db);
  }
}
```

4. Within the `ToDoDBAdapter` class, add a private instance variable to store an instance of the `toDoDBOpenHelper` class you just created; assign it within the constructor.

```
private toDoDBOpenHelper dbHelper;

public ToDoDBAdapter(Context _context) {
  this.context = _context;
  dbHelper = new toDoDBOpenHelper(context, DATABASE_NAME,
                                  null, DATABASE_VERSION);
}
```

5. Still in the adapter class, create `open` and `close` methods that encapsulate the open and close logic for your database. Start with a `close` method that simply calls close on the database object.

```
public void close() {
  db.close();
}
```

6. The `open` method should use the `toDoDBOpenHelper` class. Call `getWritableDatabase` to let the helper handle database creation and version checking. Wrap the call to try to provide a readable database if a writable instance can't be opened.

```
public void open() throws SQLiteException {
  try {
    db = dbHelper.getWritableDatabase();
  } catch (SQLiteException ex) {
    db = dbHelper.getReadableDatabase();
  }
}
```

7. Add strongly typed methods for adding, removing, and updating items.

```
// Insert a new task
public long insertTask(ToDoItem _task) {
  // Create a new row of values to insert.
  ContentValues newTaskValues = new ContentValues();
  // Assign values for each row.
  newTaskValues.put(KEY_TASK, _task.getTask());
  newTaskValues.put(KEY_CREATION_DATE, _task.getCreated().getTime());
  // Insert the row.
  return db.insert(DATABASE_TABLE, null, newTaskValues);
}

// Remove a task based on its index
public boolean removeTask(long _rowIndex) {
  return db.delete(DATABASE_TABLE, KEY_ID + "=" + _rowIndex, null) > 0;
}

// Update a task
public boolean updateTask(long _rowIndex, String _task) {
  ContentValues newValue = new ContentValues();
  newValue.put(KEY_TASK, _task);
  return db.update(DATABASE_TABLE, newValue,
                   KEY_ID + "=" + _rowIndex, null) > 0;
}
```

8. Now add helper methods to handle queries. Write three methods — one to return all the items, another to return a particular row as a Cursor, and finally, one that returns a strongly typed `ToDoItem`.

```
public Cursor getAllToDoItemsCursor() {
  return db.query(DATABASE_TABLE,
                  new String[] { KEY_ID, KEY_TASK, KEY_CREATION_DATE},
                  null, null, null, null, null);
}

public Cursor setCursorToToDoItem(long _rowIndex) throws SQLException {
  Cursor result = db.query(true, DATABASE_TABLE,
                           new String[] {KEY_ID, KEY_TASK},
                           KEY_ID + "=" + _rowIndex, null, null, null,
                           null, null);
  if ((result.getCount() == 0) || !result.moveToFirst()) {
    throw new SQLException("No to do items found for row: " +
                           _rowIndex);
  }
  return result;
}

public ToDoItem getToDoItem(long _rowIndex) throws SQLException {
  Cursor cursor = db.query(true, DATABASE_TABLE,
                           new String[] {KEY_ID, KEY_TASK},
                           KEY_ID + "=" + _rowIndex,
                           null, null, null, null, null);
  if ((cursor.getCount() == 0) || !cursor.moveToFirst()) {
    throw new SQLException("No to do item found for row: " +
                           _rowIndex);
  }

  String task = cursor.getString(TASK_COLUMN);
  long created = cursor.getLong(CREATION_DATE_COLUMN);

  ToDoItem result = new ToDoItem(task, new Date(created));
  return result;
}
```

9. That completes the database helper class. Return the `ToDoList` Activity, and update it to persist the to-do list array.

Start by updating the Activity's `onCreate` method to create an instance of the `toDoDBAdapter`, and open a connection to the database. Also include a call to the `populateTodoList` method stub.

```
ToDoDBAdapter toDoDBAdapter;

public void onCreate(Bundle icicle) {
  [ ... existing onCreate logic ... ]

  toDoDBAdapter = new ToDoDBAdapter(this);
```

```
    // Open or create the database
    toDoDBAdapter.open();

    populateTodoList();
  }

  private void populateTodoList() { }
```

10. Create a new instance variable to store a Cursor over all the to-do items in the database.

    Update the `populateTodoList` method to use the `toDoDBAdapter` instance to query the database, and call `startManagingCursor` to let the Activity manage the Cursor. It should also make a call to `updateArray`, a method that will be used to repopulate the to-do list array using the Cursor.

```
  Cursor toDoListCursor;

  private void populateTodoList() {
    // Get all the todo list items from the database.
    toDoListCursor = toDoDBAdapter.getAllToDoItemsCursor();
    startManagingCursor(toDoListCursor);

    // Update the array.
    updateArray();
  }

  private void updateArray() { }
```

11. Now implement the `updateArray` method to update the current to-do list array. Call `requery` on the result Cursor to ensure that it's fully up to date, then clear the array and iterate over the result set. When complete, call `notifyDataSetChanged` on the Array Adapter.

```
  private void updateArray() {
    toDoListCursor.requery();

    todoItems.clear();

    if (toDoListCursor.moveToFirst())
      do {
        String task =
          toDoListCursor.getString(ToDoDBAdapter.TASK_COLUMN);
        long created =
          toDoListCursor.getLong(ToDoDBAdapter.CREATION_DATE_COLUMN);

        ToDoItem newItem = new ToDoItem(task, new Date(created));
        todoItems.add(0, newItem);
      } while(toDoListCursor.moveToNext());

    aa.notifyDataSetChanged();
  }
```

12. To join the pieces together, modify the `OnKeyListener` assigned to the text entry box in the `onCreate` method, and update the `removeItem` method. Both should now use the `toDoDBAdapter` to add and remove items from the database rather than modifying the to-do list array directly.

12.1. Start with the `OnKeyListener`, insert the new item into the database, and refresh the array.

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);

  myListView = (ListView)findViewById(R.id.myListView);
  myEditText = (EditText)findViewById(R.id.myEditText);

  todoItems = new ArrayList<ToDoItem>();
  int resID = R.layout.todolist_item;
  aa = new ToDoItemAdapter(this, resID, todoItems);
  myListView.setAdapter(aa);

  myEditText.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent event) {
      if (event.getAction() == KeyEvent.ACTION_DOWN)
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
          ToDoItem newItem;
          newItem = new ToDoItem(myEditText.getText().toString());
          toDoDBAdapter.insertTask(newItem);
          updateArray();
          myEditText.setText("");
          aa.notifyDataSetChanged();
          cancelAdd();
          return true;
        }
      return false;
    }
  });

  registerForContextMenu(myListView);
  restoreUIState();

  toDoDBAdapter = new ToDoDBAdapter(this);

  // Open or create the database
  toDoDBAdapter.open();

  populateTodoList();
}
```

12.2. Then modify the `removeItem` method to remove the item from the database and refresh the array list.

```
private void removeItem(int _index) {
  // Items are added to the listview in reverse order,
  // so invert the index.
```

```
    toDoDBAdapter.removeTask(todoItems.size()-_index);
    updateArray();
  }
```

13. As a final step, override the `onDestroy` method of your Activity to close your database connection.

```
@Override
public void onDestroy() {

  // Close the database
  toDoDBAdapter.close();

 super.onDestroy();
}
```

Your to-do items will now be saved between sessions. As a further enhancement, you could change the Array Adapter to a Cursor Adapter and have the List View update dynamically, directly from changes to the underlying database.

By using a private database, your tasks are not available for other applications to view or add to them. To provide access to your tasks for other applications to leverage, you can expose them using a Content Provider.

Introducing Content Providers

Content Providers are a generic interface mechanism that lets you share data between applications. By abstracting away the underlying data source, Content Providers let you decouple your application layer from the data layer, making your applications data-source agnostic.

Content Providers feature full permission control and are accessed using a simple URI model. Shared content can be queried for results as well as supporting write access. As a result, any application with the appropriate permissions can add, remove, and update data from any other applications — including some native Android databases.

Many of the native databases have been made available as Content Providers, accessible by third-party applications. This means that your applications can have access to the phone's Contact Manager, media player, and other native database once they've been granted permission.

By publishing your own data sources as Content Providers, you make it possible for you (and other developers) to incorporate and extend your data in new applications.

Using Content Providers

Access to Content Providers is handled by the `ContentResolver` class.

The following sections demonstrate how to access a Content Resolver and how to use it to query and transact with a Content Provider. They also demonstrate some practical examples using the native Android Content Providers.

Introducing Content Resolvers

Each application Context has a single `ContentResolver`, accessible using the `getContentResolver` method, as shown in the following code snippet:

```
ContentResolver cr = getContentResolver();
```

Content Resolver includes several methods to transact and query Content Providers. You specify the provider to interact using a URI.

A Content Provider's URI is defined by its *authority* as defined in its application manifest node. An authority URI is an arbitrary string, so most providers expose a `CONTENT_URI` property that includes its authority.

Content Providers usually expose two forms of URI, one for requests against all the data and another that specifies only a single row. The form for the latter appends `/<rowID>` to the standard `CONTENT_URI`.

Querying for Content

As in databases, query results are returned as Cursors over a result set. You can extract values from the cursor using the techniques described previously within the database section on "Extracting Results from a Cursor."

Content Provider queries take a very similar form to database queries. Using the `query` method on the `ContentResolver` object, pass in:

- The URI of the content provider data you want to query
- A projection that represents the columns you want to include in the result set
- A `where` clause that defines the rows to be returned. You can include `?` wild cards that will be replaced by the values stored in the selection argument parameter.
- An array of selection argument strings that will replace the `?`'s in the `where` clause
- A string that describes the order of the returned rows

The following skeleton code demonstrates the use of a Content Resolver to apply a query to a Content Provider:

```
// Return all rows
Cursor allRows = getContentResolver().query(MyProvider.CONTENT_URI,
                                            null, null, null, null);

// Return all columns for rows where column 3 equals a set value
// and the rows are ordered by column 5.
String where = KEY_COL3 + "=" + requiredValue;
String order = KEY_COL5;
Cursor someRows = getContentResolver().query(MyProvider.CONTENT_URI,
                                             null, where, null, order);
```

You'll see some more practical examples of querying for content later in this chapter when the native Android content providers are introduced.

Adding, Updating, and Deleting Content

To perform transactions on Content Providers, use the `delete`, `update`, and `insert` methods on the `ContentResolver` object.

Inserts

The Content Resolver offers two methods for inserting new records into your Content Provider — `insert` and `bulkInsert`. Both methods accept the URI of the item type you're adding; where the former takes a single new `ContentValues` object, the latter takes an array.

The simple `insert` method will return a URI to the newly added record, while `bulkInsert` returns the number of successfully added items.

The following code snippet shows how to use the `insert` and `bulkInsert` methods:

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Assign values for each row.
newValues.put(COLUMN_NAME, newValue);
[ ... Repeat for each column ... ]

Uri myRowUri = getContentResolver().insert(MyProvider.CONTENT_URI,
                                           newValues);

// Create a new row of values to insert.
ContentValues[] valueArray = new ContentValues[5];

// TODO: Create an array of new rows

int count = getContentResolver().bulkInsert(MyProvider.CONTENT_URI,
                                            valueArray);
```

Deletes

To delete a single record using the Content Resolver, call `delete`, passing in the URI of the row you want to remove. Alternatively, you can specify a `where` clause to remove multiple rows. Both techniques are shown in the following snippet:

```
// Remove a specific row.
getContentResolver().delete(myRowUri, null, null);

// Remove the first five rows.
String where = "_id < 5";
getContentResolver().delete(MyProvider.CONTENT_URI, where, null);
```

Updates

Updates to a Content Provider are handled using the `update` method on a Content Resolver. The `update` method takes the URI of the target Content Provider, a `ContentValues` object that maps column names to updated values, and a `where` clause that specifies which rows to update.

When executed, every matching row in the `where` clause will be updated using the values in the Content Values passed in and will return the number of successful updates.

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Create a replacement map, specifying which columns you want to
// update, and what values to assign to each of them.
newValues.put(COLUMN_NAME, newValue);

// Apply to the first 5 rows.
String where = "_id < 5";

getContentResolver().update(MyProvider.CONTENT_URI, newValues, where,
                            null);
```

Accessing Files in Content Providers

Content Providers represent files as fully qualified URIs rather than raw file data. To insert a file into a Content Provider, or access a saved file, use the Content Resolvers `openOutputStream` or `openInputStream` methods, respectively. The process for storing a file is shown in the following code snippet:

```
// Insert a new row into your provider, returning its unique URI.
Uri uri = getContentResolver().insert(MyProvider.CONTENT_URI,
                                      newValues);

try {
  // Open an output stream using the new row's URI.
  OutputStream outStream = getContentResolver().openOutputStream(uri);
  // Compress your bitmap and save it into your provider.
  sourceBitmap.compress(Bitmap.CompressFormat.JPEG, 50, outStream);
}
catch (FileNotFoundException e) { }
```

Native Android Content Providers

Android exposes many Content Providers that supply access to the native databases.

You can use each of these Content Providers natively using the techniques described previously. Alternatively, the `android.provider` class includes convenience classes that simplify access to many of the most useful providers, including:

- **Browser** Use the browser Content Provider to read or modify bookmarks, browser history, or web searches.
- **CallLog** View or update the call history including both incoming and outgoing calls together with missed calls and call details, like caller ID and call durations.
- **Contacts** Use the Contacts provider to retrieve, modify, or store your contacts' details.

- **MediaStore** The Media Store provides centralized, managed access to the multimedia on your device, including audio, video, and images. You can store your own multimedia within the Media Store and make it globally available.
- **Settings** You can access the device's preferences using the Settings provider. Using it, you can view and modify Bluetooth settings, ring tones, and other device preferences.

You should use these native Content Providers wherever possible to ensure that your application integrates seamlessly with other native and third-party applications.

While a detailed description of how to use each of these helpers is beyond the scope of this chapter, the following sections describe how to use some of the more useful and powerful native Content Providers.

Using the Media Store Provider

The Android Media Store provides a managed repository for audio, video, and image files. Whenever you add a new multimedia file to the Android filesystem, it should be added to the Media Store to expose it to other applications.

The `MediaStore` class includes a number of convenience methods to simplify inserting files into the Media Store. For example, the following code snippet shows how to insert an image directly into the Media Store:

```
android.provider.MediaStore.Images.Media.insertImage(
  getContentResolver(),
  sourceBitmap,
  "my_cat_pic",
  "Photo of my cat!");
```

Using the Contacts Provider

Access to the Contact Manager is particularly powerful on a communications device. Android does the right thing by exposing all the information available from the contacts database to any application granted the `READ_CONTACTS` permission.

In the following example, an Activity gets a `Cursor` to every person in the contact database, creating an array of `Strings` that holds each contact's name and phone number.

To simplify extracting the data from the `Cursor`, Android supplies public static properties on the `People` class that expose the column names.

```
// Get a cursor over every contact.
Cursor cursor = getContentResolver().query(People.CONTENT_URI,
                                           null, null, null, null);

// Let the activity manage the cursor lifecycle.
startManagingCursor(cursor);

// Use the convenience properties to get the index of the columns
int nameIdx = cursor.getColumnIndexOrThrow(People.NAME);
```

```
int phoneIdx = cursor. getColumnIndexOrThrow(People.NUMBER);

String[] result = new String[cursor.getCount()];
if (cursor.moveToFirst())
  do {
    // Extract the name.
    String name = cursor.getString(nameIdx);
    // Extract the phone number.
    String phone = cursor.getString(phoneIdx);
    result[cursor.getPosition()] = name + " (" + phone + ")";
  } while(cursor.moveToNext());
```

To run this code snippet, you need to add the `READ_CONTACTS` *permission to your application.*

As well as querying the contacts database, you can use this Content Provider to modify, delete, or insert contact records.

Creating a New Content Provider

Create a new Content Provider by extending the abstract `ContentProvider` class. Override the `onCreate` method to open or initialize the underlying data source you're exposing with this new provider. The skeleton code for a new Content Provider is shown below:

```
import android.content.*;
import android.database.Cursor;
import android.net.Uri;
import android.database.SQLException;

public class MyProvider extends ContentProvider {

  @Override
  public boolean onCreate() {
    // TODO: Construct the underlying database.
    return true;
  }
}
```

You should also expose a public static `CONTENT_URI` variable that returns the full URI to this provider. Content URIs must be unique between providers, so it's good practice to base the URI path on your package name. The general form for defining a Content Provider's URI is

```
content://com.<CompanyName>.provider.<ApplicationName>/<DataPath>
```

For example:

```
content://com.paad.provider.myapp/items
```

Content URIs can represent either of two forms. The previous URI represents a request for all values of that type (e.g., all items).

Appending a trailing /<rownumber>, as shown below, represents a request for a single record (e.g., "the fifth item").

```
content://com.paad.provider.myapp/items/5
```

It's good form to support access to your provider using both these forms.

The simplest way to do this is using a `UriMatcher`. Configure the `UriMatcher` to parse URIs to determine their form when the provider is being accessed through a Content Resolver. The following snippet shows the skeleton code for this pattern:

```
public class MyProvider extends ContentProvider {

  private static final String myURI =
    "content://com.paad.provider.myapp/items";
  public static final Uri CONTENT_URI = Uri.parse(myURI);

  @Override
  public boolean onCreate() {
    // TODO: Construct the underlying database.
    return true;
  }

  // Create the constants used to differentiate between the different
  // URI requests.
  private static final int ALLROWS = 1;
  private static final int SINGLE_ROW = 2;

  private static final UriMatcher uriMatcher;

  // Populate the UriMatcher object, where a URI ending in 'items' will
  // correspond to a request for all items, and 'items/[rowID]'
  // represents a single row.
  static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI("com.paad.provider.myApp", "items", ALLROWS);
    uriMatcher.addURI("com.paad.provider.myApp", "items/#",
                      SINGLE_ROW);
  }
}
```

You can use the same technique to expose alternative URIs for different subsets of data, or different tables within your database from within the same Content Provider.

It's also good practice to expose the names and indexes of the columns available in your provider to simplify extracting information from `Cursor` results.

Exposing Access to the Data Source

You can expose queries and transactions with your Content Provider by implementing the `delete`, `insert`, `update`, and `query` methods.

These methods act as a generic interface to the underlying data source, allowing Android applications to share data across application boundaries without having to publish separate interfaces for each application. The most common scenario is to use a Content Provider to expose a private SQLite Database, but within these methods you can access any source of data (including files or application instance variables).

The following code snippet shows the skeleton code for implementing queries and transactions for a Content Provider. Notice that the `UriMatcher` object is used to refine the transaction and query requests.

```
@Override
public Cursor query(Uri uri,
                    String[] projection,
                    String selection,
                    String[] selectionArgs,
                    String sort) {

  // If this is a row query, limit the result set to the passed in row.
  switch (uriMatcher.match(uri)) {
    case SINGLE_ROW :
      // TODO: Modify selection based on row id, where:
      // rowNumber = uri.getPathSegments().get(1));
  }
  return null;
}

@Override
public Uri insert(Uri _uri, ContentValues _initialValues) {
  long rowID = [ ... Add a new item ... ]

  // Return a URI to the newly added item.
  if (rowID > 0) {
    return ContentUris.withAppendedId(CONTENT_URI, rowID);
  }
  throw new SQLException("Failed to add new item into " + _uri);
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
  switch (uriMatcher.match(uri)) {
    case ALLROWS:
    case SINGLE_ROW:
    default: throw new IllegalArgumentException("Unsupported URI:" +
                                                uri);

  }
}

@Override
public int update(Uri uri, ContentValues values, String where,
                  String[] whereArgs) {
  switch (uriMatcher.match(uri)) {
    case ALLROWS:
    case SINGLE_ROW:
```

```
      default: throw new IllegalArgumentException("Unsupported URI:" +
                                                  uri);
    }
  }
```

The final step in creating a Content Provider is defining the MIME type that identifies the data the provider returns.

Override the `getType` method to return a String that uniquely describes your data type. The type returned should include two forms, one for a single entry and another for all the entries, following the forms:

- **Single Item**

  ```
  vnd.<companyname>.cursor.item/<contenttype>
  ```

- **All Items**

  ```
  vnd.<companyName>.cursor.dir/<contenttype>
  ```

The following code snippet shows how to override the `getType` method to return the correct MIME type based on the URI passed in:

```
@Override
public String getType(Uri _uri) {
  switch (uriMatcher.match(_uri)) {
    case ALLROWS: return "vnd.paad.cursor.dir/myprovidercontent";
    case SINGLE_ROW: return "vnd.paad.cursor.item/myprovidercontent";
    default: throw new IllegalArgumentException("Unsupported URI: " +
                                                _uri);
  }
}
```

Registering Your Provider

Once you have completed your Content Provider, it must be added to the application manifest.

Use the `authorities` tag to specify its address, as shown in the following XML snippet:

```
<provider android:name="MyProvider"
          android:authorities="com.paad.provider.myapp"/>
```

Creating and Using an Earthquake Content Provider

Having created an application that features a list of earthquakes, you have an excellent opportunity to share this information with other applications.

By exposing these data through a Content Provider, you, and others, can create new applications based on earthquake data without having to duplicate network traffic and the associated XML parsing.

Creating the Content Provider

The following example shows how to create an Earthquake Content Provider. Each quake will be stored in an SQLite database.

1. Open the Earthquake project, and create a new `EarthquakeProvider` class that extends `ContentProvider`. Include stubs to override the `onCreate`, `getType`, `query`, `insert`, `delete`, and `update` methods.

```
package com.paad.earthquake;

import android.content.*;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;
import android.util.Log;

public class EarthquakeProvider extends ContentProvider {

  @Override
  public boolean onCreate() {
  }

  @Override
  public String getType(Uri url) {
  }

  @Override
  public Cursor query(Uri url, String[] projection, String selection,
                      String[] selectionArgs, String sort) {
  }

  @Override
  public Uri insert(Uri _url, ContentValues _initialValues) {
  }

  @Override
  public int delete(Uri url, String where, String[] whereArgs) {
  }

  @Override
  public int update(Uri url, ContentValues values,
                    String where, String[] wArgs) {
  }
}
```

2. Expose a content URI for this provider. This URI will be used to access the Content Provider from within application components using a `ContentResolver`.

```
public static final Uri CONTENT_URI =
  Uri.parse("content://com.paad.provider.earthquake/earthquakes");
```

3. Create the database that will be used to store the earthquakes. Within the `EathquakeProvider` class, create a new `SQLiteDatabase` instance, and expose public variables that describe the column names and indexes. Include an extension of `SQLiteOpenHelper` to manage database creation and version control.

```
// The underlying database
private SQLiteDatabase earthquakeDB;

private static final String TAG = "EarthquakeProvider";
private static final String DATABASE_NAME = "earthquakes.db";
private static final int DATABASE_VERSION = 1;
private static final String EARTHQUAKE_TABLE = "earthquakes";

// Column Names
public static final String KEY_ID = "_id";
public static final String KEY_DATE = "date";
public static final String KEY_DETAILS = "details";
public static final String KEY_LOCATION_LAT = "latitude";
public static final String KEY_LOCATION_LNG = "longitude";
public static final String KEY_MAGNITUDE = "magnitude";
public static final String KEY_LINK = "link";

// Column indexes
public static final int DATE_COLUMN = 1;
public static final int DETAILS_COLUMN = 2;
public static final int LONGITUDE_COLUMN = 3;
public static final int LATITUDE_COLUMN = 4;
public static final int MAGNITUDE_COLUMN = 5;
public static final int LINK_COLUMN = 6;

// Helper class for opening, creating, and managing
// database version control
private static class earthquakeDatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_CREATE =
    "create table " + EARTHQUAKE_TABLE + " ("
    + KEY_ID + " integer primary key autoincrement, "
    + KEY_DATE + " INTEGER, "
    + KEY_DETAILS + " TEXT, "
    + KEY_LOCATION_LAT + " FLOAT, "
    + KEY_LOCATION_LNG + " FLOAT, "
    + KEY_MAGNITUDE + " FLOAT, "
    + KEY_LINK + " TEXT);";

  public earthquakeDatabaseHelper(Context context, String name,
                                  CursorFactory factory, int version) {
    super(context, name, factory, version);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(DATABASE_CREATE);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
```

```
                        int newVersion) {
      Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
                 + newVersion + ", which will destroy all old data");

      db.execSQL("DROP TABLE IF EXISTS " + EARTHQUAKE_TABLE);
      onCreate(db);
    }
  }
```

4. Create a `UriMatcher` to handle requests using different URIs. Include support for queries and transactions over the entire data set (`QUAKES`) and a single record matching a quake index value (`QUAKE_ID`).

```
// Create the constants used to differentiate between the different URI
// requests.
private static final int QUAKES = 1;
private static final int QUAKE_ID = 2;

private static final UriMatcher uriMatcher;

// Allocate the UriMatcher object, where a URI ending in 'earthquakes'
// will correspond to a request for all earthquakes, and 'earthquakes'
// with a trailing '/[rowID]' will represent a single earthquake row.
static {
  uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
  uriMatcher.addURI("com.paad.provider.Earthquake", "earthquakes",
                    QUAKES);
  uriMatcher.addURI("com.paad.provider.Earthquake", "earthquakes/#",
                    QUAKE_ID);
}
```

5. Override the `getType` method to return a `String` for each of the URI structures supported.

```
@Override
public String getType(Uri uri) {
  switch (uriMatcher.match(uri)) {
    case QUAKES:
      return "vnd.android.cursor.dir/vnd.paad.earthquake";
    case QUAKE_ID:
      return "vnd.android.cursor.item/vnd.paad.earthquake";
    default:
      throw new IllegalArgumentException("Unsupported URI: " + uri);
  }
}
```

6. Override the provider's `onCreate` handler to create a new instance of the database helper class and open a connection to the database.

```
@Override
public boolean onCreate() {
  Context context = getContext();

  earthquakeDatabaseHelper dbHelper;
  dbHelper = new earthquakeDatabaseHelper(context, DATABASE_NAME, null,
                                          DATABASE_VERSION);
```

```
    earthquakeDB = dbHelper.getWritableDatabase();
    return (earthquakeDB == null) ? false : true;
  }
```

7. Implement the query and transaction stubs. Start with the `query` method; it should decode the request being made (all content or a single row) and apply the selection, projection, and sort-order criteria parameters to the database before returning a result `Cursor`.

```
@Override
public Cursor query(Uri uri,
                    String[] projection,
                    String selection,
                    String[] selectionArgs,
                    String sort) {

  SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

  qb.setTables(EARTHQUAKE_TABLE);

  // If this is a row query, limit the result set to the passed in row.
  switch (uriMatcher.match(uri)) {
    case QUAKE_ID:
      qb.appendWhere(KEY_ID + "=" + uri.getPathSegments().get(1));
      break;
    default: break;
  }

  // If no sort order is specified sort by date / time
  String orderBy;
  if (TextUtils.isEmpty(sort)) {
    orderBy = KEY_DATE;
  } else {
    orderBy = sort;
  }

  // Apply the query to the underlying database.
  Cursor c = qb.query(earthquakeDB,
                      projection,
                      selection, selectionArgs,
                      null, null,
                      orderBy);

  // Register the contexts ContentResolver to be notified if
  // the cursor result set changes.
  c.setNotificationUri(getContext().getContentResolver(), uri);

  // Return a cursor to the query result.
  return c;
}
```

8. Now implement methods for inserting, deleting, and updating content. In this case, the process is largely an exercise in mapping Content Provider transaction requests to database equivalents.

```
@Override
public Uri insert(Uri _uri, ContentValues _initialValues) {
```

```
    // Insert the new row, will return the row number if
    // successful.
    long rowID = earthquakeDB.insert(EARTHQUAKE_TABLE, "quake",
                                     _initialValues);

    // Return a URI to the newly inserted row on success.
    if (rowID > 0) {
      Uri uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
      getContext().getContentResolver().notifyChange(uri, null);
      return uri;
    }
    throw new SQLException("Failed to insert row into " + _uri);
  }

  @Override
  public int delete(Uri uri, String where, String[] whereArgs) {
    int count;

    switch (uriMatcher.match(uri)) {
      case QUAKES:
        count = earthquakeDB.delete(EARTHQUAKE_TABLE, where, whereArgs);
        break;

      case QUAKE_ID:
        String segment = uri.getPathSegments().get(1);
        count = earthquakeDB.delete(EARTHQUAKE_TABLE, KEY_ID + "="
                                    + segment
                                    + (!TextUtils.isEmpty(where) ? " AND ("
                                    + where + ')' : ""), whereArgs);
        break;

      default: throw new IllegalArgumentException("Unsupported URI: " +
                                                  uri);
    }

    getContext().getContentResolver().notifyChange(uri, null);
    return count;
  }

  @Override
  public int update(Uri uri, ContentValues values, String where,
                    String[] whereArgs) {
    int count;
    switch (uriMatcher.match(uri)) {
      case QUAKES:
        count = earthquakeDB.update(EARTHQUAKE_TABLE, values,
                                    where, whereArgs);
        break;

      case QUAKE_ID:
        String segment = uri.getPathSegments().get(1);
        count = earthquakeDB.update(EARTHQUAKE_TABLE, values, KEY_ID
                                    + "=" + segment
                                    + (!TextUtils.isEmpty(where) ? " AND ("
```

```
                                    + where + ')' : ""), whereArgs);
        break;

      default: throw new IllegalArgumentException("Unknown URI " + uri);
    }

    getContext().getContentResolver().notifyChange(uri, null);
    return count;
  }
```

9. With the Content Provider finished, register it in the manifest by creating a new node within the application tag.

```
<provider android:name=".EarthquakeProvider"
          android:authorities="com.paad.provider.earthquake" />
```

Using the Provider

You can now update the Earthquake Activity to use the Earthquake Provider to store quakes, and use them to populate the List View.

1. Within the Earthquake Activity, update the `addNewQuake` method. It should use the application's Content Resolver to insert each new earthquake into the provider. Move the existing array control logic into a separate `addQuakeToArray` method.

```
private void addNewQuake(Quake _quake) {
  ContentResolver cr = getContentResolver();
  // Construct a where clause to make sure we don't already have this
  // earthquake in the provider.
  String w = EarthquakeProvider.KEY_DATE + " =" +
             _quake.getDate().getTime();

  // If the earthquake is new, insert it into the provider.
  Cursor c = cr.query(EarthquakeProvider.CONTENT_URI,
                      null, w, null, null);
  int dbCount = c.getCount();
  c.close();

  if (dbCount > 0) {
    ContentValues values = new ContentValues();

    values.put(EarthquakeProvider.KEY_DATE,
               _quake.getDate().getTime());
    values.put(EarthquakeProvider.KEY_DETAILS, _quake.getDetails());

    double lat = _quake.getLocation().getLatitude();
    double lng = _quake.getLocation().getLongitude();
    values.put(EarthquakeProvider.KEY_LOCATION_LAT, lat);
    values.put(EarthquakeProvider.KEY_LOCATION_LNG, lng);
    values.put(EarthquakeProvider.KEY_LINK, _quake.getLink());
    values.put(EarthquakeProvider.KEY_MAGNITUDE,
               _quake.getMagnitude());

    cr.insert(EarthquakeProvider.CONTENT_URI, values);
```

```
        earthquakes.add(_quake);

        addQuakeToArray(_quake);
      }
    }

    private void addQuakeToArray(Quake _quake) {
      if (_quake.getMagnitude() > minimumMagnitude) {
        // Add the new quake to our list of earthquakes.
        earthquakes.add(_quake);

        // Notify the array adapter of a change.
        aa.notifyDataSetChanged();
      }
    }
```

2. Create a new `loadQuakesFromProvider` method that loads all the earthquakes from the Earthquake Provider and inserts them into the array list using the `addQuakeToArray` method created in Step 1.

```
private void loadQuakesFromProvider() {
// Clear the existing earthquake array
earthquakes.clear();

ContentResolver cr = getContentResolver();

  // Return all the saved earthquakes
  Cursor c = cr.query(EarthquakeProvider.CONTENT_URI,
                      null, null, null, null);

  if (c.moveToFirst())
    {
      do {
        // Extract the quake details.
        Long datems = c.getLong(EarthquakeProvider.DATE_COLUMN);
        String details;
        details = c.getString(EarthquakeProvider.DETAILS_COLUMN);
        Float lat = c.getFloat(EarthquakeProvider.LATITUDE_COLUMN);
        Float lng = c.getFloat(EarthquakeProvider.LONGITUDE_COLUMN);
        Double mag = c.getDouble(EarthquakeProvider.MAGNITUDE_COLUMN);
        String link = c.getString(EarthquakeProvider.LINK_COLUMN);

        Location location = new Location("dummy");
        location.setLongitude(lng);
        location.setLatitude(lat);

        Date date = new Date(datems);

        Quake q = new Quake(date, details, location, mag, link);
        addQuakeToArray(q);
      } while(c.moveToNext());
  }
  c.close();
}
```

3. Call `loadQuakesFromProvider` from `onCreate` to initialize the Earthquake List View at start-up.

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

  earthquakeListView =
    (ListView)this.findViewById(R.id.earthquakeListView);

  earthquakeListView.setOnItemClickListener(new OnItemClickListener() {

    public void onItemClick(AdapterView _av, View _v,
                            int _index, long arg3) {
      selectedQuake = earthquakes.get(_index);
      showDialog(QUAKE_DIALOG);
    }
  });

  int layoutID = android.R.layout.simple_list_item_1;
  aa = new ArrayAdapter<Quake>(this, layoutID , earthquakes);
  earthquakeListView.setAdapter(aa);

  loadQuakesFromProvider();

  updateFromPreferences();
  refreshEarthquakes();
}
```

4. Finally, make a change to the `refreshEarthquakes` method so that it loads the saved earthquakes from the provider after clearing the array, but before adding any new quakes received.

```
private void refreshEarthquakes() {
  [ ... exiting refreshEarthquakes method ... ]

  // Clear the old earthquakes
  earthquakes.clear();
  loadQuakesFromProvider();

  [ ... exiting refreshEarthquakes method ... ]
}
```

Summary

In this chapter, you learned how to add a persistence layer to your applications.

Starting with the ability to save the Activity instance data between sessions using the save and restore instance state handlers, you were then introduced to Shared Preferences. You used them to save instance values and user preferences that can be used across your application components.

Android provides a fully featured SQLite RDBMS to all applications. This small, efficient, and robust database lets you create relational databases to persist application data. Using Content Providers, you learned how to share private data, particularly databases, across application boundaries.

All database and Content Provider queries are returned as Cursors; you learned how to perform queries and extract data from the resulting Cursor objects.

Along the way, you also learned to:

- Save and load files directly to and from the underlying filesystem.
- Include static files as external project resources.
- Create new SQLite databases.
- Interact with databases to insert, update, and delete rows.
- Use the native Content Providers included with Android to access and manage native data like media and contacts.

With a solid foundation in the fundamentals of Android development, the remainder of this book will investigate some of the more interesting optional Android features.

Starting in the next chapter, you'll be introduced to the geographic APIs. Android offers a rich suite of geographical functionality including location-based services (such as GPS), forward and reverse geocoding, as well as a fully integrated Google Maps implementation. Using Google Maps, you can create map-based Activities that feature annotations to develop native map-mashup style applications.

7

Maps, Geocoding, and Location-Based Services

One of the defining features of mobile phones is their portability, so it's not surprising that some of the most enticing Android features are the services that let you find, contextualize, and map physical locations.

You can create map-based Activities using Google Maps as a User Interface element. You have full access to the map, allowing you to control display settings, alter the zoom level, and move the centered location. Using Overlays, you can annotate maps and handle user input to provide map-contextualized information and functionality.

Also covered in this chapter are the location-based services (LBS) — the services that let you find the device's current location. They include technologies like GPS and Google's cell-based location technology. You can specify which location-sensing technology to use explicitly by name, or implicitly by defining a set of criteria in terms of accuracy, cost, and other requirements.

Maps and location-based services use latitude and longitude to pinpoint geographic locations, but your users are more likely to think in terms of an address. Android provides a Geocoder that supports forward and reverse geocoding. Using the Geocoder, you can convert back and forth between latitude/longitude values and real-world addresses.

Used together, the mapping, geocoding, and location-based services provide a powerful toolkit for incorporating your phone's native mobility into your mobile applications.

In this chapter, you'll learn to:

- ❑ Set up your emulator to test location-based services.
- ❑ Find and track the device location.
- ❑ Create proximity alerts.

- Turn geographical locations into street addresses and vice versa.
- Create and customize map-based Activities using MapView and MapActivity.
- Add Overlays to your maps.

Using Location-Based Services

Location-based services (LBS) is an umbrella term used to describe the different technologies used to find the device's current location. The two main LBS elements are:

- **LocationManager** Provides hooks to the location-based services.
- **LocationProviders** Each of which represents a different location-finding technology used to determine the device's current location.

Using the Location Manager, you can:

- Obtain your current location.
- Track movement.
- Set proximity alerts for detecting movement into and out of a specified area.

Setting up the Emulator with Test Providers

Location-based services are dependant on device hardware for finding the current location. When developing and testing with the emulator, your hardware is virtualized, and you're likely to stay in pretty much the same location.

To compensate, Android includes hooks that let you emulate Location Providers for testing location-based applications. In this section, you'll learn how to mock the position of the supported GPS provider.

> **If you're planning on doing location-based application development and using the Android emulator, this section will show how to create an environment that simulates real hardware and location changes. In the remainder of this chapter, it will be assumed that you have used the examples in this section to update the location for, the `GPS_PROVIDER` within the emulator.**

Updating Locations in Emulator Location Providers

Use the Location Controls available from the DDMS perspective in Eclipse (shown in Figure 7-1) to push location changes directly into the test `GPS_PROVIDER`.

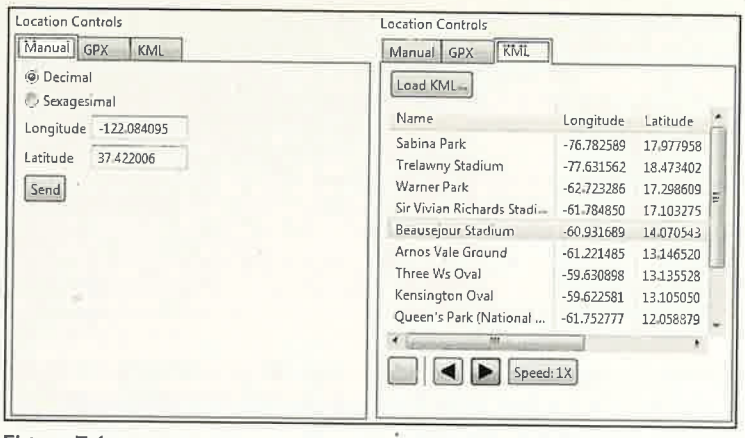


Figure 7-1

Figure 7-1 shows the Manual and KML tabs. Using the Manual tab, you can specify particular latitude/longitude pairs. Alternatively, the KML and GPX tabs let you load KML (Keyhole Markup Language) and GPX (GPS Exchange Format) files, respectively. Once loaded, you can jump to particular waypoints (locations) or play back each location sequentially.

> *Most GPS systems record track files using GPX, while KML is used extensively online to define geographic information. You can handwrite your own KML file or generate one automatically using Google Earth and finding directions between two locations.*

All location changes applied using the DDMS Location Controls will be applied to the GPS receiver, which must be enabled and active. Note that the GPS values returned by `getLastKnownLocation` will not change unless at least one application has requested location updates.

Create an Application to Manage Test Location Providers

In this example, you'll create a new project to set up the emulator to simplify testing other location-based applications. Running this project will ensure that the GPS provider is active and updating regularly.

1. Create a new Android project, TestProviderController, which includes a `TestProviderController` Activity.

```
package com.paad.testprovidercontroller;

import java.util.List;
import android.app.Activity;
import android.content.Context;
import android.location.Criteria;
import android.location.Location;
import android.location.LocationManager;
import android.location.LocationListener;
import android.location.LocationProvider;
```

```
import android.os.Bundle;
import android.widget.TextView;

public class TestProviderController extends Activity {

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

}
```

2. Add an instance variable to store a reference to the `LocationManager`, then get that reference to it from within the `onCreate` method. Add stubs for creating a new test provider and to enable the GPS provider for testing.

```
LocationManager locationManager;

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

  String location_context = Context.LOCATION_SERVICE;
  locationManager = (LocationManager)getSystemService(location_context);
  testProviders();
}

public void testProviders() {}
```

3. Add a `FINE_LOCATION` permission to test the providers.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

4. Update the `testProviders` method to check the enabled status of each provider and return the last known location; also request periodic updates for each provider to force Android to start updating the locations for other applications. The methods used here are presented without comment; you'll learn more about how to use each of them in the remainder of this chapter.

```
public void testProviders() {
  TextView tv = (TextView)findViewById(R.id.myTextView);
  StringBuilder sb = new StringBuilder("Enabled Providers:");

  List<String> providers = locationManager.getProviders(true);

  for (String provider : providers) {

    locationManager.requestLocationUpdates(provider, 1000, 0,
                                           new LocationListener() {
      public void onLocationChanged(Location location) {}
      public void onProviderDisabled(String provider){}
```

```
        public void onProviderEnabled(String provider){}
        public void onStatusChanged(String provider, int status,
                                    Bundle extras){}
      });

      sb.append("\n").append(provider).append(": ");

      Location location = locationManager.getLastKnownLocation(provider);
      if (location != null) {
        double lat = location.getLatitude();
        double lng = location.getLongitude();
        sb.append(lat).append(", ").append(lng);
      } else {
        sb.append("No Location");
      }
    }
    tv.setText(sb);
  }
```

5. The final step before you run the application is to update the main.xml layout resource to add an ID for the text label you're updating in Step 4.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:id="@+id/myTextView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
  />
</LinearLayout>
```

6. Run your application, and it should appear as shown in Figure 7-2.



Figure 7-2

7. Android will now update the last known position for any applications using location-based services. You can update the current location using the techniques described in the previous section.

The test provider controller application you just wrote needs to be restarted to reflect any changes in the current location. Below in this chapter, you'll learn how to request updates based on the elapsed time and distance traveled.

Selecting a Location Provider

Depending on the device, there may be several technologies that Android can use to determine the current location. Each technology, or Location Provider, will offer different capabilities including power consumption, monetary cost, accuracy, and the ability to determine altitude, speed, or heading information.

To get an instance of a specific provider, call `getProvider`, passing in the name:

```
String providerName = LocationManager.GPS_PROVIDER;
LocationProvider gpsProvider;
gpsProvider = locationManager.getProvider(providerName);
```

This is generally only useful for determining the abilities of a particular provider. Most Location Manager methods require only a provider name to perform location-based services.

Finding the Available Providers

The `LocationManager` class includes static string constants that return the provider name for the two most common Location Providers:

- `LocationManager.GPS_PROVIDER`
- `LocationManager.NETWORK_PROVIDER`

To get a list of names for all the providers available on the device, call `getProviders`, using a Boolean to indicate if you want all, or only the enabled, providers to be returned:

```
boolean enabledOnly = true;
List<String> providers = locationManager.getProviders(enabledOnly);
```

Finding Providers Based on Requirement Criteria

In most scenarios, it's unlikely that you will want to explicitly choose the Location Provider to use. More commonly, you'll specify the requirements that a provider must meet and let Android determine the best technology to use.

Use the `Criteria` class to dictate the requirements of a provider in terms of accuracy (fine or coarse), power use (low, medium, high), cost, and the ability to return values for altitude, speed, and bearing.

The following code creates Criteria that require coarse accuracy, low power consumption, and no need for altitude, bearing, or speed. The provider is permitted to have an associated cost.

```
Criteria criteria = new Criteria();
criteria.setAccuracy(Criteria.ACCURACY_COARSE);
criteria.setPowerRequirement(Criteria.POWER_LOW);
criteria.setAltitudeRequired(false);
criteria.setBearingRequired(false);
criteria.setSpeedRequired(false);
criteria.setCostAllowed(true);
```

Having defined the required Criteria, you can use `getBestProvider` to return the best matching Location Provider or `getProviders` to return all the possible matches. The following snippet demonstrates using `getBestProvider` to return the best provider for your criteria where the Boolean lets you restrict the result to a currently enabled provider:

```
String bestProvider = locationManager.getBestProvider(criteria, true);
```

If more than one Location Provider matches your criteria, the one with the greatest accuracy is returned. If no Location Providers meet your requirements, the criteria are loosened, in the following order, until a provider is found:

- Power use
- Accuracy
- Ability to return bearing, speed, and altitude

The criterion for allowing a device with monetary cost is never implicitly relaxed. If no provider is found, null is returned.

To see a list of names for all the providers that match your criteria, you can use `getProviders`. It accepts Criteria and returns a filtered String list of all available Location Providers that match them. As with the `getBestProvider` call, if no matching providers are found, this call returns null.

```
List<String> matchingProviders = locationManager.getProviders(criteria,
                                                               false);
```

Finding Your Location

The purpose of location-based services is to find the physical location of the device.

Access to the location-based services is handled using the Location Manager system Service. To access the Location Manager, request an instance of the `LOCATION_SERVICE` using the `getSystemService` method, as shown in the following snippet:

```
String serviceString = Context.LOCATION_SERVICE;
LocationManager locationManager;
locationManager = (LocationManager)getSystemService(serviceString);
```

Before you can use the Location Manager, you need to add one or more `uses-permission` tags to your manifest to support access to the LBS hardware.

The following snippet shows the *fine* and *coarse* permissions. Of the default providers, the GPS provider requires fine permission, while the Network provider requires only coarse. An application that has been granted fine permission will have coarse permission granted implicitly.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

You can find the last location fix determined by a particular Location Provider using the `getLastKnownLocation` method, passing in the name of the Location Provider. The following example finds the last location fix taken by the GPS provider:

```
String provider = LocationManager.GPS_PROVIDER;
Location location = locationManager.getLastKnownLocation(provider);
```

Note that `getLastKnownLocation` does not ask the Location Provider to update the current position. If the device has not recently updated the current position this value may be be out of date.

The Location object returned includes all the position information available from the provider that supplied it. This can include latitude, longitude, bearing, altitude, speed, and the time the location fix was taken. All these properties are available using `get` methods on the Location object. In some instances, additional details will be included in the extras Bundle.

"Where Am I?" Example

The following example — "Where Am I?" — features a new Activity that finds the device's current location using the GPS Location Provider. You will expand on this example throughout the chapter as you learn new geographic functionality.

This example assumes that you have enabled the `GPS_PROVIDER` Location Provider using the techniques shown previously in this chapter, or that you're running it on a device that supports GPS and has that hardware enabled.

1. Create a new WhereAmI project with a `WhereAmI` Activity. This example uses the GPS provider (either mock or real), so modify the manifest file to include the `uses-permission` tag for `ACCESS_FINE_LOCATION`.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.paad.whereami">
  <application
    android:icon="@drawable/icon">
    <activity
      android:name=".WhereAmI"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

</manifest>
```

2. Modify the main.xml layout resource to include an `android:ID` attribute for the `TextView` control so that you can access it from within the Activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
      android:id="@+id/myLocationText"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/hello"
    />
</LinearLayout>
```

3. Override the `onCreate` method of the `WhereAmI` Activity to get a reference to the Location Manager. Call `getLastKnownLocation` to get the last location fix value, and pass it in to the `updateWithNewLocation` method stub.

```
package com.paad.whereami;

import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.TextView;

public class WhereAmI extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    LocationManager locationManager;
    String context = Context.LOCATION_SERVICE;
    locationManager = (LocationManager)getSystemService(context);

    String provider = LocationManager.GPS_PROVIDER;
    Location location = locationManager.getLastKnownLocation(provider);

    updateWithNewLocation(location);
  }

  private void updateWithNewLocation(Location location) {
  }
}
```

4. Fill in the `updateWithNewLocation` method to display the passed-in Location in the Text View by extracting the latitude and longitude values.

```
private void updateWithNewLocation(Location location) {
  String latLongString;
  TextView myLocationText;
  myLocationText = (TextView)findViewById(R.id.myLocationText);
  if (location != null) {
    double lat = location.getLatitude();
    double lng = location.getLongitude();
```

```
            latLongString = "Lat:" + lat + "\nLong:" + lng;
          } else {
            latLongString = "No location found";
          }
          myLocationText.setText("Your Current Position is:\n" +
                                 latLongString);
        }
```

5. When running, your Activity should look like Figure 7-3.



Figure 7-3

Tracking Movement

Most location-sensitive applications will need to be reactive to user movement. Simply polling the Location Manager will not force it to get new updates from the Location Providers.

Use the `requestLocationUpdates` method to get updates whenever the current location changes, using a `LocationListener`. Location Listeners also contain hooks for changes in a provider's status and availability.

The `requestLocationUpdates` method accepts either a specific Location Provider name or a set of Criteria to determine the provider to use.

To optimize efficiency and minimize cost and power use, you can also specify the minimum time and the minimum distance between location change updates.

The following snippet shows the skeleton code for requesting regular updates based on a minimum time and distance.

```
String provider = LocationManager.GPS_PROVIDER;

int t = 5000; // milliseconds
int distance = 5; // meters

LocationListener myLocationListener = new LocationListener() {

  public void onLocationChanged(Location location) {
    // Update application based on new location.
  }

  public void onProviderDisabled(String provider){
    // Update application if provider disabled.
```

```
    }

    public void onProviderEnabled(String provider){
      // Update application if provider enabled.
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras){
      // Update application if provider hardware status changed.
    }
  };

  locationManager.requestLocationUpdates(provider, t, distance,
                                         myLocationListener);
```

When the minimum time and distance values are exceeded, the attached Location Listener will execute its `onLocationChanged` event.

> *You can request multiple location updates pointing to different Location Listeners and using different minimum thresholds. A common design pattern is to create a single listener for your application that broadcasts Intents to notify other components of location changes. This centralizes your listeners and ensures that the Location Provider hardware is used as efficiently as possible.*

To stop location updates, call `removeUpdates`, as shown below. Pass in the Location Listener instance you no longer want to have triggered.

```
locationManager.removeUpdates(myLocationListener);
```

Most GPS hardware incurs significant power cost. To minimize this, you should disable updates whenever possible in your application, specifically when location changes are being used to update an Activity's User Interface. You can improve performance further by extending the minimum time between updates as long as possible.

Updating Your Location in "Where Am I?"

In the following example, "Where Am I?" is enhanced to track your current location by listening for location changes. Updates are restricted to one every 2 seconds, and only when movement of more than 10 meters has been detected.

Rather than explicitly selecting the GPS provider, in this example, you'll create a set of Criteria and let Android choose the best provider available.

1. Start by opening the `WhereAmI` Activity in the WhereAmI project. Update the `onCreate` method to find the best Location Provider that features high accuracy and draws as little power as possible.

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
```

```
    setContentView(R.layout.main);

    LocationManager locationManager;
    String context = Context.LOCATION_SERVICE;
    locationManager = (LocationManager)getSystemService(context);

    Criteria criteria = new Criteria();
    criteria.setAccuracy(Criteria.ACCURACY_FINE);
    criteria.setAltitudeRequired(false);
    criteria.setBearingRequired(false);
    criteria.setCostAllowed(true);
    criteria.setPowerRequirement(Criteria.POWER_LOW);

    String provider = locationManager.getBestProvider(criteria, true);

    Location location = locationManager.getLastKnownLocation(provider);

    updateWithNewLocation(location);}
```

2. Create a new `LocationListener` instance variable that fires the existing `updateWithNewLocation` method whenever a location change is detected.

```
private final LocationListener locationListener = new LocationListener() {
  public void onLocationChanged(Location location) {
    updateWithNewLocation(location);
  }

  public void onProviderDisabled(String provider){
    updateWithNewLocation(null);
  }

  public void onProviderEnabled(String provider){ }
  public void onStatusChanged(String provider, int status,
                              Bundle extras){ }
};
```

3. Return to `onCreate` and execute `requestLocationUpdates`, passing in the new Location Listener object. It should listen for location changes every 2 seconds but fire only when it detects movement of more than 10 meters.

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

  LocationManager locationManager;
  String context = Context.LOCATION_SERVICE;
  locationManager = (LocationManager)getSystemService(context);

  Criteria criteria = new Criteria();
  criteria.setAccuracy(Criteria.ACCURACY_FINE);
  criteria.setAltitudeRequired(false);
  criteria.setBearingRequired(false);
  criteria.setCostAllowed(true);
```

```
        criteria.setPowerRequirement(Criteria.POWER_LOW);

        String provider = locationManager.getBestProvider(criteria, true);

        Location location = locationManager.getLastKnownLocation(provider);

        updateWithNewLocation(location);

        locationManager.requestLocationUpdates(provider, 2000, 10,
                                               locationListener);
    }
```

If you run the application and start changing the device location, you will see the Text View update accordingly.

Using Proximity Alerts

It's often useful to have your applications react when a user moves toward, or away from, a specific location. Proximity alerts let your applications set triggers that are fired when a user moves within or beyond a set distance from a geographic location.

> *Internally, Android may use different Location Providers depending on how close you are to the outside edge of your target area. This allows the power use and cost to be minimized when the alert is unlikely to be fired based on your distance from the interface.*

To set a proximity alert for a given coverage area, select the center point (using longitude and latitude values), a radius around that point, and an expiry time-out for the alert. The alert will fire if the device crosses over that boundary, both when it moves within the radius and when it moves beyond it.

When triggered, proximity alerts fire Intents, most commonly broadcast Intents. To specify the Intent to fire, you use a `PendingIntent`, a class that wraps an Intent in a kind of method pointer, as shown in the following code snippet:

```
Intent intent = new Intent(MY_ACTIVITY);
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, -1, intent, 0);
```

The following example sets a proximity alert that never expires and triggers when the device moves within 10 meters of its target:

```
private static String TREASURE_PROXIMITY_ALERT = "com.paad.treasurealert";

private void setProximityAlert() {
  String locService = Context.LOCATION_SERVICE;
  LocationManager locationManager;
  locationManager = (LocationManager)getSystemService(locService);

  double lat = 73.147536;
  double lng = 0.510638;
  float radius = 100f; // meters
```

```
    long expiration = -1; // do not expire

    Intent intent = new Intent(TREASURE_PROXIMITY_ALERT);
    PendingIntent proximityIntent = PendingIntent.getBroadcast(this, -1,
                                                               intent,
                                                               0);
    locationManager.addProximityAlert(lat, lng, radius, expiration,
                                      proximityIntent);}
```

When the Location Manager detects that you have moved either within or beyond the specified radius, the packaged Intent will be fired with an `extra` keyed as `LocationManager.KEY_PROXIMITY_ENTERING` set to `true` or `false` accordingly.

To handle proximity alerts, you need to create a `BroadcastReceiver`, such as the one shown in the following snippet:

```
public class ProximityIntentReceiver extends BroadcastReceiver {

  @Override
  public void onReceive (Context context, Intent intent) {
    String key = LocationManager.KEY_PROXIMITY_ENTERING;

    Boolean entering = intent.getBooleanExtra(key, false);
    [ ... perform proximity alert actions ... ]
  }

}
```

To start listening for them, register your receiver, as shown in the snippet below:

```
IntentFilter filter = new IntentFilter(TREASURE_PROXIMITY_ALERT);
registerReceiver(new ProximityIntentReceiver(), filter);
```

Using the Geocoder

Geocoding lets you translate between street addresses and longitude/latitude map coordinates. This can give you a recognizable context for the locations and coordinates used in location-based services and map-based Activities.

The `Geocoder` class provides access to two geocoding functions:

- **Forward Geocoding** Finds the latitude and longitude of an address.
- **Reverse Geocoding** Finds the street address for a given latitude and longitude.

The results from these calls will be contextualized using a locale, where a locale is used to define your usual location and language. The following snippet shows how you set the locale when creating your Geocoder. If you don't specify a locale, it will assume your device's default.

```
Geocoder geocoder = new Geocoder(getApplicationContext(),
                                 Locale.getDefault());
```

Both geocoding functions return a list of `Address` objects. Each list can contain several possible results, up to a limit you specify when making the call.

Each Address object is populated with as much detail as the Geocoder was able to resolve. This can include the latitude, longitude, phone number, and increasingly granular address details from country to street and house number.

> *Geocoder lookups are performed synchronously, so they will block the calling thread. For slow data connections, this can lead to an Application Unresponsive dialog. In most cases, it's good form to move these lookups into a Service or background thread, as shown in Chapter 8.*
>
> *For clarity and brevity, the calls made in the code samples within this chapter are made on the main application thread.*

Reverse Geocoding

Reverse geocoding returns street addresses for physical locations, specified by latitude/longitude pairs. It provides a recognizable context for the locations returned by location-based services.

To perform a reverse lookup, you pass the target latitude and longitude to a Geocoder's `getFromLocation` method. It will return a list of possible matching addresses. If the Geocoder could not resolve any addresses for the specified coordinate, it will return null.

The following example shows how to reverse-geocode your last known location:

```
location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);

double latitude = location.getLatitude();
double longitude = location.getLongitude();

Geocoder gc = new Geocoder(this, Locale.getDefault());

List<Address> addresses = null;
try {
  addresses = gc.getFromLocation(latitude, longitude, 10);
} catch (IOException e) {}
```

The accuracy and granularity of reverse lookups are entirely dependent on the quality of data in the geocoding database; as such, the quality of the results may vary widely between different countries and locales.

Forward Geocoding

Forward geocoding (or just geocoding) determines map coordinates for a given location.

> *What constitutes a valid location varies depending on the locale (geographic area) within which you're searching. Generally, it will include regular street addresses of varying granularity (from country to street name and number), postcodes, train stations, landmarks, and hospitals. As a general guide, valid search terms will be similar to the addresses and locations you can enter into the Google Maps search bar.*

To do a forward-geocoding lookup, call `getFromLocationName` on a Geocoder instance. Pass in the location you want the coordinates for and the maximum number of results to return, as shown in the snippet below:

```
List<Address> result = geocoder.getFromLocationName(aStreetAddress, maxResults);
```

The returned list of Addresses can include multiple possible matches for the named location. Each address result will include latitude and longitude and any additional address information available for those coordinates. This is useful to confirm that the correct location was resolved, as well as providing address specifics when searching for landmarks.

As with reverse geocoding, if no matches are found, null will be returned. The availability, accuracy, and granularity of geocoding results will depend entirely on the database available for the area you're searching.

When doing forward lookups, the Locale specified when creating the Geocoder object is particularly important. The Locale provides the geographical context for interpreting your search requests, as the same location names can exist in multiple areas. Where possible, consider selecting a regional Locale to help avoid place name ambiguity.

Additionally, try to use as many address details as possible. For example, the following code snippet demonstrates a forward geocode for a New York street address:

```
Geocoder fwdGeocoder = new Geocoder(this, Locale.US);
String streetAddress = "160 Riverside Drive, New York, New York";

List<Address> locations = null;
try {
  locations = fwdGeocoder.getFromLocationName(streetAddress, 10);
} catch (IOException e) {}
```

For even more specific results, use the `getFromLocationName` overload, that lets you restrict your search to within a geographical bounding box, as shown in the following snippet:

```
List<Address> locations = null;
try {
  locations = fwdGeocoder.getFromLocationName(streetAddress, 10,
                                              n, e, s, w);
} catch (IOException e) {}
```

This overload is particularly useful in conjunction with a MapView as you can restrict the search to within the visible map.

Geocoding "Where Am I?"

Using the Geocoder, you can determine the street address at your current location. In this example, you'll further extend the "Where Am I?" project to include and update the current street address whenever the device moves.

Open the `WhereAmI` Activity. Modify the `updateWithNewLocation` method to instantiate a new Geocoder object, and call the `getFromLocation` method, passing in the newly received location and limiting the results to a single address.

Extract each line in the street address, as well as the locality, postcode, and country, and append this information to an existing Text View string.

```
private void updateWithNewLocation(Location location) {
  String latLongString;
  TextView myLocationText;
  myLocationText = (TextView)findViewById(R.id.myLocationText);

  String addressString = "No address found";

  if (location != null) {
    double lat = location.getLatitude();
    double lng = location.getLongitude();
    latLongString = "Lat:" + lat + "\nLong:" + lng;

    double latitude = location.getLatitude();
    double longitude = location.getLongitude();

    Geocoder gc = new Geocoder(this, Locale.getDefault());
    try {
      List<Address> addresses = gc.getFromLocation(latitude, longitude, 1);
      StringBuilder sb = new StringBuilder();
      if (addresses.size() > 0) {
        Address address = addresses.get(0);

        for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
          sb.append(address.getAddressLine(i)).append("\n");

          sb.append(address.getLocality()).append("\n");
          sb.append(address.getPostalCode()).append("\n");
          sb.append(address.getCountryName());
      }
      addressString = sb.toString();
    } catch (IOException e) {}
  } else {
    latLongString = "No location found";
  }
  myLocationText.setText("Your Current Position is:\n" +
                         latLongString + "\n" + addressString);
}
```

If you run the example now, it should appear as shown in Figure 7-4.



Figure 7-4

Creating Map-Based Activities

The `MapView` provides a compelling User Interface option for presentation of geographical data.

One of the most intuitive ways of providing context for a physical location or address is to display it on a map. Using a `MapView`, you can create Activities that feature an interactive map.

Map Views support annotation using both Overlays and by pinning Views to geographical locations. Map Views offer full programmatic control of the map display, letting you control the zoom, location, and display modes — including the option to display satellite, street, and traffic views.

In the following sections, you'll see how to use Overlays and the MapController to create dynamic map-based Activities. Unlike online mashups, your map Activities will run natively on the device, allowing you to leverage its hardware and mobility to provide a more customized and personal user experience.

Introducing MapView and MapActivity

This section introduces several classes used to support Android maps:

- `MapView` is the actual Map View (control).
- `MapActivity` is the base class you extend to create a new Activity that can include a Map View. The `MapActivity` class handles the application life cycle and background service management required for displaying maps. As a result, you can only use a `MapView` within `MapActivity`-derived Activities.
- `Overlay` is the class used to annotate your maps. Using Overlays, you can use a Canvas to draw onto any number of layers that are displayed on top of a Map View.
- `MapController` is used to control the map, allowing you to set the center location and zoom levels.
- `MyLocationOverlay` is a special overlay that can be used to display the current position and orientation of the device.
- `ItemizedOverlays` and `OverlayItems` are used together to let you create a layer of map markers, displayed using drawable with associated text.

Creating a Map-Based Activity

To use maps in your applications, you need to create a new Activity that extends `MapActivity`. Within it, add a `MapView` to the layout to display a Google Maps interface element. The Android map library is not a standard package; as an optional API, it must be explicitly included in the application manifest before it can be used. Add the library to your manifest using a `uses-library` tag within the `application` node, as shown in the XML snippet below:

```
<uses-library android:name="com.google.android.maps"/>
```

Google Maps downloads the map tiles on demand; as a result, it implicitly requires permission to use the Internet. To see map tiles in your Map View, you need to add a `uses-permission` tag to your application manifest for `android.permission.INTERNET`, as shown below:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Once you've added the library and configured your permission, you're ready to create your new map-based Activity.

`MapView` controls can only be used within an Activity that extends `MapActivity`. Override the `onCreate` method to lay out the screen that includes a `MapView`, and override `isRouteDisplayed` to return `true` if the Activity will be displaying routing information (such as traffic directions).

The following skeleton code shows the framework for creating a new map-based Activity:

```
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;
import android.os.Bundle;

public class MyMapActivity extends MapActivity {
  private MapView mapView;
  private MapController mapController;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.map_layout);
    mapView = (MapView)findViewById(R.id.map_view);
  }

  @Override
  protected boolean isRouteDisplayed() {
    // IMPORTANT: This method must return true if your Activity
    // is displaying driving directions. Otherwise return false.
    return false;
  }
}
```

The corresponding layout file used to include the `MapView` is shown below. Note that you need to include a maps API key in order to use a Map View in your application.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.google.android.maps.MapView
    android:id="@+id/map_view"
    android:layout_width="fill_parent"
```

```
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="mymapapikey"
    />
</LinearLayout>
```

At the time of publication, it was unclear how developers would apply for map keys. Invalid or disabled API keys will result in your `MapView` not loading the map image tiles. Until this process is revealed, you can use any text as your API key value.

Figure 7-5 shows an example of a basic map-based Activity.



Figure 7-5

Android currently recommends that you include no more than one `MapActivity` *and one* `MapView` *in each application.*

Configuring and Using Map Views

The `MapView` class is a View that displays the actual map; it includes several options for deciding how the map is displayed.

By default, the Map View will show the standard street map, as shown in Figure 7-5. In addition, you can choose to display a satellite view, StreetView, and expected traffic, as shown in the code snippet below:

```
mapView.setSatellite(true);
mapView.setStreetView(true);
mapView.setTraffic(true);
```

You can also query the Map View to find the current and maximum available zoom level, as well as the center point and currently visible longitude and latitude span (in decimal degrees). The latter (shown below) is particularly useful for performing geographically limited Geocoder lookups:

```
GeoPoint center = mapView.getMapCenter();
int latSpan = mapView.getLatitudeSpan();
int longSpan = mapView.getLongitudeSpan();
```

You can also optionally display the standard map zoom controls. The following code snippet shows how to get a reference to the Zoom Control View and pin it to a screen location. The Boolean parameter lets you assign focus to the controls once they're added.

```
int y = 10;
int x = 10;

MapView.LayoutParams lp;
lp = new MapView.LayoutParams(MapView.LayoutParams.WRAP_CONTENT,
                              MapView.LayoutParams.WRAP_CONTENT,
                              x, y,
                              MapView.LayoutParams.TOP_LEFT);

View zoomControls = mapView.getZoomControls();
mapView.addView(zoomControls, lp);
mapView.displayZoomControls(true);
```

The technique used to pin the zoom controls to the `MapView` is covered in more detail later in this chapter.

Using the Map Controller

You use the Map Controller to pan and zoom a `MapView`. You can get a reference to a `MapView`'s controller using `getController`, as shown in the following code snippet:

```
MapController mapController = myMapView.getController();
```

Map locations in the Android mapping classes are represented by `GeoPoint` objects, which contain latitude and longitude measured in microdegrees (i.e., degrees multiplied by 1E6 [or 1,000,000]).

Before you can use the latitude and longitude values stored in the Location objects used by the location-based services, you'll need to convert them to microdegrees and store them as GeoPoints, as shown in the following code snippet:

```
Double lat = 37.422006*1E6;
Double lng = -122.084095*1E6;
GeoPoint point = new GeoPoint(lat.intValue(), lng.intValue());
```

Re-center and zoom the `MapView` using the `setCenter` and `setZoom` methods available on the `MapView`'s `MapController`, as shown in the snippet below:

```
mapController.setCenter(point);
mapController.setZoom(1);
```

When using `setZoom`, 1 represents the widest (or furthest away) zoom and 21 the tightest (nearest) view.

The actual zoom level available for a specific location depends on the resolution of Google's maps and imagery for that area. You can also use `zoomIn` and `zoomOut` to change the zoom level by one step.

The `setCenter` method will "jump" to a new location; to show a smooth transition, use `animateTo` as shown in the code below:

```
mapController.animateTo(point);
```

Mapping "Where Am I?"

In the following code example, the "Where Am I?" project is extended again. This time you'll add mapping functionality by transforming it into a Map Activity. As the device location changes, the map will automatically re-center on the new position.

1. Start by adding the `uses-permission` tag for Internet access to the application manifest. Also import the Android maps library within the `application` tag.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.paad.whereami">
  <application
    android:icon="@drawable/icon">
    <activity
      android:name=".WhereAmI"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <uses-library android:name="com.google.android.maps"/>
  </application>

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

</manifest>
```

2. Change the inheritance of `WhereAmI` to descend from `MapActivity` instead of `Activity`. You'll also need to include an override for the `isRouteDisplayed` method. Because this Activity won't show routing directions, you can return `false`.

```
public class WhereAmI extends MapActivity {

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }

  [ ... existing Activity code ... ]

}
```

3. Modify the main.xml layout resource to include a `MapView` using the fully qualified class name. Be sure to include an `android:apikey` attribute within the `com.android.MapView` node. If you have an Android maps API key, use it here.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:id="@+id/myLocationText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
  />
  <com.google.android.maps.MapView
    android:id="@+id/myMapView"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:enabled="true"
    android:clickable="true"
    android:apiKey="myMapKey"
  />
</LinearLayout>
```

4. Running the application now should display the original geolocation text, with a `MapView` beneath it, as shown in Figure 7-6.



Figure 7-6

5. Configure the Map View and store a reference to its `MapController` as an instance variable. Set up the Map View display options to show the satellite and `StreetView` and zoom in for a closer look.

```
MapController mapController;

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

  // Get a reference to the MapView
  MapView myMapView = (MapView)findViewById(R.id.myMapView);
  // Get the Map View's controller
  mapController = myMapView.getController();

  // Configure the map display options
  myMapView.setSatellite(true);
  myMapView.setStreetView(true);
  myMapView.displayZoomControls(false);

  // Zoom in
  mapController.setZoom(17);

  LocationManager locationManager;
  String context = Context.LOCATION_SERVICE;
  locationManager = (LocationManager)getSystemService(context);

  Criteria criteria = new Criteria();
  criteria.setAccuracy(Criteria.ACCURACY_FINE);
  criteria.setAltitudeRequired(false);
  criteria.setBearingRequired(false);
  criteria.setCostAllowed(true);
  criteria.setPowerRequirement(Criteria.POWER_LOW);
  String provider = locationManager.getBestProvider(criteria, true);

  Location location = locationManager.getLastKnownLocation(provider);

  updateWithNewLocation(location);

  locationManager.requestLocationUpdates(provider, 2000, 10,
                                         locationListener);
}
```

6. The final step is to modify the `updateWithNewLocation` method to re-center the map to the current location using the Map Controller.

```
private void updateWithNewLocation(Location location) {
  String latLongString;
  TextView myLocationText;
  myLocationText = (TextView)findViewById(R.id.myLocationText);
```

```
        String addressString = "No address found";

        if (location != null) {
          // Update the map location.
          Double geoLat = location.getLatitude()*1E6;
          Double geoLng = location.getLongitude()*1E6;
          GeoPoint point = new GeoPoint(geoLat.intValue(),
                                        geoLng.intValue());

          mapController.animateTo(point);

          double lat = location.getLatitude();
          double lng = location.getLongitude();
          latLongString = "Lat:" + lat + "\nLong:" + lng;

          double latitude = location.getLatitude();
          double longitude = location.getLongitude();

          Geocoder gc = new Geocoder(this, Locale.getDefault());
          try {
            List<Address> addresses = gc.getFromLocation(latitude, longitude, 1);
            StringBuilder sb = new StringBuilder();
            if (addresses.size() > 0) {
              Address address = addresses.get(0);

              for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
                sb.append(address.getAddressLine(i)).append("\n");

                sb.append(address.getLocality()).append("\n");
                sb.append(address.getPostalCode()).append("\n");
                sb.append(address.getCountryName());
            }
            addressString = sb.toString();
          } catch (IOException e) {}
        } else {
          latLongString = "No location found";
        }
        myLocationText.setText("Your Current Position is:\n" +
                               latLongString + "\n" + addressString);
    }
```

Creating and Using Overlays

Overlays are a way to add annotations and click handling to `MapViews`. Each Overlay lets you draw 2D primitives including text, lines, images and shapes directly onto a canvas, which is then overlaid onto a Map View.

You can add several Overlays onto a single map. All the Overlays assigned to a Map View are added as layers, with newer layers potentially obscuring older ones. User clicks are passed through the stack until they are either handled by an Overlay or registered as a click on the Map View itself.

Creating New Overlays

Each overlay is a canvas with a transparent background that you can layer on top of a Map View and use to handle map touch events.

To add a new Overlay, create a new class that extends `Overlay`. Override the `draw` method to draw the annotations you want to add, and override `onTap` to react to user clicks (generally when the user taps an annotation added by this overlay).

The following code snippet shows the framework for creating a new Overlay that can draw annotations and handle user clicks:

```
import android.graphics.Canvas;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;

public class MyOverlay extends Overlay {

  @Override
  public void draw(Canvas canvas, MapView mapView, boolean shadow) {
    if (shadow == false) {
      [ ... Draw annotations on main map layer ... ]
    }
    else {
      [ ... Draw annotations on the shadow layer ... ]
    }
  }

  @Override
  public boolean onTap(GeoPoint point, MapView mapView) {
   // Return true if screen tap is handled by this overlay
   return false;
  }
}
```

Introducing Projections

The Canvas used to draw Overlay annotations is a standard `Canvas` that represents the visible display surface. To add annotations based on physical locations, you need to convert between geographical points and screen coordinates.

The Projection class lets you translate between latitude/longitude coordinates (stored as `GeoPoints`) and x/y screen pixel coordinates (stored as `Points`).

A map's Projection may change between subsequent calls to draw, so it's good practice to get a new instance each time. Get a Map View's Projection by calling `getProjection`, as shown in the snippet below:

```
Projection projection = mapView.getProjection();
```

Use the `fromPixel` and `toPixel` methods to translate from GeoPoints to Points and vice versa.

For performance reasons, the `toPixel` Projection method is best used by passing a Point object to be populated (rather than relying on the return value), as shown below:

```
Point myPoint = new Point();

// To screen coordinates
projection.toPixels(geoPoint, myPoint);

// To GeoPoint location coordinates
projection.fromPixels(myPoint.x, myPoint.y);
```

Drawing on the Overlay Canvas

Canvas drawing for Overlays is handled by overriding the Overlay's `draw` handler.

The passed-in Canvas is the surface on which you draw your annotations, using the same techniques introduced in Chapter 4 when creating custom User Interfaces for Views. The Canvas object includes the methods for drawing 2D primitives on your map (including lines, text, shapes, ellipses, images, etc.). Use `Paint` objects to define the style and color.

The following code snippet uses a Projection to draw text and an ellipse at a given location:

```
@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow) {
  Projection projection = mapView.getProjection();

  Double lat = -31.960906*1E6;
  Double lng = 115.844822*1E6;
  GeoPoint geoPoint = new GeoPoint(lat.intValue(), lng.intValue());

  if (shadow == false) {
    Point myPoint = new Point();
    projection.toPixels(geoPoint, myPoint);

    // Create and setup your paint brush
    Paint paint = new Paint();
    paint.setARGB(250, 255, 0, 0);
    paint.setAntiAlias(true);
    paint.setFakeBoldText(true);

    // Create the circle
    int rad = 5;
    RectF oval = new RectF(myPoint.x-rad, myPoint.y-rad,
                           myPoint.x+rad, myPoint.y+rad);

    // Draw on the canvas
    canvas.drawOval(oval, paint);
    canvas.drawText("Red Circle", myPoint.x+rad, myPoint.y, paint);
  }
}
```

For more advanced drawing features, check out Chapter 11, where gradients, strokes, and filters are introduced that provide powerful tools for drawing attractive and compelling map overlays.

Handling Map Tap Events

To handle map taps (user clicks), override the `onTap` event handler within the Overlay extension class.

The `onTap` handler receives two parameters:

- A `GeoPoint` that contains the latitude/longitude of the map location tapped
- The `MapView` that was tapped to trigger this event

When overriding `onTap`, the method should return `true` if it has handled a particular tap and `false` to let another overlay handle it, as shown in the skeleton code below:

```
@Override
public boolean onTap(GeoPoint point, MapView mapView) {
  // Perform hit test to see if this overlay is handling the click
  if ([... perform hit test ... ]) {
    [ ... execute on tap functionality ... ]
    return true;
  }

  // If not handled return false
  return false;
}
```

Adding and Removing Overlays

Each `MapView` contains a list of Overlays currently displayed. You can get a reference to this list by calling `getOverlays`, as shown in the snippet below:

```
List<Overlay> overlays = mapView.getOverlays();
```

Adding and removing items from the list is thread safe and synchronized, so you can modify and query the list safely. Iterating over the list should still be done within a synchronization block synchronized on the List.

To add an Overlay onto a Map View, create a new instance of the Overlay, and add it to the list, as shown in the snippet below:

```
List<Overlay> overlays = mapView.getOverlays();

MyOverlay myOverlay = new MyOverlay();
overlays.add(myOverlay);
mapView.postInvalidate();
```

The added Overlay will be displayed the next time the Map View is redrawn, so it's usually good practice to call `postInvalidate` after you modify the list to update the changes on the map display.

Annotating "Where Am I?"

This final modification to "Where Am I?" creates and adds a new Overlay that displays a red circle at the device's current position.

1. Start by creating a new `MyPositionOverlay` Overlay class in the WhereAmI project.

```
package com.paad.whereami;

import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Point;
import android.graphics.RectF;
import android.location.Location;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;
import com.google.android.maps.Projection;

public class MyPositionOverlay extends Overlay {

  @Override
  public void draw(Canvas canvas, MapView mapView, boolean shadow) {
  }

  @Override
  public boolean onTap(GeoPoint point, MapView mapView) {
    return false;
  }
}
```

2. Create a new instance variable to store the current Location, and add setter and getter methods for it.

```
Location location;

public Location getLocation() {
  return location;
}
public void setLocation(Location location) {
  this.location = location;
}
```

3. Override the `draw` method to add a small red circle at the current location.

```
private final int mRadius = 5;

@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow) {
  Projection projection = mapView.getProjection();

  if (shadow == false) {
    // Get the current location
    Double latitude = location.getLatitude()*1E6;
    Double longitude = location.getLongitude()*1E6;
    GeoPoint geoPoint;
    geoPoint = new GeoPoint(latitude.intValue(),longitude.intValue());

    // Convert the location to screen pixels
```

```
        Point point = new Point();
        projection.toPixels(geoPoint, point);

        RectF oval = new RectF(point.x - mRadius, point.y - mRadius,
                               point.x + mRadius, point.y + mRadius);

        // Setup the paint
        Paint paint = new Paint();
        paint.setARGB(250, 255, 0, 0);
        paint.setAntiAlias(true);
        paint.setFakeBoldText(true);

        Paint backPaint = new Paint();
        backPaint.setARGB(175, 50, 50, 50);
        backPaint.setAntiAlias(true);

        RectF backRect = new RectF(point.x + 2 + mRadius,
                                   point.y - 3*mRadius,
                                   point.x + 65, point.y + mRadius);

        // Draw the marker
        canvas.drawOval(oval, paint);
        canvas.drawRoundRect(backRect, 5, 5, backPaint);

        canvas.drawText("Here I Am", point.x + 2*mRadius, point.y, paint);
      }
      super.draw(canvas, mapView, shadow);
    }
```

4. Now open the `WhereAmI` Activity class, and add the `MyPositionOverlay` to the `MapView`.

   Start by adding a new instance variable to store the `MyPositionOverlay`, then override `onCreate` to create a new instance of the class, and add it to the `MapView`'s Overlay list.

```
MyPositionOverlay positionOverlay;

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

  MapView myMapView = (MapView)findViewById(R.id.myMapView);
  mapController = myMapView.getController();

  myMapView.setSatellite(true);
  myMapView.setStreetView(true);
  myMapView.displayZoomControls(false);

  mapController.setZoom(17);

  // Add the MyPositionOverlay
  positionOverlay = new MyPositionOverlay();
  List<Overlay> overlays = myMapView.getOverlays();
```

```
        overlays.add(positionOverlay);

        LocationManager locationManager;
        String context = Context.LOCATION_SERVICE;
        locationManager = (LocationManager)getSystemService(context);

        Criteria criteria = new Criteria();
        criteria.setAccuracy(Criteria.ACCURACY_FINE);
        criteria.setAltitudeRequired(false);
        criteria.setBearingRequired(false);
        criteria.setCostAllowed(true);
        criteria.setPowerRequirement(Criteria.POWER_LOW);
        String provider = locationManager.getBestProvider(criteria, true);

        Location location = locationManager.getLastKnownLocation(provider);

        updateWithNewLocation(location);

        locationManager.requestLocationUpdates(provider, 2000, 10,
                                               locationListener);
    }
```

5. Finally, update the `updateWithNewLocation` method to pass the new location to the overlay.

```
    private void updateWithNewLocation(Location location) {
      String latLongString;
      TextView myLocationText;
      myLocationText = (TextView)findViewById(R.id.myLocationText);
      String addressString = "No address found";

      if (location != null) {
        // Update my location marker
        positionOverlay.setLocation(location);

        // Update the map location.
        Double geoLat = location.getLatitude()*1E6;
        Double geoLng = location.getLongitude()*1E6;
        GeoPoint point = new GeoPoint(geoLat.intValue(),
                                      geoLng.intValue());

        mapController.animateTo(point);

        double lat = location.getLatitude();
        double lng = location.getLongitude();
        latLongString = "Lat:" + lat + "\nLong:" + lng;

        double latitude = location.getLatitude();
        double longitude = location.getLongitude();

        Geocoder gc = new Geocoder(this, Locale.getDefault());
        try {
          List<Address> addresses = gc.getFromLocation(latitude, longitude, 1);
          StringBuilder sb = new StringBuilder();
```

```
            if (addresses.size() > 0) {
              Address address = addresses.get(0);

              for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
                sb.append(address.getAddressLine(i)).append("\n");

                sb.append(address.getLocality()).append("\n");
                sb.append(address.getPostalCode()).append("\n");
                sb.append(address.getCountryName());
              }
              addressString = sb.toString();
            } catch (IOException e) {}
          } else {
            latLongString = "No location found";
          }
          myLocationText.setText("Your Current Position is:\n" +
                                 latLongString + "\n" + addressString);
        }
```

When run, your application will display your current device location with a red circle and supporting text, as shown in Figure 7-7.



Figure 7-7

It's worth noting that this is not the preferred technique for displaying your current location on a map. This functionality is implemented natively by Android through the `MyLocationOverlay` *class. If you want to display and follow your current location, you should consider using this class (as shown in the next section) instead of implementing it manually as shown here.*

Introducing MyLocationOverlay

The `MyLocationOverlay` class is a special Overlay designed to show your current location and orientation on a `MapView`.

To use the My Location Overlay, you need to create a new instance, passing in the application Context and target Map View, and add it to the `MapView`'s Overlay list, as shown below:

```
List<Overlay> overlays = mapView.getOverlays();
MyLocationOverlay myLocationOverlay = new MyLocationOverlay(this, mapView);
overlays.add(myLocationOverlay);
```

You can use the My Location Overlay to display both your current location (represented as a flashing blue marker) and orientation (shown as a compass on the map display).

The following snippet shows how to enable both the compass and marker; in this instance, the Map View's `MapController` is also passed in, allowing the overlay to automatically scroll the map if the marker moves off screen.

```
myLocationOverlay.enableCompass();
myLocationOverlay.enableMyLocation(mapView.getMapController());
```

Introducing ItemizedOverlays and OverlayItems

`OverlayItems` are used to supply simple maker functionality to your `MapViews` using the `ItemizedOverlay` class.

You can create your own Overlays that draw markers onto a map, but `ItemizedOverlays` provide a convenient shortcut, letting you assign a marker image and associated text to a particular geographical position. The `ItemizedOverlay` instance handles the drawing, placement, click handling, focus control, and layout optimization of each `OverlayItem` marker for you.

> **At the time of going to print, the `ItemizedOverlay`/`OverlayItem` functionality was not fully supported. While it was possible to implement each required class, the markers were not displayed on the map.**

To add an `ItemizedOverlay` marker layer to your map, start by creating a new class that extends `ItemizedOverlay<OverlayItem>`, as shown in the skeleton code below:

```
import android.graphics.drawable.Drawable;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.ItemizedOverlay;
import com.google.android.maps.OverlayItem;

public class MyItemizedOverlay extends ItemizedOverlay<OverlayItem> {

  public MyItemizedOverlay(Drawable defaultMarker) {
    super(defaultMarker);
```

```
    // Create each of the overlay items included in this layer.
    populate();
  }

  @Override
  protected OverlayItem createItem(int index) {
    switch (index) {
      case 1:
        Double lat = 37.422006*1E6;
        Double lng = -122.084095*1E6;
        GeoPoint point = new GeoPoint(lat.intValue(), lng.intValue());

        OverlayItem oi;
        oi = new OverlayItem(point, "Marker", "Marker Text");
        return oi;
    }

    return null;
  }

  @Override
  public int size() {
    // Return the number of markers in the collection
    return 1;
  }

}
```

`ItemizedOverlay` is a generic class that lets you create extensions based on any `OverlayItem`-derived subclass.

Within the implementation, override `size` to return the number of markers to display and `createItem` to create a new item based on the index of each marker. You will also need to make a call to `populate` within the class's constructor. This call is a requirement and is used to trigger the creation of each `OverlayItem`; it must be called as soon as you have the data required to create all the items.

To add an `ItemizedOverlay` implementation to your map, create a new instance (passing in the default drawable marker image to use), and add it to the map's Overlay list, as shown in the following snippet:

```
List<Overlay> overlays = mapView.getOverlays();

MyItemizedOverlay markrs = new MyItemizedOverlay(r.getDrawable(R.drawable.marker));
overlays.add(markrs);
```

Pinning Views to the Map and Map Positions

Previously in this chapter, you saw how to add the Zoom View to a Map View by pinning it to a specific screen location. You can pin any View-derived object to a Map View (including layouts and other View Groups), attaching it to either a screen position or a geographical map location.

In the latter case, the View will move to follow its pinned position on the map, effectively acting as an interactive map marker. As a more resource-intensive solution, this is usually reserved for supplying the detail "balloons" often displayed on mashups to provide further detail when a marker is clicked.

Both pinning mechanisms are implemented by calling `addView` on the `MapView`, usually from the `onCreate` or `onRestore` methods within the `MapActivity` containing it. Pass in the View you want to pin and the layout parameters to use.

The `MapView.LayoutParams` parameters you pass in to `addView` determine how, and where, the View is added to the map.

To add a new View to the map relative to the screen, specify a new `MapView.LayoutParams` including arguments that set the height and width of the View, the x/y screen coordinates to pin to, and the alignment to use for positioning, as shown below:

```
int y = 10;
int x = 10;

MapView.LayoutParams screenLP;
screenLP = new MapView.LayoutParams(MapView.LayoutParams.WRAP_CONTENT,
                                    MapView.LayoutParams.WRAP_CONTENT,
                                    x, y,
                                    MapView.LayoutParams.TOP_LEFT);

EditText editText1 = new EditText(getApplicationContext());
editText1.setText("Screen Pinned");

mapView.addView(editText1, screenLP);
```

To pin a View relative to a physical map location, pass four parameters when constructing the new `MapView LayoutParams`, representing the height, width, GeoPoint to pin to, and the layout alignment.

```
Double lat = 37.422134*1E6;
Double lng = -122.084069*1E6;
GeoPoint geoPoint = new GeoPoint(lat.intValue(), lng.intValue());

MapView.LayoutParams geoLP;
geoLP = new MapView.LayoutParams(MapView.LayoutParams.WRAP_CONTENT,
                                 MapView.LayoutParams.WRAP_CONTENT,
                                 geoPoint,
                                 MapView.LayoutParams.TOP_LEFT);

EditText editText2 = new EditText(getApplicationContext());
editText2.setText("Location Pinned");

mapView.addView(editText2, geoLP);
```

Panning the map will leave the first `TextView` stationary in the upper left corner, while the second `TextView` will move to remain pinned to a particular position on the map.

To remove a View from a MapView, call removeView, passing in the View instance you wish to remove, as shown below:

```
mapView.removeView(editText2);
```

Mapping Earthquakes Example

The following step-by-step guide demonstrates how to build a map-based Activity for the Earthquake project you started in Chapter 5. The new MapActivity will display a map of recent earthquakes using techniques you learned within this chapter.

1. Create a new earthquake_map.xml layout resource that includes a MapView, being sure to include an android:id attribute and a android:apiKey attribute that contains your Android Maps API key.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.google.android.maps.MapView
    android:id="@+id/map_view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:enabled="true"
    android:clickable="true"
    android:apiKey="myapikey"
  />
</LinearLayout>
```

2. Create a new EarthquakeMap Activity that inherits from MapActivity. Use setContentView within onCreate to inflate the earthquake_map resource you created in Step 1.

```
package com.paad.earthquake;

import android.os.Bundle;
import com.google.android.maps.MapActivity;

public class EarthquakeMap extends MapActivity {

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.earthquake_map);
  }

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }

}
```

3. Update the application manifest to include your new `EarthquakeMap` Activity and import the map library.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.paad.earthquake">
  <application android:icon="@drawable/icon">
    <activity
      android:name=".Earthquake"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity android:name=".Preferences"
              android:label="Earthquake Preferences"/>
    <activity android:name=".EarthquakeMap"
              android:label="View Earthquakes"/>
    <provider android:name=".EarthquakeProvider"
              android:authorities="com.paad.provider.earthquake" />
    <uses-library android:name="com.google.android.maps"/>
  </application>
  <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```

4. Add a new menu option to the Earthquake Activity to display the EarthquakeMap Activity.

   4.1. Start by adding a new string to the strings.xml resource for the menu text.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Earthquake</string>
  <string name="quake_feed">
    http://earthquake.usgs.gov/eqcenter/catalogs/1day-M2.5.xml
  </string>
  <string name="menu_update">Refresh Earthquakes</string>
  <string name="auto_update_prompt">Auto Update?</string>
  <string name="update_freq_prompt">Update Frequency</string>
  <string name="min_quake_mag_prompt">Minimum Quake Magnitude</string>
  <string name="menu_preferences">Preferences</string>
  <string name="menu_earthquake_map">Earthquake Map</string>
</resources>
```

   4.2. Then add a new menu identifier before modifying the `onCreateOptionsMenu` handler to add the new Menu Item. It should use the text defined in Step 4.1, and when selected, it should fire an Intent to explicitly start the `EarthquakeMap` Activity.

```
static final private int MENU_EARTHQUAKE_MAP = Menu.FIRST+2;

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

  menu.add(0, MENU_UPDATE, Menu.NONE, R.string.menu_update);
  menu.add(0, MENU_PREFERENCES, Menu.NONE, R.string.menu_preferences);
```

```
    Intent startMap = new Intent(this, EarthquakeMap.class);
    menu.add(0, MENU_EARTHQUAKE_MAP,
                Menu.NONE,
                R.string.menu_earthquake_map).setIntent(startMap);
    return true;
  }
```

5. Now create a new `EarthquakeOverlay` class that extends `Overlay`. It will draw the position and magnitude of each earthquake on the Map View.

```
package com.paad.earthquake;

import java.util.ArrayList;
import android.database.Cursor;
import android.database.DataSetObserver;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Point;
import android.graphics.RectF;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;
import com.google.android.maps.Projection;

public class EarthquakeOverlay extends Overlay {

  @Override
  public void draw(Canvas canvas, MapView mapView, boolean shadow) {
    Projection projection = mapView.getProjection();

    if (shadow == false) {
     // TODO: Draw earthquakes
    }
  }
}
```

   5.1. Add a new constructor that accepts a `Cursor` to the current earthquake data, and store that Cursor as an instance variable.

```
Cursor earthquakes;

public EarthquakeOverlay(Cursor cursor, ContentResolver resolver) {
  super();

  earthquakes = cursor;
}
```

   5.2. Create a new `refreshQuakeLocations` method that iterates over the results Cursor and extracts the location of each earthquake, extracting the latitude and longitude before storing each coordinate in a List of `GeoPoints`.

```
ArrayList<GeoPoint> quakeLocations;

private void refreshQuakeLocations() {
  if (earthquakes.moveToFirst())
```

```
    do {
      Double lat;
      lat = earthquakes.getFloat(EarthquakeProvider.LATITUDE_COLUMN) * 1E6;
      Double lng;
      lng = earthquakes.getFloat(EarthquakeProvider.LONGITUDE_COLUMN) * 1E6;

      GeoPoint geoPoint = new GeoPoint(lng.intValue(), lat.intValue());

      quakeLocations.add(geoPoint);

    } while(earthquakes.moveToNext());
}
```

5.3. Call `refreshQuakeLocations` from the Overlay's constructor. Also register a `DataSetObserver` on the results Cursor that refreshes the Earthquake Location list if a change in the Earthquake Cursor is detected.

```
public EarthquakeOverlay(Cursor cursor) {
  super();
  earthquakes = cursor;

  quakeLocations = new ArrayList<GeoPoint>();
  refreshQuakeLocations();
  earthquakes.registerDataSetObserver(new DataSetObserver() {
    @Override
    public void onChanged() {
      refreshQuakeLocations();
    }
  });
}
```

5.4. Complete the `EarthquakeOverlay` by overriding the `draw` method to iterate over the list of `GeoPoints`, drawing a marker at each earthquake location. In this example, a simple red circle is drawn, but it could easily be modified to include additional information, such as by adjusting the size of each circle based on the magnitude of the quake.

```
int rad = 5;

@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow) {
  Projection projection = mapView.getProjection();

  // Create and setup your paint brush
  Paint paint = new Paint();
  paint.setARGB(250, 255, 0, 0);
  paint.setAntiAlias(true);
  paint.setFakeBoldText(true);

  if (shadow == false) {
    for (GeoPoint point : quakeLocations) {

      Point myPoint = new Point();
      projection.toPixels(point, myPoint);
```

```
            RectF oval = new RectF(myPoint.x-rad, myPoint.y-rad,
                                   myPoint.x+rad, myPoint.y+rad);

            canvas.drawOval(oval, paint);
          }
        }
      }
```

6. Return to the `EarthquakeMap` class. Within the `onCreate` method, create a Cursor that returns the earthquakes you want to display on the map. Use this Cursor to create a new `EarthquakeOverlay` before adding the new instance to the Map View's list of overlays.

```
Cursor earthquakeCursor;

@Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.earthquake_map);

String earthquakeURI = EarthquakeProvider.CONTENT_URI;
    earthquakeCursor = getContentResolver().query(earthquakeURI,
                                                  null, null, null, null);

    MapView earthquakeMap = (MapView)findViewById(R.id.map_view);
    EarthquakeOverlay eo = new EarthquakeOverlay(earthquakeCursor);
    earthquakeMap.getOverlays().add(eo);
}
```

7. Finally, override `onResume` to call `requery` on the Earthquake result set whenever this Activity becomes visible. Also, override `onPause` and `onDestroy` to optimize use of the Cursor resources.

```
@Override
public void onResume() {
  earthquakeCursor.requery();
  super.onResume();
}

@Override
public void onPause() {
  earthquakeCursor.deactivate();
  super.onPause();
}

@Override
public void onDestroy() {
  earthquakeCursor.close();
  super.onDestroy();
}
```

8. If you run the application and select Earthquake Map from the main menu, your application should appear as shown in Figure 7-8.



Figure 7-8

Summary

Location-based services, the Geocoder, and MapViews are available to create intuitive, location-aware applications that feature geographical information.

This chapter introduced the Geocoder and showed how to perform forward and reverse geocoding lookups to translate between map coordinates and street addresses. You were introduced to location-based services, used to find the current geographical position of the device. You also used them to track movement and create proximity alerts.

Then you created interactive map applications. Using Overlays and Views, you annotated `MapViews` with 2D graphics, as well as markers in the form of `OverlayItems` and Views (including ViewGroups and layouts).

In Chapter 8, you'll learn how to work from the background. You'll be introduced to the Service component and learn how to move processing onto background threads. To interact with the user while hidden from view, you'll use Toasts to display transient messages and the Notification Manager to ring, vibrate, and flash the phone.

8

Working in the Background

Because of the limited screen size of most mobile devices, typically only one application is visible and active on a device at any given time. This offers a perfect environment for applications that run in the background without a User Interface — responding to events, polling for data, or updating Content Providers.

Android offers the `Service` class to create application components specifically to handle operations and functionality that should run silently, without a User Interface. Android accords Services a higher priority than inactive Activities, so they're less likely to be killed when the system requires resources. In fact, should the run time prematurely terminate a Service that's been started, it will be restarted as soon as sufficient resources are available. By using Services, you can ensure that your applications continue to run and respond to events, even when they're not in active use.

Services run without a dedicated GUI, but, like Activities and Broadcast Receivers, they still execute in the main thread of the application's process. To help keep your applications responsive, you'll learn to move time-consuming processes (like network lookups) into background threads.

Android offers several techniques for application components (particularly Services) to communicate with users without an Activity providing a direct User Interface. In this chapter, you'll learn how to use Notifications and Toasts to politely alert and update users, without interrupting the active application.

Toasts are a transient, non-modal Dialog-box mechanism used to display information to users without stealing focus from the active application. You'll learn to display Toasts from any application component to send unobtrusive on-screen messages to your users.

Where Toasts are silent and transient, *Notifications* represent a more robust mechanism for alerting users. For many users, when they're not actively using their mobile phones, they sit silent and unwatched in a pocket or on a desk until it rings, vibrates, or flashes. Should a user miss these alerts, status bar icons are used to indicate that an event has occurred. All of these attention-grabbing antics are available within Android as Notifications.

Alarms provide a mechanism for firing Intents at set times, outside the control of your application life cycle. You'll learn to use Alarms to start Services, open Activities, or broadcast Intents based on either the clock time or the time elapsed since device boot. An Alarm will fire even after its owner application has been closed, and can (if required) wake a device from sleep.

Introducing Services

Unlike Activities, which present a rich graphical interface to users, Services run in the background — updating your Content Providers, firing Intents, and triggering Notifications. They are the perfect way to perform regular processing or handle events even after your application's Activities are invisible, inactive, or have been closed.

With no visual interface, Services are started, stopped, and controlled from other application components including other Services, Activities, and Broadcast Receivers. If your application regularly, or continuously, performs actions that don't depend directly on user input, Services may be the answer.

Started Services receive higher priority than inactive or invisible Activities, making them less likely to be terminated by the run time's resource management. The only time Android will stop a Service prematurely is when it's the only way for a foreground Activity to gain required resources; if that happens, your Service will be restarted automatically when resources become available.

Applications that update regularly but only rarely or intermittently need user interaction are good candidates for implementation as Services. MP3 players and sports-score monitors are examples of applications that should continue to run and update without an interactive visual component (Activity) visible.

Further examples can be found within the software stack itself; Android implements several Services including the Location Manager, Media Controller, and the Notification Manager.

Creating and Controlling Services

Services are designed to run in the background, so they need to be started, stopped, and controlled by other application components.

In the following sections, you'll learn how to create a new Service, and how to start and stop it using Intents and the `startService` method. Later you'll learn how to bind a Service to an Activity, providing a richer interface for interactivity.

Creating a Service

To define a Service, create a new class that extends the `Service` base class. You'll need to override `onBind` and `onCreate`, as shown in the following skeleton class:

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class MyService extends Service {

  @Override
  public void onCreate() {
```

```
    // TODO: Actions to perform when service is created.
  }

  @Override
  public IBinder onBind(Intent intent) {
    // TODO: Replace with service binding implementation.
    return null;
  }
}
```

In most cases, you'll also want to override `onStart`. This is called whenever the Service is started with a call to `startService`, so it can be executed several times within the Service's lifetime. You should ensure that your Service accounts for this.

The snippet below shows the skeleton code for overriding the `onStart` method:

```
@Override
public void onStart(Intent intent, int startId) {
  // TODO: Actions to perform when service is started.
}
```

Once you've constructed a new Service, you have to register it in the application manifest.

Do this by including a `service` tag within the `application` node. You can use attributes on the `service` tag to enable or disable the Service and specify any permissions required to access it from other applications using a `requires-permission` flag.

Below is the `service` tag you'd add for the skeleton Service you created above:

```
<service android:enabled="true" android:name=".MyService"></service>
```

Starting, Controlling, and Interacting with a Service

To start a Service, call `startService`; you can either implicitly specify a Service to start using an action against which the Service is registered, or you can explicitly specify the Service using its class.

If the Service requires permissions that your application does not have, this call will throw a `SecurityException`. The snippet below demonstrates both techniques available for starting a Service:

```
// Implicitly start a Service
startService(new Intent(MyService.MY_ACTION));
// Explicitly start a Service
startService(new Intent(this, MyService.class));
```

To use this example, you would need to include a `MY_ACTION` *property in the* `MyService` *class and use an Intent Filter to register it as a provider of* `MY_ACTION`.

To stop a Service, use `stopService`, passing an Intent that defines the Service to stop. This next code snippet first starts and then stops a Service both explicitly and by using the component name returned when calling `startService`:

```
ComponentName service = startService(new Intent(this, BaseballWatch.class));
// Stop a service using the service name.
stopService(new Intent(this, service.getClass()));
```

```
// Stop a service explicitly.
try {
  Class serviceClass = Class.forName(service.getClassName());
  stopService(new Intent(this, serviceClass));
} catch (ClassNotFoundException e) {}
```

If `startService` is called on a Service that's already running, the Service's `onStart` method will be executed again. Calls to `startService` do not nest, so a single call to `stopService` will terminate it no matter how many times `startService` has been called.

An Earthquake Monitoring Service Example

In this chapter, you'll modify the Earthquake example you started in Chapter 5 (and continued to enhance in Chapters 6 and 7). In this example, you'll move the earthquake updating and processing functionality into a separate Service component.

Later in this chapter, you'll build additional functionality within this Service, starting by moving the network lookup and XML parsing to a background thread. Later, you'll use Toasts and Notifications to alert users of new earthquakes.

1. Start by creating a new `EarthquakeService` that extends `Service`.

```
package com.paad.earthquake;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import java.util.Timer;
import java.util.TimerTask;

public class EarthquakeService extends Service {

  @Override
  public void onStart(Intent intent, int startId) {
    // TODO: Actions to perform when service is started.
  }

  @Override
  public void onCreate() {
    // TODO: Initialize variables, get references to GUI objects
  }

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }
}
```

2. Add this new Service to the manifest by adding a new `service` tag within the `application` node.

```
<service android:enabled="true" android:name=".EarthquakeService"></service>
```

3. Move the `refreshEarthquakes` and `addNewQuake` methods out of the `Earthquake` Activity and into the `EarthquakeService`.

You'll need to remove the calls to `addQuakeToArray` and `loadQuakesFromProvider` (leave both of these methods in the Earthquake Activity because they're still required). In the `EarthquakeService` also remove all references to the earthquakes ArrayList.

```
private void addNewQuake(Quake _quake) {
  ContentResolver cr = getContentResolver();
  // Construct a where clause to make sure we don't already have this
  // earthquake in the provider.
  String w = EarthquakeProvider.KEY_DATE + " = " + _quake.getDate().getTime();

  // If the earthquake is new, insert it into the provider.
  Cursor c = cr.query(EarthquakeProvider.CONTENT_URI, null, w, null, null);
  if (c.getCount()==0){
    ContentValues values = new ContentValues();

    values.put(EarthquakeProvider.KEY_DATE, _quake.getDate().getTime());
    values.put(EarthquakeProvider.KEY_DETAILS, _quake.getDetails());

    double lat = _quake.getLocation().getLatitude();
    double lng = _quake.getLocation().getLongitude();
    values.put(EarthquakeProvider.KEY_LOCATION_LAT, lat);
    values.put(EarthquakeProvider.KEY_LOCATION_LNG, lng);
    values.put(EarthquakeProvider.KEY_LINK, _quake.getLink());
    values.put(EarthquakeProvider.KEY_MAGNITUDE, _quake.getMagnitude());

    cr.insert(EarthquakeProvider.CONTENT_URI, values);
  }
  c.close();
}

private void refreshEarthquakes() {
  // Get the XML
  URL url;
  try {
    String quakeFeed = getString(R.string.quake_feed);
    url = new URL(quakeFeed);

    URLConnection connection;
    connection = url.openConnection();

    HttpURLConnection httpConnection = (HttpURLConnection)connection;
    int responseCode = httpConnection.getResponseCode();

    if (responseCode == HttpURLConnection.HTTP_OK) {
      InputStream in = httpConnection.getInputStream();

      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();

      // Parse the earthquake feed.
      Document dom = db.parse(in);
      Element docEle = dom.getDocumentElement();

      // Get a list of each earthquake entry.
      NodeList nl = docEle.getElementsByTagName("entry");
      if (nl != null && nl.getLength() > 0) {
```

```
            for (int i = 0 ; i < nl.getLength(); i++) {
              Element entry = (Element)nl.item(i);
              Element title;
              title = (Element)entry.getElementsByTagName("title").item(0);
              Element g;
              g = (Element)entry.getElementsByTagName("georss:point").item(0);
              Element when;
              when = (Element)entry.getElementsByTagName("updated").item(0);
              Element link = (Element)entry.getElementsByTagName("link").item(0);

              String details = title.getFirstChild().getNodeValue();
              String hostname = "http://earthquake.usgs.gov";
              String linkString = hostname + link.getAttribute("href");

              String point = g.getFirstChild().getNodeValue();
              String dt = when.getFirstChild().getNodeValue();
              SimpleDateFormat sdf;
              sdf = new SimpleDateFormat("yyyy-MM-dd'T'hh:mm:ss'Z'");
              Date qdate = new GregorianCalendar(0,0,0).getTime();
              try {
                qdate = sdf.parse(dt);
              } catch (ParseException e) {
                e.printStackTrace();
              }

              String[] location = point.split(" ");
              Location l = new Location("dummyGPS");
              l.setLatitude(Double.parseDouble(location[0]));
              l.setLongitude(Double.parseDouble(location[1]));

              String magnitudeString = details.split(" ")[1];
              int end =  magnitudeString.length()-1;
              double magnitude;
              magnitude = Double.parseDouble(magnitudeString.substring(0, end));

              details = details.split(",")[1].trim();

              Quake quake = new Quake(qdate, details, l, magnitude, linkString);

              // Process a newly found earthquake
              addNewQuake(quake);
            }
          }
        }
      } catch (MalformedURLException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      } catch (ParserConfigurationException e) {
        e.printStackTrace();
      } catch (SAXException e) {
        e.printStackTrace();
      }
      finally {
      }
    }
```

4. Within the Earthquake Activity, create a new `refreshEarthquakes` method. It should explicitly start the `EarthquakeService`.

```
private void refreshEarthquakes() {
  startService(new Intent(this, EarthquakeService.class));
}
```

5. Return to the `EarthquakeService`. Override the `onStart` and `onCreate` methods to support a new Timer that will be used to update the earthquake list. Use the `SharedPreference` object created in Chapter 6 to determine if the earthquakes should be regularly updated.

```
private Timer updateTimer;
private float minimumMagnitude;

@Override
public void onStart(Intent intent, int startId) {
  // Retrieve the shared preferences
  SharedPreferences prefs = getSharedPreferences(Preferences.USER_PREFERENCE,
                                                 Activity.MODE_PRIVATE);

  int minMagIndex = prefs.getInt(Preferences.PREF_MIN_MAG, 0);
  if (minMagIndex < 0)
    minMagIndex = 0;

  int freqIndex = prefs.getInt(Preferences.PREF_UPDATE_FREQ, 0);
  if (freqIndex < 0)
    freqIndex = 0;

  boolean autoUpdate = prefs.getBoolean(Preferences.PREF_AUTO_UPDATE, false);

  Resources r = getResources();
  int[] minMagValues = r.getIntArray(R.array.magnitude);
  int[] freqValues = r.getIntArray(R.array.update_freq_values);

  minimumMagnitude = minMagValues[minMagIndex];
  int updateFreq = freqValues[freqIndex];

  updateTimer.cancel();
  if (autoUpdate) {
    updateTimer = new Timer("earthquakeUpdates");
    updateTimer.scheduleAtFixedRate(doRefresh, 0, updateFreq*60*1000);
  }
  else
    refreshEarthquakes();
};

private TimerTask doRefresh = new TimerTask() {
  public void run() {
    refreshEarthquakes();
  }
};

@Override
public void onCreate() {
  updateTimer = new Timer("earthquakeUpdates");
}
```

6. The `EarthquakeService` will now update the earthquake provider each time it is asked to refresh, as well as on an automated schedule (if one is specified). This information is not yet passed back to the Earthquake Activity's ListView or the EathquakeMap Activity.

   To alert those components, and any other applications interested in earthquake data, modify the `EarthquakeService` to broadcast a new Intent whenever a new earthquake is added.

   6.1. Modify the `addNewQuake` method to call a new `announceNewQuake` method.

```
public static final String NEW_EARTHQUAKE_FOUND = "New_Earthquake_Found";

private void addNewQuake(Quake _quake) {
  ContentResolver cr = getContentResolver();
  // Construct a where clause to make sure we don't already have this
  // earthquake in the provider.
  String w = EarthquakeProvider.KEY_DATE +
             " = " + _quake.getDate().getTime();

  // If the earthquake is new, insert it into the provider.
  Cursor c = cr.query(EarthquakeProvider.CONTENT_URI, null, w, null, null);
  if (c.getCount()==0){
    ContentValues values = new ContentValues();

    values.put(EarthquakeProvider.KEY_DATE, _quake.getDate().getTime());
    values.put(EarthquakeProvider.KEY_DETAILS, _quake.getDetails());

    double lat = _quake.getLocation().getLatitude();
    double lng = _quake.getLocation().getLongitude();
    values.put(EarthquakeProvider.KEY_LOCATION_LAT, lat);
    values.put(EarthquakeProvider.KEY_LOCATION_LNG, lng);
    values.put(EarthquakeProvider.KEY_LINK, _quake.getLink());
    values.put(EarthquakeProvider.KEY_MAGNITUDE, _quake.getMagnitude());

    cr.insert(EarthquakeProvider.CONTENT_URI, values);
     announceNewQuake(_quake);
  }
  c.close();
}

private void announceNewQuake(Quake quake) {
}
```

   6.2. Within `announceNewQuake`, broadcast a new Intent whenever a new earthquake is found.

```
private void announceNewQuake(Quake quake) {
  Intent intent = new Intent(NEW_EARTHQUAKE_FOUND);
  intent.putExtra("date", quake.getDate().getTime());
  intent.putExtra("details", quake.getDetails());
  intent.putExtra("longitude", quake.getLocation().getLongitude());
  intent.putExtra("latitude", quake.getLocation().getLatitude());
  intent.putExtra("magnitude", quake.getMagnitude());

  sendBroadcast(intent);
}
```

7. That completes the `EarthquakeService` implementation. You still need to modify the two Activity components to listen for the Service Intent broadcasts and refresh their displays accordingly.

   7.1. Within the Earthquake Activity, create a new internal `EarthquakeReceiver` class that extends `BroadcastReceiver`. Override the `onReceive` method to call `loadFromProviders` to update the earthquake array and refresh the list.

```
public class EarthquakeReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    loadQuakesFromProvider();
  }
}
```

   7.2. Override the `onResume` method to register the new Receiver and update the LiveView contents when the Activity becomes active. Override `onPause` to unregister it when the Activity moves out of the foreground.

```
EarthquakeReceiver receiver;

@Override
public void onResume() {
  IntentFilter filter;
  filter = new IntentFilter(EarthquakeService.NEW_EARTHQUAKE_FOUND);
  receiver = new EarthquakeReceiver();
  registerReceiver(receiver, filter);

  loadQuakesFromProvider();
  super.onResume();
}

@Override
public void onPause() {
  unregisterReceiver(receiver);
  super.onPause();
}
```

   7.3. Do the same for the EarthquakeMap Activity, this time calling `requery` on the result Cursor before invalidating the MapView whenever the Intent is received.

```
EarthquakeReceiver receiver;

@Override
public void onResume() {
  earthquakeCursor.requery();

  IntentFilter filter;
  filter = new IntentFilter(EarthquakeService.NEW_EARTHQUAKE_FOUND);
  receiver = new EarthquakeReceiver();
  registerReceiver(receiver, filter);

  super.onResume();
```

```
    }

    @Override
    public void onPause() {
      earthquakeCursor.deactivate();
      super.onPause();
    }

    public class EarthquakeReceiver extends BroadcastReceiver {
      @Override
      public void onReceive(Context context, Intent intent) {
        earthquakeCursor.requery();
        MapView earthquakeMap = (MapView)findViewById(R.id.map_view);
        earthquakeMap.invalidate();
      }
    }
```

Now when the Earthquake Activity is launched, it will start the Earthquake Service. This Service will then continue to run, updating the earthquake Content Provider in the background, even after the Activity is suspended or closed.

You'll continue to upgrade and enhance the Earthquake Service throughout the chapter, first using Toasts and later Notifications.

At this stage, the earthquake processing is done in a Service, but it's still being executed on the main thread. Later in this chapter, you'll learn how to move time-consuming operations onto background threads to improve performance and avoid "Application Unresponsive" messages.

Binding Activities to Services

When an Activity is bound to a Service, it maintains a reference to the Service instance itself, allowing you to make method calls on the running Service as you would any other instantiated class.

Binding is available for Activities that would benefit from a more detailed interface with a Service. To support binding for a Service, implement the `onBind` method as shown in the simple example below:

```
private final IBinder binder = new MyBinder();

@Override
public IBinder onBind(Intent intent) {
  return binder;
}

public class MyBinder extends Binder {
  MyService getService() {
    return MyService.this;
  }
}
```

The connection between the Service and Activity is represented as a `ServiceConnection`. You'll need to implement a new `ServiceConnection`, overriding the `onServiceConnected`

and `onServiceDisconnected` methods to get a reference to the Service instance once a connection has been established.

```
// Reference to the service
private MyService serviceBinder;

// Handles the connection between the service and activity
private ServiceConnection mConnection = new ServiceConnection() {

  public void onServiceConnected(ComponentName className, IBinder service) {
    // Called when the connection is made.
    serviceBinder = ((MyService.MyBinder)service).getService();
  }

  public void onServiceDisconnected(ComponentName className) {
    // Received when the service unexpectedly disconnects.
    serviceBinder = null;
  }
};
```

To perform the binding, call `bindService`, passing in an Intent (either explicit or implicit) that selects the Service to bind to and an instance of your new `ServiceConnection` implementation, as shown in this skeleton code:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  // Bind to the service
  Intent bindIntent = new Intent(MyActivity.this, MyService.class);
  bindService(bindIntent, mConnection, Context.BIND_AUTO_CREATE);
}
```

Once the Service has been bound, all of its public methods and properties are available through the `serviceBinder` object obtained from the `onServiceConnected` handler.

Android applications do not (normally) share memory, but in some cases, your application may want to interact with (and bind to) Services running in different application processes.

You can communicate with a Service running in a different process using broadcast Intents or through the extras Bundle in the Intent used to start the Service. If you need a more tightly coupled connection, you can make a Service available for binding across application boundaries using AIDL. AIDL defines the Service's interface in terms of OS level primitives, allowing Android to transmit objects across process boundaries. AIDL definitions are covered in Chapter 11.

Using Background Worker Threads

To ensure that your applications remain responsive, it's good practice to move all slow, time-consuming operations off the main application thread and onto a child thread.

All Android application components — including Activities, Services, and Broadcast Receivers — run on the main application thread. As a result, time-consuming processing in any component will block all other components including Services and the visible Activity.

Using background threads is vital to avoid the "Application Unresponsive" Dialog box described in Chapter 2. Unresponsiveness is defined in Android as Activities that don't respond to an input event (such as a key press) within 5 seconds and Broadcast Receivers that don't complete their `onReceive` handlers within 10 seconds.

Not only do you want to avoid this scenario, you don't want to even get close. Use background threads for all time-consuming processing, including file operations, network lookups, database transactions, and complex calculations.

Creating New Threads

You can create and manage child threads using Android's `Handler` class and the threading classes available within `java.lang.Thread`. The following skeleton code shows how to move processing onto a child thread:

```
// This method is called on the main GUI thread.
private void mainProcessing() {
   // This moves the time consuming operation to a child thread.
  Thread thread = new Thread(null, doBackgroundThreadProcessing, "Background");
  thread.start();
}

// Runnable that executes the background processing method.
private Runnable doBackgroundThreadProcessing = new Runnable() {
  public void run() {
    backgroundThreadProcessing();
  }
};

// Method which does some processing in the background.
private void backgroundThreadProcessing() {
  [ ... Time consuming operations ... ]
}
```

Synchronizing Threads for GUI Operations

Whenever you're using background threads in a GUI environment, it's important to synchronize child threads with the main application (GUI) thread before creating or modifying graphical components.

The `Handler` class allows you to post methods onto the thread in which the Handler was created. Using the `Handler` class, you can post updates to the User Interface from a background thread using the `Post` method. The following example shows the outline for using the Handler to update the GUI thread:

```
// Initialize a handler on the main thread.
private Handler handler = new Handler();

private void mainProcessing() {
  Thread thread = new Thread(null, doBackgroundThreadProcessing, "Background");
```

```
    thread.start();
  }
  private Runnable doBackgroundThreadProcessing = new Runnable() {
    public void run() {
      backgroundThreadProcessing();
    }
  };

  // Method which does some processing in the background.
  private void backgroundThreadProcessing() {
    [ ... Time consuming operations ... ]
    handler.post(doUpdateGUI);
  }

  // Runnable that executes the update GUI method.
  private Runnable doUpdateGUI = new Runnable() {
    public void run() {
      updateGUI();
    }
  };

  private void updateGUI() {
    [ ... Open a dialog or modify a GUI element ... ]
  }
```

The `Handler` class lets you delay posts or execute them at a specific time, using the `postDelayed` and `postAtTime` methods, respectively.

In the specific case of actions that modify Views, the `UIThreadUtilities` class provides the `runOnUIThread` method, which lets you force a method to execute on the same thread as the specified View, Activity, or Dialog.

Within your application components, Notifications and Intents are always received and handled on the GUI thread. In all other cases, operations that explicitly interact with objects created on the GUI thread (such as Views) or that display messages (like Toasts) must be invoked on the main thread.

Moving the Earthquake Service to a Background Thread

The following example shows how to move the network lookup and XML processing done in the `EarthquakeService` onto a background thread:

1. Rename the `refreshEarthquakes` method to `doRefreshEarthquakes`.

```
  private void doRefreshEarthquakes() {
    [ ... previous refreshEarthquakes method ... ]
  }
```

2. Create a new `refreshEarthquakes` method. It should start a background thread that executes the newly named `doRefreshEarthquakes` method.

```
  private void refreshEarthquakes() {
    Thread updateThread = new Thread(null, backgroundRefresh,
                                     "refresh_earthquake");
    updateThread.start();
```

```
}

private Runnable backgroundRefresh = new Runnable() {
  public void run() {
      doRefreshEarthquakes();
  }
};
```

Let's Make a Toast

Toasts are transient Dialog boxes that remain visible for only a few seconds before fading out. Toasts don't steal focus and are non-modal, so they don't interrupt the active application.

Toasts are perfect for informing your users of events without forcing them to open an Activity or read a Notification. They provide an ideal mechanism for alerting users to events occurring in background Services without interrupting foreground applications.

The `Toast` class includes a static `makeText` method that creates a standard Toast display window. Pass the application Context, the text message to display, and the length of time to display it (`LENGTH_SHORT` or `LENGTH_LONG`) in to the `makeText` method to construct a new Toast. Once a Toast has been created, display it by calling `show`, as shown in the following snippet:

```
Context context = getApplicationContext();
String msg = "To health and happiness!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, msg, duration);
toast.show();
```

Figure 8-1 shows a Toast. It will remain on screen for around 2 seconds before fading out. The application behind it remains fully responsive and interactive while the Toast is visible.



Figure 8-1

Customizing Toasts

The standard Toast text message window is often sufficient, but in many situations you'll want to customize its appearance and screen position. You can modify a Toast by setting its display position and assigning it alternative Views or layouts.

The following snippet shows how to align a Toast to the bottom of the screen using the `setGravity` method:

```
Context context = getApplicationContext();
String msg = "To the bride an groom!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, msg, duration);
int offsetX = 0;
int offsetY = 0;
toast.setGravity(Gravity.BOTTOM, offsetX, offsetY);
toast.show();
```

When a text message just isn't going to get the job done, you can specify a custom View or layout to use a more complex, or more visual, display. Using `setView` on a Toast object, you can specify any View (including layouts) to display using the transient message window mechanism.

For example, the following snippet assigns a layout, containing the `CompassView` widget from Chapter 4 along with a `TextView`, to be displayed as a Toast.

```
Context context = getApplicationContext();
String msg = "Cheers!";
int duration = Toast.LENGTH_LONG;
Toast toast = Toast.makeText(context, msg, duration);
toast.setGravity(Gravity.TOP, 0, 0);

LinearLayout ll = new LinearLayout(context);
ll.setOrientation(LinearLayout.VERTICAL);

TextView myTextView = new TextView(context);
CompassView cv = new CompassView(context);

myTextView.setText(msg);

int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;

ll.addView(cv, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));

ll.setPadding(40, 50, 0, 50);

toast.setView(ll);
toast.show();
```

The resulting Toast will appear as shown in Figure 8-2.



Figure 8-2

Using Toasts in Worker Threads

As GUI components, Toasts must be opened on the GUI thread or risk throwing a cross thread exception. In the following example, a Handler is used to ensure that the Toast is opened on the GUI thread:

```
private void mainProcessing() {
  Thread thread = new Thread(null, doBackgroundThreadProcessing, "Background");
  thread.start();
}

private Runnable doBackgroundThreadProcessing = new Runnable() {
  public void run() {
    backgroundThreadProcessing();
  }
};

private void backgroundThreadProcessing() {
  handler.post(doUpdateGUI);
}

// Runnable that executes the update GUI method.
private Runnable doUpdateGUI = new Runnable() {
  public void run() {
    Context context = getApplicationContext();
    String msg = "To open mobile development!";
    int duration = Toast.LENGTH_SHORT;
    Toast.makeText(context, msg, duration).show();
  }
};
```

Introducing Notifications

Notifications are a way for your applications to alert users, without using an Activity. Notifications are handled by the Notification Manger, and currently include the ability to:

- Create a new status bar icon.
- Display additional information (and launch an Intent) in the extended status bar window.
- Flash the lights/LEDs.
- Vibrate the phone.
- Sound audible alerts (ringtones, media store sounds).

Notifications are the preferred way for invisible application components (Broadcast Receivers, Services, and inactive Activities) to alert users that events have occurred that require attention.

As a User Interface metaphor, Notifications are particularly well suited to mobile devices. It's likely that your users will have their phones with them at all times but quite unlikely that they will be paying attention to them, or your application, at any given time. Generally, users will have several applications open in the background, and they won't be paying attention to any of them.

In this environment, it's important that your applications be able to alert users when specific events occur that require their attention.

Notifications can be persisted through insistent repetition, or (more commonly) by using an icon on the status bar. Status bar icons can be updated regularly or expanded to show additional information using the expanded status bar window shown in Figure 8-3.

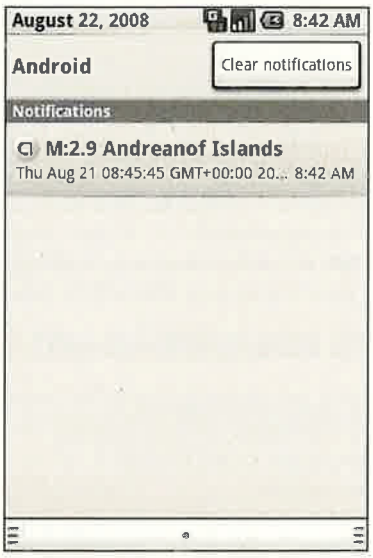


Figure 8-3

To display the expanded status bar view, click a status bar icon and drag it toward the bottom of the screen. To "lock" it in place, ensure that you release your drag only after the window covers the entire screen. To hide it, simply drag it back upward.

Introducing the Notification Manager

The *Notification Manager* is a system Service used to handle Notifications. Get a reference to it using the `getSystemService` method, as shown in the snippet below:

```
String svcName = Context.NOTIFICATION_SERVICE;

NotificationManager notificationManager;
notificationManager = (NotificationManager)getSystemService(svcName);
```

Using the Notification Manager, you can trigger new Notifications, modify existing ones, or remove those that are no longer needed or wanted.

Creating Notifications

Creating and configuring a new Notification is done in three parts.

Firstly, you create a new Notification object, passing in the icon to display in the status bar, along with the status bar ticker-text, and the time of this Notification, as shown in the following code snippet:

```
// Choose a drawable to display as the status bar icon
int icon = R.drawable.icon;
// Text to display in the status bar when the notification is launched
String tickerText = "Notification";
// The extended status bar orders notification in time order
long when = System.currentTimeMillis();

Notification notification = new Notification(icon, tickerText, when);
```

The ticker-text will scroll along the status bar when the Notification is fired.

Secondly, configure the appearance of the Notification within the extended status window using the `setLatestEventInfo` method. This extended status window displays the icon and time defined in the constructor and also shows a title and a details string. Notifications often represent a request for action or attention, so you can specify a `PendingIntent` that will be fired if a user clicks the Notification item.

The code snippet below uses `setLatestEventInfo` to set these values:

```
Context context = getApplicationContext();
// Text to display in the extended status window
String expandedText = "Extended status text";
// Title for the expanded status
String expandedTitle = "Notification Title";
// Intent to launch an activity when the extended text is clicked
Intent intent = new Intent(this, MyActivity.class);
PendingIntent launchIntent = PendingIntent.getActivity(context, 0, intent, 0);

notification.setLatestEventInfo(context,
```

```
                                    expandedTitle,
                                    expandedText,
                                    launchIntent);
```

It's good form to use one Notification icon to represent multiple instances of the same event (e.g., receiving multiple SMS messages). To demonstrate this to users, update the values set by `setLatestEventInfo` to reflect the most recent message and re-trigger the Notification to update its values.

You can also use the `number` property to display the number of events a status bar icon represents. Setting this value greater than 1, as shown below, overlays the values as a small number over the status bar icon:

```
notification.number++;
```

As with all changes to a Notification, you will need to re-trigger it to apply the change. To remove the overlay, set the `number` value to 0 or -1.

Finally, you can enhance Notifications using various properties on the Notification object to flash the device LEDs, vibrate the phone, and play audio files. These advanced features are detailed later in this chapter.

Triggering Notifications

To fire a Notification, pass it in to the `notify` method on the `NotificationManager` along with an integer reference ID, as shown in the following snippet:

```
int notificationRef = 1;
notificationManager.notify(notificationRef, notification);
```

To update a Notification that's already been fired, re-trigger, passing the same reference ID. You can pass in either the same Notification object or an entirely new one. As long as the ID values are the same, the new Notification will be used to replace the status icon and extended status window details.

You also use the reference ID to cancel Notifications by calling the `cancel` method on the Notification Manager, as shown below:

```
notificationManager.cancel(notificationRef);
```

Canceling a Notification removes its status bar icon and clears it from the extended status window.

Adding Notifications to the Earthquake Monitor

In the following example, the `EarthquakeService` is enhanced to trigger a Notification for each new earthquake. As well as displaying a status bar icon, the expanded Notification view will display the magnitude and location of the latest quake, and selecting it will open the Earthquake Activity.

1. Within the `EarthquakeService`, start by creating a new `Notification` instance variable to store the Notification object used to control the status bar icon and extended status window item details.

```
private Notification newEarthquakeNotification;
public static final int NOTIFICATION_ID = 1;
```

2. Extend the `onCreate` method to create this Notification object.

```
@Override
public void onCreate() {
  updateTimer = new Timer("earthquakeUpdates");

  int icon = R.drawable.icon;
  String tickerText = "New Earthquake Detected";
  long when = System.currentTimeMillis();

  newEarthquakeNotification= new Notification(icon, tickerText, when);
}
```

3. Now extend the `announceNewQuake` method to trigger the Notification after each new earthquake is added to the Content Provider. Before initiating the Notification, update the extended details using `setLatestEventInfo`.

```
private void announceNewQuake(Quake quake) {
  String svcName = Context.NOTIFICATION_SERVICE;
  NotificationManager notificationManager;
  notificationManager = (NotificationManager)getSystemService(svcName);

  Context context = getApplicationContext();
  String expandedText = quake.getDate().toString();
  String expandedTitle = "M:" + quake.getMagnitude() + " " +
                         quake.getDetails();
  Intent startActivityIntent = new Intent(this, Earthquake.class);
  PendingIntent launchIntent = PendingIntent.getActivity(context,
                                                         0,
                                                         startActivityIntent,
                                                         0);

  newEarthquakeNotification.setLatestEventInfo(context,
                                               expandedTitle,
                                               expandedText,
                                               launchIntent);
  newEarthquakeNotification.when = java.lang.System.currentTimeMillis();

  notificationManager.notify(NOTIFICATION_ID, newEarthquakeNotification);

  Intent intent = new Intent(NEW_EARTHQUAKE_FOUND);
  intent.putExtra("date", quake.getDate().getTime());
  intent.putExtra("details", quake.getDetails());
  intent.putExtra("longitude", quake.getLocation().getLongitude());
  intent.putExtra("latitude", quake.getLocation().getLatitude());
  intent.putExtra("magnitude", quake.getMagnitude());

  sendBroadcast(intent);
}
```

4. The final step is to clear and disable Notifications within the two Activity classes. This is done to dismiss the status icon when the application is active.

4.1. Starting with the Earthquake Activity, modify the `onCreate` method to get a reference to the Notification Manager.

```
NotificationManager notificationManager;

@Override
public void onCreate(Bundle icicle) {
  [ ... existing onCreate ... ]

  String svcName = Context.NOTIFICATION_SERVICE;
  notificationManager = (NotificationManager)getSystemService(svcName);
}
```

4.2. Modify the `onReceive` method of the `EarthquakeReceiver`. As this is only registered (so it will only execute) when the Activity is active, you can safely cancel all Notification earthquake Notifications here as soon as they're triggered.

```
@Override
public void onReceive(Context context, Intent intent) {
  loadQuakesFromProvider();

  notificationManager.cancel(EarthquakeService.NOTIFICATION_ID);
}
```

4.3. Next, extend the `onResume` method to cancel the Notification when the Activity becomes active.

```
@Override
public void onResume() {
  notificationManager.cancel(EarthquakeService.NOTIFICATION_ID);

  IntentFilter filter;
  filter = new IntentFilter(EarthquakeService.NEW_EARTHQUAKE_FOUND);
  receiver = new EarthquakeReceiver();
  registerReceiver(receiver, filter);
  super.onResume();
}
```

4.4. Repeat the same process with the `EarthquakeMap` Activity.

```
NotificationManager notificationManager;

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.earthquake_map);

  ContentResolver cr = getContentResolver();
  earthquakeCursor = cr.query(EarthquakeProvider.CONTENT_URI,
                              null, null, null, null);

  MapView earthquakeMap = (MapView)findViewById(R.id.map_view);
  earthquakeMap.getOverlays().add(new EarthquakeOverlay(earthquakeCursor));

  String svcName = Context.NOTIFICATION_SERVICE;
  notificationManager = (NotificationManager)getSystemService(svcName);
```

```
    }

    @Override
    public void onResume() {
      notificationManager.cancel(EarthquakeService.NOTIFICATION_ID);

      earthquakeCursor.requery();

      IntentFilter filter;
      filter = new IntentFilter(EarthquakeService.NEW_EARTHQUAKE_FOUND);
      receiver = new EarthquakeReceiver();
      registerReceiver(receiver, filter);

      super.onResume();
    }

    public class EarthquakeReceiver extends BroadcastReceiver {
      @Override
      public void onReceive(Context context, Intent intent) {
        notificationManager.cancel(EarthquakeService.NOTIFICATION_ID);

        earthquakeCursor.requery();
        MapView earthquakeMap = (MapView)findViewById(R.id.map_view);
        earthquakeMap.invalidate();
      }
    }
```

Advanced Notification Techniques

In the following sections, you'll learn to enhance Notifications to provide additional alerting through hardware, in particular, by making the device ring, flash, and vibrate.

As each enhancement is described, you will be provided with a code snippet that can be added to the Earthquake example to provide user feedback on the severity of each earthquake as it's detected.

> *To use the Notification techniques described here without also displaying the status bar icon, simply cancel the Notification directly after triggering it. This stops the icon from displaying but doesn't interrupt the other effects.*

Making Sounds

Using an audio alert to notify the user of a device event (like incoming calls) is a technique that predates the mobile, and has stood the test of time. Most native phone events from incoming calls to new messages and low battery are announced by an audible ringtone.

Android lets you play any audio file on the phone as a Notification by assigning a location URI to the `sound` property, as shown in the snippet below:

```
notification.sound = ringURI;
```

To use your own custom audio, push the file onto your device, or include it as a raw resource, as described in Chapter 6.

The following snippet can be added to the `announceNewQuake` method within the Earthquake Service from the earlier example. It adds an audio component to the earthquake Notification, ringing one of the default phone ringtones if a significant earthquake (one with magnitude greater than 6) occurs.

```
if (quake.getMagnitude() > 6) {
  Uri ringURI = Uri.fromFile(new File("/system/media/audio/ringtones/ringer.mp3"));
  newEarthquakeNotification.sound = ringURI;
}
```

Vibrating the Phone

You can use the phone's vibration function to execute a vibration pattern specific to your Notification. Android lets you control the pattern of a vibration; you can use vibration to convey information as well as get the user's attention.

To set a vibration pattern, assign an array of `longs` to the Notification's `vibrate` property. Construct the array so that every alternate number is the length of time (in milliseconds) to vibrate or pause, respectively.

Before you can use vibration in your application, you need to be granted permission. Add a `uses-permission` to your application to request access to the device vibration using the following code snippet:

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

The following example shows how to modify a Notification to vibrate in a repeating pattern of 1 second on, 1 second off, for 5 seconds total.

```
long[] vibrate = new long[] { 1000, 1000, 1000, 1000, 1000 };
notification.vibrate = vibrate;
```

You can take advantage of this fine-grained control to pass information to your users. In the following update to the `announceNewQuake` method, the phone is set to vibrate in a pattern based on the power of the quake. Earthquakes are measured on an exponential scale, so you'll use the same scale when creating the vibration pattern.

For a barely perceptible magnitude 1 quake, the phone will vibrate for a fraction of a second; but for magnitude 10, an earthquake that would split the earth in two, your users will have a head start on the Apocalypse when their devices vibrate for a full 20 seconds. Most significant quakes fall between 3 and 7 on the Richter scale, so the more likely scenario is a more reasonable 200-millisecond to 4-second vibration duration range.

```
double vibrateLength = 100*Math.exp(0.53*quake.getMagnitude());
long[] vibrate = new long[] {100, 100, (long)vibrateLength };
newEarthquakeNotification.vibrate = vibrate;
```

The current Android Emulator does not visually or audibly indicate that the device is vibrating. To confirm that your Notification is behaving appropriately, you can monitor the log for "Vibration On"/"Vibration Off."

Flashing the Lights

Notifications also include properties to configure the color and flash frequency of the device's LED.

The `ledARGB` property can be used to set the LED's color, while the `ledOffMS` and `ledOnMS` properties let you set the frequency and pattern of the flashing LED. You can turn the LED on by setting the `ledOnMS` property to 1 and the `ledOffMS` property to 0, or turn it off by setting both properties to 0.

Once you have configured the LED settings, you must also add the `FLAG_SHOW_LIGHTS` flag to the Notification's `flags` property.

The following code snippet shows how to turn on the red device LED:

```
notification.ledARGB = Color.RED;
notification.ledOffMS = 0;
notification.ledOnMS = 1;
notification.flags = notification.flags | Notification.FLAG_SHOW_LIGHTS;
```

Controlling the color and flash frequency is another opportunity to pass additional information to users.

In the Earthquake monitoring example, you can help your users perceive the nuances of an exponential scale by also using the device's LED to help convey the magnitude. In the snippet below, the color of the LED depends on the size of the quake, and the frequency of the flashing is inversely related to the power of the quake:

```
int color;
if (quake.getMagnitude() < 5.4)
  color = Color.GREEN;
else if (quake.getMagnitude() < 6)
  color = Color.YELLOW;
else
  color = Color.RED;

newEarthquakeNotification.ledARGB = color;
newEarthquakeNotification.ledOffMS = (int)vibrateLength;
newEarthquakeNotification.ledOnMS = (int)vibrateLength;
newEarthquakeNotification.flags = newEarthquakeNotification.flags |
                                  Notification.FLAG_SHOW_LIGHTS;
```

The current Android Emulator does not visually illustrate the LEDs. This makes it quite difficult to confirm that your LEDs are flashing correctly. In hardware, each device may have different limitations in regard to setting the color of the LED. In such cases, as close an approximation as possible will be used.

Ongoing and Insistent Notifications

Notifications can be configured as ongoing and/or insistent by setting the `FLAG_INSISTENT` and `FLAG_ONGOING_EVENT` flags.

Notifications flagged as ongoing, as in the snippet below, are used to represent events that are currently in progress (such as an incoming call). Ongoing events are separated from "normal" Notifications within the extended status bar window.

```
notification.flags = notification.flags | Notification.FLAG_ONGOING_EVENT;
```

Insistent Notifications repeat continuously until canceled. The code snippet below shows how to set a Notification as insistent:

```
notification.flags = notification.flags | Notification.FLAG_INSISTENT;
```

Insistent Notifications are handled by continuously repeating the initial Notification effects until the Notification is canceled. Insistent Notifications should be reserved for situations like Alarms, where timely and immediate response is required.

Using Alarms

Alarms are an application independent way of firing Intents at predetermined times.

Alarms are set outside the scope of your applications, so they can be used to trigger application events or actions even after your application has been closed. They can be particularly powerful in combination with Broadcast Receivers, allowing you to set Alarms that launch applications or perform actions without applications needing to be open and active until they're required.

For example, you can use Alarms to implement an alarm clock application, perform regular network lookups, or schedule time-consuming or cost-bound operations at "off peak" times.

> *For timing operations that occur only during the lifetime of your applications, the* `Handler` *class in combination with Timers and Threads is a better approach as it allows Android better control over system resources.*

Alarms in Android remain active while the device is in sleep mode and can optionally be set to wake the device; however, all Alarms are canceled whenever the device is rebooted.

Alarm operations are handled through the `AlarmManager`, a system Service accessed via `getSystemService` as shown below:

```
AlarmManager alarms = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
```

To create a new Alarm, use the `set` method and specify an alarm type, trigger time, and a Pending Intent to fire when the Alarm triggers. If the Alarm you set occurs in the past, it will be triggered immediately.

There are four alarm types available. Your selection will determine if the time value passed in the `set` method represents a specific time or an elapsed wait:

- **RTC_WAKEUP** Wakes up the device to fire the Intent at the clock time specified when setting the Alarm.
- **RTC** Will fire the Intent at an explicit time, but will not wake the device.
- **ELAPSED_REALTIME** The Intent will be fired based on the amount of time elapsed since the device was booted, but will not wake the device. The elapsed time includes any period of time the device was asleep. Note that the time elapsed is since it was last booted.
- **ELAPSED_REALTIME_WAKEUP** Will wake up the device if necessary and fire the Intent after a specified length of time has passed since the device was booted.

The Alarm creation process is demonstrated in the snippet below:

```
int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;
long timeOrLengthofWait = 10000;
String ALARM_ACTION = "ALARM_ACTION";
Intent intentToFire = new Intent(ALARM_ACTION);
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intentToFire, 0);

alarms.set(alarmType, timeOrLengthofWait, pendingIntent);
```

When the Alarm goes off, the Pending Intent you specified will be fired. Setting a second Alarm using the same Pending Intent replaces the preexisting Alarm.

To cancel an Alarm, call `cancel` on the Alarm Manager, passing in the Pending Intent you no longer wish to trigger, as shown in the snippet below:

```
alarms.cancel(pendingIntent);
```

In the following code snippet, two Alarms are set and the first one is subsequently canceled. The first is explicitly set to a specific time and will wake up the device in order to fire. The second is set for 30 minutes of time elapsed since the device was started, but will not wake the device if it's sleeping.

```
AlarmManager alarms = (AlarmManager)getSystemService(Context.ALARM_SERVICE);

String MY_RTC_ALARM = "MY_RTC_ALARM";
String ALARM_ACTION = "MY_ELAPSED_ALARM";

PendingIntent rtcIntent = PendingIntent.getBroadcast(this, 0,
                                                     new Intent(MY_RTC_ALARM),
                                                     1);
PendingIntent elapsedIntent = PendingIntent.getBroadcast(this, 0,
                                                         new Intent(ALARM_ACTION),
                                                         1);

// Wakeup and fire intent in 5 hours.
Date t = new Date();
t.setTime(java.lang.System.currentTimeMillis() + 60*1000*5);
alarms.set(AlarmManager.RTC_WAKEUP, t.getTime(), rtcIntent);

// Fire intent in 30 mins if already awake.
alarms.set(AlarmManager.ELAPSED_REALTIME, 30*60*1000, elapsedIntent);

// Cancel the first alarm.
alarms.cancel(rtcIntent);
```

Using Alarms to Update Earthquakes

In this final modification to the Earthquake example, you'll use Alarms to replace the Timer currently used to schedule Earthquake network refreshes.

1. Start by creating a new `EarthquakeAlarmReceiver` class that extends `BroadcastReceiver`.

```
package com.paad.earthquake;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class EarthquakeAlarmReceiver extends BroadcastReceiver {

}
```

2. Override the `onReceive` method to explicitly start the `EarthquakeService`.

```
@Override
public void onReceive(Context context, Intent intent) {
  Intent startIntent = new Intent(context, EarthquakeService.class);
  context.startService(startIntent);
}
```

3. Create a new public static String to define the action that will be used to trigger the Broadcast Receiver.

```
public static final String ACTION_REFRESH_EARTHQUAKE_ALARM =
                        "com.paad.earthquake.ACTION_REFRESH_EARTHQUAKE_ALARM";
```

4. Add the new `EarthquakeAlarmReceiver` to the manifest, including an `intent-filter` tag that listens for the action defined in Step 3.

```
<receiver android:name=".EarthquakeAlarmReceiver">
  <intent-filter>
    <action
      android:name="com.paad.earthquake.ACTION_REFRESH_EARTHQUAKE_ALARM"
    />
  </intent-filter>
</receiver>
```

5. Within the EarthquakeService, update the `onCreate` method to get a reference to the `AlarmManager`, and create a new `PendingIntent` that will be fired when the Alarm goes off. You can also remove the `timerTask` initialization.

```
AlarmManager alarms;
PendingIntent alarmIntent;

@Override
public void onCreate() {
  int icon = R.drawable.icon;
  String tickerText = "New Earthquake Detected";
  long when = System.currentTimeMillis();

  newEarthquakeNotification = new Notification(icon, tickerText, when);

  alarms = (AlarmManager)getSystemService(Context.ALARM_SERVICE);

  String ALARM_ACTION;
  ALARM_ACTION = EarthquakeAlarmReceiver.ACTION_REFRESH_EARTHQUAKE_ALARM;
  Intent intentToFire = new Intent(ALARM_ACTION);
```

```
    alarmIntent = PendingIntent.getBroadcast(this, 0, intentToFire, 0);
  }
```

6. Modify the `onStart` method to set an Alarm rather than use a Timer to schedule the next refresh (if automated updates are enabled). Setting a new Intent with the same action will automatically cancel the previous Alarm.

```
@Override
public void onStart(Intent intent, int startId) {
  SharedPreferences prefs = getSharedPreferences(Preferences.USER_PREFERENCE,
                                                 Activity.MODE_PRIVATE);

  int minMagIndex = prefs.getInt(Preferences.PREF_MIN_MAG, 0);
  if (minMagIndex < 0)
    minMagIndex = 0;

  int freqIndex = prefs.getInt(Preferences.PREF_UPDATE_FREQ, 0);
  if (freqIndex < 0)
    freqIndex = 0;

  boolean autoUpdate = prefs.getBoolean(Preferences.PREF_AUTO_UPDATE, false);

  Resources r = getResources();
  int[] minMagValues = r.getIntArray(R.array.magnitude);
  int[] freqValues = r.getIntArray(R.array.update_freq_values);

  minimumMagnitude = minMagValues[minMagIndex];
  int updateFreq = freqValues[freqIndex];

  if (autoUpdate) {
    int alarmType = AlarmManager.ELAPSED_REALTIME_WAKEUP;
    long timeToRefresh = SystemClock.elapsedRealtime() + updateFreq*60*1000;
    alarms.set(alarmType, timeToRefresh, alarmIntent);
  }
  else
    alarms.cancel(alarmIntent);

  refreshEarthquakes();
};
```

7. You can now remove the `updateTimer` instance variable and the TimerTask instance `doRefresh`.

Summary

Services are one of the most compelling reasons to develop applications on the Android platform. In this chapter, you learned how to use these invisible application components to perform processing while your applications are hidden in the background.

You were introduced to Toasts, a transient message box that lets you display information to users without stealing focus or interrupting their workflow.

You used the Notification Manager to send alerts to your users from within Services and Activities using customized LEDs, vibration patterns, and audio files to convey detailed event information.

Using Alarms, you were able to preset events and actions on the device using Intents to broadcast actions or start Activities or Services.

This chapter also demonstrated how to:

- Bind a Service to an Activity to make use of a more detailed, structured interface.
- Ensure that your applications remain responsive by moving time-consuming processing like network lookups onto worker threads.
- Use handlers to synchronize child threads with the main application GUI when performing operations using visual controls and Toasts.
- Create insistent and ongoing Notifications.

In Chapter 9, you'll be introduced to the communications features of Android. Starting with a look at the GTalk Service, you'll learn how to send and receive text and data messages to transmit data between devices. You'll then investigate the SMS functionality available for you to send and receive SMS text and data messages.

9

Peer-to-Peer Communication

In this chapter, you'll learn to use Android's peer-to-peer (P2P) text and data communication protocols, specifically, instant messaging and SMS (short messaging service). Using these technologies, you can create applications that can communicate between devices, including multiplayer games and collaborative mobile social applications.

When this chapter was originally written, the Android SDK included a comprehensive instant messaging (IM) service (powered by GTalk) that offered access to the instant messaging framework. This included the ability to send and receive text messages, set user status through presence, and determine the presence of IM contacts. Unfortunately, owing to security concerns the IM API has since been removed, though it's expected that later releases of Android will expose developer access to an IM framework. This chapter will show how earlier releases of Android allowed this technology to be used for sending text IM messages and as a mechanism for broadcasting Intents to remote Android devices — a mechanism that allowed you to create applications that interact between devices in real time.

Android still offers full access to SMS functionality, letting you send and receive SMS text messages within your applications. Using the Android APIs, you can create your own SMS client application to replace the native applications available as part of the software stack. Alternatively, you can incorporate the messaging functionality within your own applications.

At the end of this chapter, you'll use the SMS Manager in a detailed project that shows how to create an emergency SMS responder. In emergency situations, it will let users quickly, or automatically, respond to people asking after their safety.

Introducing Android Instant Messaging

Largely as a result of security concerns, developer access to the GTalk IM Service has been restricted for Android SDK version 1.0. As a result, the functionality described in this section will not be available to developers using the first full release of the Android SDK.

Rather than remove the affected sections, they have been left here in full as a guide for use with future Android releases.

Later releases of Android will expose a full suite of instant messaging functionality through an XMPP based IM Service. This will include management of contact rosters, presence notification, and the transmission and receipt of instant messages.

> *Google Talk (GTalk) is an instant messaging protocol for peer-to-peer (P2P) communication. Once connected, GTalk maintains a persistent socket connection with the GTalk server, meaning fast response times and low latency.*

This section is based on an early SDK implementation that used GTalk. GTalk is based on the XMPP protocol, but it's a Google-specific variant that currently requires that users have a Gmail account.

What makes the GTalk Service particularly interesting for developers is the ability to broadcast Intents over the air (OTA) between Android devices using data messaging. Data messages received by a remote device are re-broadcast as Intents locally, meaning that this mechanism lets you broadcast an Intent on a remote device.

The GTalk Service can be used to create your own multi-user, social, or collaborative applications. It provides the framework for building a range of applications, including distributed emergency warning systems, dynamic route guidance applications, family social networks, and augmented reality gaming systems.

Android will eventually include all the interfaces needed to create a Google Talk Instant Messaging client, including full control over presence management and subscription handling. You can, if you're so inclined, build a replacement for the native client — or simply use the relevant components within your own applications.

Using the GTalk Service

Before you can access the GTalk Service, you need to import the `gtalkservice` library into your application with a `uses-library` tag inside the application node of the project manifest, as shown below:

```
<uses-library android:name="com.google.android.gtalkservice"/>
```

You also need to add the GTalk `uses-permission` tag, as shown in this XML snippet:

```
<uses-permission android:name="android.permission.GTALK"/>
```

Android Instant Messaging functionality is exposed through various interfaces as described below:

- `IGTalkService` Is used to create, access, and manage GTalk connections.
- `IGTalkConnection` A GTalk Connection represents a persistent socket connection between the device and the server it's connecting to. The GTalk Service creates a default connection upon start-up that you can access by calling `getDefaultConnection` on the GTalk Service object.
- `IImSession` Most instant messaging functionality is handled through the `IImSession` interface. It's used to retrieve the IM roster, set the user presence, obtain the presence of contacts, and manage chat sessions. Each GTalk Connection creates a default session, available through the `getDefaultSession` method.
- `IChatSession` All instant messaging chats are handled through the `IChatSession` interface. New Chat Sessions are created by initiating new chats, or joining existing ones, from an IM Session object. Using the Chat Session interface, you can send new chat messages, invite new participants to a group chat, and return a list of people involved in a chat.
- `IChatListener` Implement `IChatListener` to listen for messages in an IM Session or Chat Session. The `IChatListener` interface handlers listen for incoming messages, new chat participants, and people leaving a chat.
- `IGroupChatInvitationListener` Implement `IGroupChatInvitationListener` to listen for invitations to join group chats. The `onInvitationReceived` handler is passed a `GroupChatInvitation` that includes the username of the inviter, the room address, a "reason" (usually the room description), and the password you need in order to join the group chat.
- `IRosterListener` You can monitor your IM contacts roster, and the presence of the people on it, by implementing the `IRosterListener` interface. The Roster Listener includes event handlers that are fired when there are changes in a contact's presence as well as upon the addition and removal of contacts from the roster.

Binding to the GTalk Service

To use the GTalk Service, it must be bound to your application component using `bindService`.

The `bindService` method accepts two input parameters, an Intent, which specifies a component to bind to, and a `ServiceConnection` implementation. The following skeleton code demonstrates the pattern used to bind to the GTalk service:

```
IGTalkService gtalkService;

private void bindGTalk() {
  Intent i = new Intent();

  i.setComponent(GTalkServiceConstants.GTALK_SERVICE_COMPONENT);
  bindService(i, gTalkConnection, 0);
}

private ServiceConnection gTalkConnection = new ServiceConnection() {
```

```
    // When the service connects, get the default GTalk Session
    public void onServiceConnected(ComponentName className, IBinder service) {
      gtalkService = IGTalkService.Stub.asInterface(service);
    }

    // If the service disconnects
    public void onServiceDisconnected(ComponentName className) {
      gtalkService = null;
    }
  };
```

A bound GTalk Service represents a connection between your application and the GTalk Service APIs. Before you can use the Service to use Android's Instant Messaging functionality, you need to initiate a new `GTalkConnection`, as shown in the following section.

Making a GTalk Connection and Starting an IM Session

A GTalk Connection represents a conduit between the device and a GTalk server. An IM Session is the message pathway used to handle all the instant message traffic; all the instant messages for a given session flow through this pipe.

You can create several different connections and multiple IM Sessions connecting to different GTalk servers or IM providers.

Under normal circumstances, a device needs a single GTalk Connection supporting a single IM Session that uses the device owner's username. You can access the default connection and session using `getDefaultConnection` and `getDefaultSession` on the GTalk Service and default connection, respectively, as shown in the snippet below:

```
IGTalkConnection gTalkConnection = gtalkService.getDefaultConnection();
IImSession imSession = gTalkConnection.getDefaultImSession();
```

IM Sessions are used to send text and data messages, set user presence, manage the IM contact roster, and manage group chats.

The IM Session is your primary interface for handling instant messaging in Android applications. As a result, the following code snippet shows a more typical implementation of the `ServiceConnection` used to bind the GTalk Service to an application. It ensures that an IM Session object is always valid.

```
private IGTalkConnection gTalkConnection = null;
private IImSession imSession = null;

private ServiceConnection gTalkServiceConnection = new ServiceConnection() {

  // When the service connects, get the default GTalk session.
  public void onServiceConnected(ComponentName className, IBinder service) {
    IGTalkService gtalkService = IGTalkService.Stub.asInterface(service);
    try {
      gTalkConnection = gtalkService.getDefaultConnection();
      imSession = gTalkConnection.getDefaultImSession();
    } catch (RemoteException e) { }
```

```
    }

    // When the service disconnects, clear the GTalk session.
    public void onServiceDisconnected(ComponentName className) {
      gTalkConnection = null;
      imSession = null;
    }
  };
```

Introducing Presence and the Contact Roster

Presence is a lightweight mechanism used in instant messaging to broadcast a user's availability.

Originally, presence was represented as a simple flag that indicated when a user was logged on and available to chat. This has gradually evolved into a more detailed status indicator that lets users describe their availability more accurately by indicating if they're available, busy, away from the computer, or offline. The recent popularity of applications like FriendFeed and Twitter has resulted in presence being expanded to include custom messages that can describe anything from a user's current activity to the music they're listening to.

Users can see the presence of all the people in their contact roster. The *contact roster* is a list of all the contacts with whom a user has an agreement to exchange messages and share presence information.

When adding someone to their roster, users are implicitly subscribing to updates of that person's presence, and changes to their own presence are propagated to all the contacts on their roster.

Instant messaging is an inherently portable technology — a user's presence and contact roster are maintained by the GTalk server, so the roster on an Android device is synchronized with Gmail chat and any desktop IM clients.

Managing the Contact Roster

Developers can access the contact roster to determine the presence of any of a user's IM contacts, monitor presence updates, add new contacts, remove existing ones, and handle subscription requests.

Accessing the IM Contact Roster

When it's made available, the contact roster should be accessible through a native Content Provider using the helper class `android.provider.Im.Contacts`. You can query it as you would any other Content Provider.

In the following snippet, you can see how to iterate over the roster to find the presence of each IM contact:

```
Uri uri = android.provider.Im.Contacts.CONTENT_URI_CHAT_CONTACTS;
Cursor c = managedQuery(uri, null, null, null);
if (c.moveToFirst()) {
  do {
    String username = c.getString(c.getColumnIndexOrThrow(Contacts.USERNAME));
    int presence = c.getInt(c.getColumnIndexOrThrow(Contacts.PRESENCE_STATUS));

    if (presence == Contacts.AVAILABLE) {
```

```
        // TODO: Do something
      }

    } while (c.moveToNext());
  }
```

Monitoring the Roster for Changes

To monitor the roster for changes and presence updates, implement an `IRosterListener` and register it with an IM Session using `addRemoteRosterListener`, as shown in the skeleton code below:

```
IRosterListener listener = new IRosterListener.Stub() {
  public void presenceChanged(String contact) throws RemoteException {
    // TODO Update the presence icon for the user.
  }

  public void rosterChanged() throws RemoteException {
    // TODO Update the roster UI.
  }

  public void selfPresenceChanged() throws RemoteException {
    // TODO Update the user's presence.
  }
};

try {
  imSession.addRemoteRosterListener(listener);
} catch (RemoteException e) { }
```

The Roster Listener includes event handlers that will be triggered when a contact has been added or removed from the current user's roster, when a contact's presence has changed, and if the user's presence has changed.

Adding Contacts to a Roster

To add a new contact to the user's roster, use `addContact`, specifying the contact username and a personal nickname to customize their entry on the roster, as shown below:

```
imSession.addContact("jim@dundermifflin.com", "Big Tuna", null);
```

The specified nickname is private and will only be visible to the device user.

People are only added to the roster after they've approved the request to become an instant messaging contact. After you attempt to add a contact, the target user receives an invitation (represented as a subscription request) that he or she can either approve or decline.

If the target user accepts the invitation, your user is placed in the target user's roster (and vice versa), and he or she will be able to exchange instant messages and receive presence updates.

Subscription requests are asynchronous, so you'll need to listen for changes in the roster to determine when a subscription request has been granted.

Handling Subscription Requests

Requests from others to add the device user to their contact lists should be presented to the user for his or her explicit approval or rejection.

Once the user has indicated his or her preference, you can approve or decline subscription requests using the `approveSubscriptionRequest` and `declineSubscriptionRequest` methods on an IM Session. As shown below, both methods take a contact name as a parameter; the `approve` method also accepts an optional nickname for the new contact being added.

```
imSession.approveSubscriptionRequest(sender, "nickname", null);
imSession.declineSubscriptionRequest(sender);
```

Removing and Blocking Contacts

In these times of fleeting attention and fickle friendships, there may come a time when a contact once added to a roster is no longer considered worthy of the honor. In extreme cases, users may choose to block all messages from a particular user.

Call `removeContact` from an IM Session to remove a contact from the user's roster and unsubscribe from his or her presence updates.

```
imSession.removeContact("whathaveyoudoneforme@lately.com");
```

When ignoring someone isn't enough, users can choose to block their messages entirely. The `blockContact` method effectively reverses the initial subscription-request approval and automatically denies any new subscription requests:

```
imSession.blockContact("ex@girlfriend.com");
```

Blocked contacts are added to the users "blocked list," which, like the roster itself, resides on the server. A contact blocked from Android will also be blocked in all other Google Talk clients.

Managing the User's Presence

The presence of the logged-in IM Session user is available using the `getPresence` method, as shown in the snippet below:

```
Presence p = imSession.getPresence();
```

This `Presence` object can be used to determine the user's IM visibility, his status, and any custom status message.

To change the user's presence, modify the `Presence` object and transmit it to the instant messaging server by calling `setPresence` on the IM Session.

The following code snippet shows how to set the user presence to `DO_NOT_DISTURB` and specifies a custom status message:

```
String customMessage = "Developing applications for Android. Professionally";
p.setStatus(Presence.Show.DND, customMessage);
imSession.setPresence(p);
```

Changes to a user's presence won't take effect until after they've been committed on the server. The best practice is to use a Roster Listener to react to the change in the user's presence once it's been applied on the server side.

Managing Chat Sessions

Chat Sessions are created within IM Sessions and are used to manage and participate in person-to-person chats and chat rooms. All text-based instant message chats are handled using the `IChatSession` interface, which offers methods for sending text or data messages and inviting new participants into a chat. You can attach a Chat Listener to a Chat Session to listen to the messages associated with it.

Handling Chat Sessions is particularly useful for integrating text messaging within your own applications. Using a Chat Session, you can create a chat room for multiplayer games, or integrate person-to-person messaging within a mobile social networking application.

Starting or Joining a Chat Session

A Chat Session represents the conduit through which all instant messaging communication with a target user passes, so you can only maintain a single Chat Session per contact per IM Session.

New Chat Sessions are created through an IM Session, using the `getChatSession` or `createChatSession` methods.

If a Chat Session already exists for a given contact, retrieve it by passing in the username of the person with whom you wish to converse, as shown in the following snippet. If there is no active Chat Session with the specified user, this method returns null.

```
IChatSession cs = imSession.getChatSession(targetContactEmailAddress);
```

If you haven't established a Chat Session with a particular user, create one using the `createChatSession` method, passing in the target contact's username. If the IM Session is unable to create a new Chat Session, this method will return null.

```
IChatSession chatSession = imSession.createChatSession(targetContactEmailAddress);
```

The following pattern checks to see if there is an existing Chat Session with a target user before creating a new one if necessary:

```
IChatSession chatSession = imSession.getChatSession(targetContactEmailAddress);
if (chatSession == null)
  chatSession = imSession.createChatSession(targetContactEmailAddress);
```

Group Chat Sessions are also represented using the `IChatSession` interface, but they're handled a little differently. Group chat functionality is explored in more detail later in this chapter.

Sending Instant Text Messages

Once you have an active Chat Session, use the `sendChatMessage` method to send messages to the contact(s) in that session, as shown in the following code snippet:

```
chatSession.sendChatMessage("Hello World!");
```

The message text specified will be transmitted to all the contacts involved in the current Chat Session.

Receiving Instant Text Messages

To listen for incoming messages, implement the `IChatListener` interface, overriding its `newMessageReceived` handler. You can register this interface with either a specific Chat Session or the more generic IM Session using the `addRemoteChatListener` method.

The following snippet shows the skeleton code for creating and registering a new Chat Listener interface for both a specific Chat Session and an IM Session. Note that the `IChatListener` interface includes a `Stub` class that you should extend when creating your own Chat Listener implementation.

```
IChatListener chatListener = new IChatListener.Stub() {

  public void newMessageReceived(String from, String body) {
    // TODO Handle incoming messages.
  }

  // Required group chat implementation stubs.
  public void convertedToGroupChat(String oldJid,
                                   String groupChatRoom,
                                   long groupId) {}
  public void participantJoined(String groupChatRoom, String nickname) {}
  public void participantLeft(String groupChatRoom, String nickname) {}
  public void chatClosed(String groupChatRoom) throws RemoteException {}
  public void chatRead(String arg0) throws RemoteException {}
};

// Add Chat Listener to the chat session.
chatSession.addRemoteChatListener(chatListener);

// Add Chat Listener to the instant messaging session.
imSession.addRemoteChatListener(chatListener);
```

Chat Listeners registered with an IM Session receive every message received by any Chat Session associated with that session, so the message handling here should be fairly generic. In contrast, listeners registered to a single Chat Session are only notified of messages and events relevant to that specific session.

Chat Rooms and Group Chats

Chat rooms are an excellent way to encourage a sense of community within a collaborative or multi-user application.

The GTalk Service supports chat rooms and group chats. They are managed using the same `IChatSession` interface used for simple P2P Chat Sessions.

To create a new chat room, use the `createGroupChatSession` method on an IM Session, passing in a nickname for the room and a list of users to invite, as shown in the following snippet:

```
String nickname = "Android Development";
String[] contacts = { "bill", "fred" };
imSession.createGroupChatSession(nickname, contacts);
```

Alternatively, you may want to join group chats that others have invited you to. Use the `IGroupChatInvitationListener` interface to listen for group chat invitations. Each invitation includes the address and password needed to join an existing chat room.

To join an existing chat room, use the `joinGroupChatSession` method from an active IM Session, passing in the address of the room you want to join, a nickname for you to identify it, and the password required to join, as shown in the following snippet:

```
imSession.joinGroupChatSession(address, nickname, password);
```

The following skeleton code shows how to register a Group Chat Invitation Listener on an active IM Session to listen for, and accept, invitations to join chat rooms.

```
IGroupChatInvitationListener listener = new IGroupChatInvitationListener.Stub() {
  public boolean onInvitationReceived(GroupChatInvitation _invite)
  throws RemoteException {
    String address = _invite.getRoomAddress();
    String password = _invite.getPassword();
    String nickname = _invite.getInviter();
    imSession.joinGroupChatSession(address, nickname, password);
    return true;
  }
};

try {
  imSession.addGroupChatInvitationListener(listener);
} catch (RemoteException e) { }
```

Managing Group Chat Sessions

You can get a list of participants in a Chat Session using the `getParticipants` method. You can also send text or data messages to each chat member as you would in a normal chat, as well as invite new members using `inviteContact`. The `leave` method lets you exit a chat room and end the session.

As with normal chats, you can listen to chat room messages by implementing and registering an `IChatListener`. As well as listening for chat messages, you can react to people joining or leaving the room.

The following skeleton code shows the implementation of a Chat Listener highlighting the group chat event handlers:

```
IChatListener groupChatListener = new IChatListener.Stub() {
  // Fired when a one-to-one chat becomes a group chat.
  public void convertedToGroupChat(String oldJid,
                                   String groupChatRoom,
                                   long groupId) throws RemoteException {
    // TODO Notify user that the conversation is now a group chat.
  }

  // Fired when a new person joins a chat room.
  public void participantJoined(String groupChatRoom, String nickname)
    throws RemoteException {
```

```
        // TODO Notify user that a new participant has joined the conversation.
      }

      // Fired when a participant leaves a chat room.
      public void participantLeft(String groupChatRoom, String nickname)
        throws RemoteException {
        // TODO Notify user a chat participant left.
      }

      // Fired when the group chat is closed
      public void chatClosed(String groupChatRoom) throws RemoteException {
        // TODO Close the chat.
      }

      public void chatRead(String arg0) throws RemoteException { }

      public void newMessageReceived(String from, String body) { }
    };
```

Sending and Receiving Data Messages

The GTalk Service includes functionality to transmit data messages between applications running on different devices. These data messages are handled separately from normal text chat messages and are invisible to users.

> **The functionality described in this section was removed prior to the version 1.0 release of the Android SDK. This is largely because of the security implications associated with the ability to remotely execute code on a target device. It is expected that this API will be exposed for developer access in future releases of Android, although it may differ from the implementation described here.**

GTalk data messages are a mechanism that lets you broadcast Intents over the air (OTA) to remote user devices. On the target device, the GTalk Service extracts the Intent from the received message and re-broadcasts it locally, where it's handled by the Intent resolution mechanism in the same way as locally broadcast Intents. The process is illustrated in Figure 9-1.

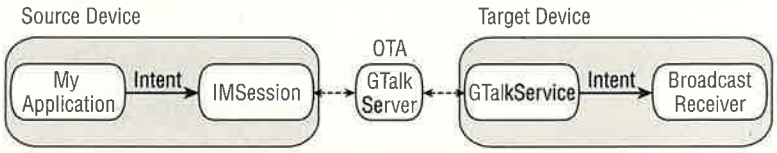


Figure 9-1

The result is an interface for broadcasting Intents on remote devices using instant messenger contacts. The broadcast Intent will be received by any Broadcast Receiver registered for the action represented by the Intent.

By extending the reach of your applications beyond the scope of the device on which they're running, you take on additional responsibilities to ensure that your applications are well behaved, and to take all possible precautions to ensure that your applications aren't open to exploitation by those looking to use this mechanism maliciously.

Data messages are an excellent way to support multi-user applications on distributed mobile devices, thanks to the low latency and rapid response times provided by the instant messaging architecture.

Transmitting Data Messages

The best practice is to create custom actions to use when broadcasting an Intent to a remote device, such as the one shown in the snippet below:

```
public static final String ACTION_OTA_ELIMINATE = "com.paad.ota_eliminate_action";
```

The next snippet shows how to create a simple Intent that will be packaged within a data message to transmit the above action to a remote device:

```
Intent intent = new Intent(ACTION_OTA_ELIMINATE);
```

As with normal broadcast Intents, you can package additional information within the Intent using the extras Bundle. These extras will be included in the Intent when it's re-broadcast on the remote device.

```
intent.putExtra("long", String.valueOf(location.getLatitude()));
intent.putExtra("lat", String.valueOf(location.getLatitude()));
intent.putExtra("target", "Sarah Conner");
intent.putExtra("sender", gTalk.getUsername());
```

Only String extras are currently supported in the OTA Intent broadcast mechanism. Non-string extras will be disregarded before transmission and won't be available on the target device.

Send the message using the `sendDataMessage` method, passing in the target username and the Intent to broadcast. The `sendDataMessage` is available on IM Session or Chat Session objects, as shown below:

```
String username = "T1000@sky.net";

// Send to target user.
imSession.sendDataMessage(username, intent);

// Send to all chat room participants.
chatSession.sendDataMessage(intent);
```

Receiving Data Messages

To listen for data messages, register a Broadcast Receiver that filters on the action String included in a transmitted Intent.

GTalk data messages are processed as normal broadcast Intents, so they have no sender information associated when they're received by a Broadcast Receiver. If you require such metadata, you should include them in the extras Bundle of the source Intent as was done in the code shown in the previous section.

The following skeleton code shows how to register a simple Broadcast Receiver implementation that can handle the Intent transmitted in the previous example:

```
BroadcastReceiver otaGTalkIntentReceiver = new BroadcastReceiver() {
  @Override
  public void onReceive(Context _context, Intent _intent) {
    if (_intent.getAction().equals(ACTION_OTA_ELIMINATE)) {
      String sender = _intent.getStringExtra("sender");
      String target = _intent.getStringExtra("target");

      String lat = _intent.getStringExtra("lat");
      String lng = _intent.getStringExtra("long");
      Location location = new Location(LocationManager.GPS_PROVIDER);
      location.setLatitude(Double.parseDouble(lat));
      location.setLongitude(Double.parseDouble(lng));

      // TODO: Do something with the data transmitted.
    }
  }
};

IntentFilter filter = new IntentFilter(ACTION_OTA_ELIMINATE);
registerReceiver(otaGTalkIntentReceiver, filter);
```

Introducing SMS

If you own a mobile phone that's less than two decades old, chances are you're familiar with SMS messaging. SMS (short messaging service) is now one of the most-used features on mobile phones, with many people favoring it over making phone calls.

SMS technology is designed to send short text messages between mobile phones. It provides support for sending both text messages (designed to be read by people) and data messages (meant to be consumed by applications).

As a mature mobile technology, there's a lot of information out there that describes the technical details of how an SMS message is constructed and transmitted over the air. Rather than rehash that here, the following sections focus on the practicalities of sending and receiving text and data messages within Android.

Using SMS in Your Application

Android offers full access to SMS functionality from within your applications with the `SMSManager`. Using the SMS Manager, you can replace the native SMS application or create new applications that send text messages, react to incoming texts, or use SMS as a data transport layer.

SMS message delivery is not timely, so SMS is not really suitable for anything that requires real-time responsiveness. That said, the widespread adoption and resiliency of SMS networks make it a particularly good tool for delivering content to non-Android users and reducing the dependency on third-party servers.

As a ubiquitous technology, SMS offers a mechanism you can use to send text messages to other mobile phone users, irrespective of whether they have Android phones.

Compared to the instant messaging mechanism available through the GTalk Service, using SMS to pass data messages between applications is slow, possibly expensive, and suffers from high latency. On the other hand, SMS is supported by almost every phone on the planet, so where latency is not an issue, and updates are infrequent, SMS data messages are an excellent alternative.

Sending SMS Messages

SMS messaging in Android is handled by the `SmsManager`. You can get a reference to the SMS Manager using the static method `SmsManger.getDefault`, as shown in the snippet below.

```
SmsManager smsManager = SmsManager.getDefault();
```

To send SMS messages, your applications require the `SEND_SMS` permission. To request this permission, add it to the manifest using a `uses-permission` tag, as shown below:

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

Sending Text Messages

To send a text message, use `sendTextMessage` from the SMS Manager, passing in the address (phone number) of your recipient and the text message you want to send, as shown in the snippet below:

```
String sendTo = "5551234";
String myMessage = "Android supports programmatic SMS messaging!";

smsManager.sendTextMessage(sendTo, null, myMessage, null, null);
```

The second parameter can be used to specify the SMS service center to use; entering null as shown in the previous snippet uses the default service center for your carrier.

The final two parameters let you specify Intents to track the transmission and successful delivery of your messages.

To react to these Intents, create and register Broadcast Receivers as shown in the next section.

Tracking and Confirming SMS Message Delivery

To track the transmission and delivery success of your outgoing SMS messages, implement and register Broadcast Receivers that listen for the actions you specify when creating the Pending Intents you pass in to the `sendTextMessage` method.

The first Pending Intent parameter, `sentIntent`, is fired when the message is either successfully sent or fails to send. The result code for the Broadcast Receiver that receives this Intent will be one of:

- `Activity.RESULT_OK` To indicate a successful transmission.
- `SmsManager.RESULT_ERROR_GENERIC_FAILURE` To indicate a nonspecific failure.

- `SmsManager.RESULT_ERROR_RADIO_OFF` When the connection radio is turned off.
- `SmsManager.RESULT_ERROR_NULL_PDU` To indicate a PDU failure.

The second Pending Intent parameter, `deliveryIntent`, is fired only after the destination recipient receives your SMS message.

The following code snippet shows a typical pattern for sending an SMS and monitoring the success of its transmission and delivery:

```
String SENT_SMS_ACTION = "SENT_SMS_ACTION";
String DELIVERED_SMS_ACTION = "DELIVERED_SMS_ACTION";

// Create the sentIntent parameter
Intent sentIntent = new Intent(SENT_SMS_ACTION);
PendingIntent sentPI = PendingIntent.getBroadcast(getApplicationContext(),
                                                  0,
                                                  sentIntent,
                                                  0);

// Create the deliveryIntent parameter
Intent deliveryIntent = new Intent(DELIVERED_SMS_ACTION);
PendingIntent deliverPI = PendingIntent.getBroadcast(getApplicationContext(),
                                                     0,
                                                     deliveryIntent,
                                                     0);

// Register the Broadcast Receivers
registerReceiver(new BroadcastReceiver() {
                    @Override
                    public void onReceive(Context _context, Intent _intent) {
                      switch (getResultCode()) {
                        case Activity.RESULT_OK:
                          [… send success actions … ]; break;
                        case SmsManager.RESULT_ERROR_GENERIC_FAILURE:
                          [… generic failure actions … ]; break;
                        case SmsManager.RESULT_ERROR_RADIO_OFF:
                          [… radio off failure actions … ]; break;
                        case SmsManager.RESULT_ERROR_NULL_PDU:
                          [… null PDU failure actions … ]; break;
                      }
                    }
                  },
                  new IntentFilter(SENT_SMS_ACTION));

registerReceiver(new BroadcastReceiver() {
                    @Override
                    public void onReceive(Context _context, Intent _intent) {
                      [… SMS delivered actions … ]
                    }
                  },
                  new IntentFilter(DELIVERED_SMS_ACTION));

// Send the message
smsManager.sendTextMessage(sendTo, null, myMessage, sentPI, deliverPI);
```

Monitoring Outgoing SMS Messages

The Android debugging bridge supports sending SMS messages between multiple emulator instances. To send an SMS from one emulator to another, specify the port number of the target emulator as the "to" address when sending a new message.

Android will automatically route your message to the target emulator instance, where it'll be handled as a normal SMS.

Conforming to the Maximum SMS Message Size

SMS text messages are normally limited to 160 characters, so longer messages need to be broken into a series of smaller parts. The SMS Manager includes the `divideMessage` method, which accepts a string as an input and breaks it into an ArrayList of messages wherein each is less than the allowable size. Use `sendMultipartTextMessage` to transmit the array of messages, as shown in the snippet below:

```
ArrayList<String> messageArray = smsManager.divideMessage(myMessage);
ArrayList<PendingIntent> sentIntents = new ArrayList<PendingIntent>();
for (int i = 0; i < messageArray.size(); i++)
  sentIntents.add(sentPI);

smsManager.sendMultipartTextMessage(sendTo,
                                    null,
                                    messageArray,
                                    sentIntents, null);
```

The `sentIntent` and `deliveryIntent` parameters in the `sendMultipartTextMessage` method are ArrayLists that can be used to specify different Pending Intents to fire for each message part.

Sending Data Messages

You can send binary data via SMS using the `sendDataMessage` method on an SMS Manager. The `sendDataMessage` method works much like `sendTextMessage`, but includes additional parameters for the destination port and an array of bytes that constitute the data you want to send.

The following skeleton code shows the basic structure of sending a data message:

```
Intent sentIntent = new Intent(SENT_SMS_ACTION);
PendingIntent sentPI = PendingIntent.getBroadcast(getApplicationContext(),
                                                  0,
                                                  sentIntent,
                                                  0);

short destinationPort = 80;
byte[] data = [ ... your data ... ];
smsManager.sendDataMessage(sendTo, null, destinationPort, data, sentPI, null);
```

Listening for SMS Messages

When a new SMS message is received by the device, a new broadcast Intent is fired with the `android.provider.Telephony.SMS_RECEIVED` action. Note that this is a String literal, SDK 1.0 does not include a reference to this string so you must specify it explicitly when using it in your applications.

For an application to listen for SMS Intent broadcasts, it first needs to be have the `RECEIVE_SMS` permission granted. Request this permission by adding a `uses-permission` tag to the application manifest, as shown in the following snippet:

```
<uses-permission
 android:name="android.permission.RECEIVE_SMS"/>
```

The SMS broadcast Intent includes the incoming SMS details. To extract the array of `SmsMessage` objects packaged within the SMS broadcast Intent bundle, use the `pdu` key to extract an array of SMS pdus, each of which represents an SMS message. To convert each pdu byte array into an SMS Message object, call `SmsMessage.createFromPdu`, passing in each byte array as shown in the snippet below:

```
Bundle bundle = intent.getExtras();
if (bundle != null) {
  Object[] pdus = (Object[]) bundle.get("pdus");
  SmsMessage[] messages = new SmsMessage[pdus.length];
  for (int i = 0; i < pdus.length; i++)
    messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
}
```

Each `SmsMessage` object contains the SMS message details, including the originating address (phone number), time stamp, and the message body.

The following example shows a Broadcast Receiver implementation whose `onReceive` handler checks incoming SMS texts that start with the string `@echo`, and then sends the same text back to the phone that sent it:

```
public class IncomingSMSReceiver extends BroadcastReceiver {
  private static final String queryString = "@echo ";
  private static final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";

  public void onReceive(Context _context, Intent _intent) {
    if (_intent.getAction().equals(SMS_RECEIVED)) {
      SmsManager sms = SmsManager.getDefault();

      Bundle bundle = _intent.getExtras();
      if (bundle != null) {
        Object[] pdus = (Object[]) bundle.get("pdus");
        SmsMessage[] messages = new SmsMessage[pdus.length];
        for (int i = 0; i < pdus.length; i++)
          messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);

        for (SmsMessage message : messages) {
          String msg = message.getMessageBody();
          String to = message.getOriginatingAddress();

          if (msg.toLowerCase().startsWith(queryString)) {
            String out = msg.substring(queryString.length());
            sms.sendTextMessage(to, null, out, null, null);
          }
        }
      }
    }
  }
}
```

To listen for incoming messages, register the Broadcast Receiver using an Intent Filter that listens for the `android.provider.Telephony.SMS_RECEIVED` action String, as shown in the code snippet below:

```
final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
IntentFilter filter = new IntentFilter(SMS_RECEIVED);
BroadcastReceiver receiver = new IncomingSMSReceiver();
registerReceiver(receiver, filter);
```

Simulating Incoming SMS Messages

There are two techniques available for simulating incoming SMS messages in the emulator. The first was described previoulsy in this section; you can send an SMS message from one emulator to another by using its port number as the destination address.

Alternatively, you can use the Android debug tools introduced in Chapter 2 to simulate incoming SMS messages from arbitrary numbers, as shown in Figure 9-2.

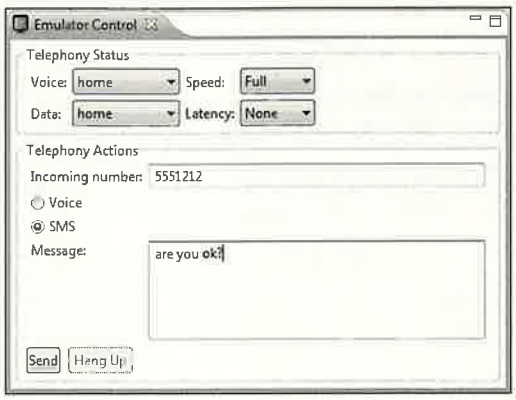


Figure 9-2

Handling Data SMS Messages

> **For security reasons, the version 1 release has restricted access to receiving data messages. The following section has been left to indicate how likely future functionality may be made available.**

Data messages are received in the same way as a normal SMS text message and are extracted in the same way as shown in the above section.

To extract the data transmitted within a data SMS, use the `getUserData` and `getUserDataHeader` methods, as shown in the following snippet:

```
byte[] data = msg.getUserData();
SmsHeader header = msg.getUserDataHeader();
```

The `getUserData` method returns a byte array of the data included in the message, while `getUserDataHeader` returns an array of metadata elements used to describe the data contained in the message.

Emergency Responder SMS Example

In this example, you'll create an SMS application that turns an Android phone into an emergency response beacon.

Once finished, the next time you're in unfortunate proximity to an alien invasion or find yourself in a robot-uprising scenario, you can set your phone to automatically respond to your friends' and family members' pleas for a status update with a friendly message (or a desperate cry for help).

To make things easier for your would-be saviors, you'll use location-based services to tell your rescuers exactly where to find you. The robustness of SMS network infrastructure makes SMS an excellent option for applications like this where reliability and accessibility are critical.

1. Start by creating a new `EmergencyResponder` project that features an `EmergencyResponder` Activity.

```
package com.paad.emergencyresponder;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Locale;
import java.util.concurrent.locks.ReentrantLock;
import java.util.List;
import android.app.Activity;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;

import android.content.BroadcastReceiver;
import android.content.SharedPreferences;
import android.location.Address;
import android.location.Geocoder;
import android.location.Location;
import android.location.LocationManager;

import android.os.Bundle;
import android.telephony.gsm.SmsManager;
import android.telephony.gsm.SmsMessage;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.ListView;

public class EmergencyResponder extends Activity {

  @Override
```