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);
}
```


REGIONS 1062
PART 2 OF 3

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.

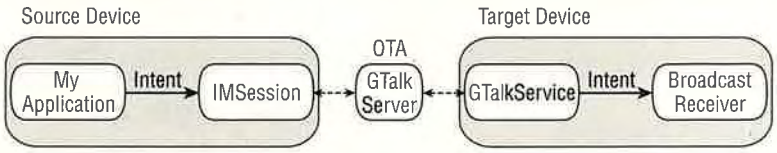


Figure 9-1

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.

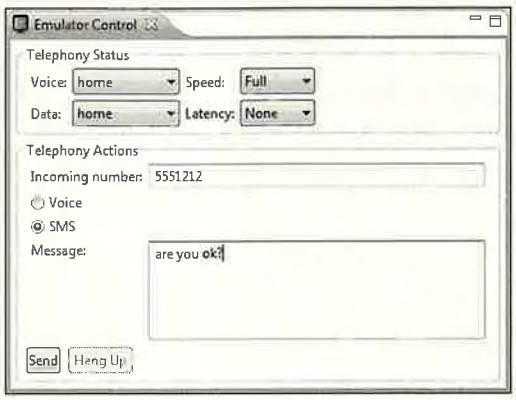


Figure 9-2

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
```

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

2. Add permissions for finding your location as well as sending and receiving incoming SMS messages to the project manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.paad.emergencyresponder">
  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
    <activity android:name=".EmergencyResponder"
              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_GPS"/>

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

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

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

</manifest>
```

3. Modify the main.xml layout resource. Include a List View to show the people requesting a status update and a series of buttons that users can press to send response SMS messages. Use external resource references to fill in the button text; you'll create them in Step 4.

```
<?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">
  <TextView
    android:id="@+id/labelRequestList"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="These people want to know if you're ok"
    android:layout_alignParentTop="true"/>
  <LinearLayout
    android:id="@+id/buttonLayout"
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="5px"
    android:layout_alignParentBottom="true">
    <CheckBox
      android:id="@+id/checkboxSendLocation"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Include Location in Reply"/>
    <Button
      android:id="@+id/okButton"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/respondAllClearButtonText"/>
    <Button
      android:id="@+id/notOkButton"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/respondMaydayButtonText"/>
    <Button
      android:id="@+id/autoResponder"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Setup Auto Responder"/>
  </LinearLayout>
  <ListView
    android:id="@+id/myListView"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_below="@id/labelRequestList"
    android:layout_above="@id/buttonLayout"/>
</RelativeLayout>
```

4. Update the external strings.xml resource to include the text for each button and default response messages to use when responding, with "I'm safe" or "I'm in danger" messages. You should also define the incoming message text to use when detecting requests for status responses.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Emergency Responder</string>

  <string name="respondAllClearButtonText">I am Safe and Well</string>
  <string name="respondMaydayButtonText">MAYDAY! MAYDAY! MAYDAY!</string>

  <string name="respondAllClearText">I am safe and well. Worry not!</string>
  <string name="respondMaydayText">Tell my mother I love her.</string>

  <string name="querystring">are you ok?</string>
</resources>
```

5. At this point, the GUI will be complete, so starting the application should show you the screen shown in Figure 9-3.



Figure 9-3

6. Create a new Array List of Strings within the `EmergencyResponder` Activity to store the phone numbers of the incoming requests for your status. Bind the Array List to the List View, using an Array Adapter in the Activity's `onCreate` method, and create a new `ReentrantLock` object to ensure thread safe handling of the Array List.

   Take the opportunity to get a reference to the Check Box and to add Click Listeners for each of the response buttons. Each button should call the `respond` method, while the Setup Auto Responder button should call the `startAutoResponder` stub.

```
ReentrantLock lock;

CheckBox locationCheckBox;

ArrayList<String> requesters;
ArrayAdapter<String> aa;

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

  lock = new ReentrantLock();
  requesters = new ArrayList<String>();
  wireUpControls();
}

private void wireUpControls() {
  locationCheckBox = (CheckBox)findViewById(R.id.checkboxSendLocation);
```

```
    ListView myListView = (ListView)findViewById(R.id.myListView);

    int layoutID = android.R.layout.simple_list_item_1;
    aa = new ArrayAdapter<String>(this, layoutID, requesters);

    myListView.setAdapter(aa);

    Button okButton = (Button)findViewById(R.id.okButton);
    okButton.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
        respond(true, locationCheckBox.isChecked());
      }
    });

    Button notOkButton = (Button)findViewById(R.id.notOkButton);
    notOkButton.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
        respond(false, locationCheckBox.isChecked());
      }
    });

    Button autoResponderButton = (Button)findViewById(R.id.autoResponder);
    autoResponderButton.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
        startAutoResponder();
      }
    });
  }

  public void respond(boolean _ok, boolean _includeLocation) {}

  private void startAutoResponder() {}
```

7. Next, implement a Broadcast Receiver that will listen for incoming SMS messages.

   7.1 Start by creating a new static string variable to store the incoming SMS message intent action.

```
public static final String SMS_RECEIVED =
  "android.provider.Telephony.SMS_RECEIVED";
```

   7.2 Then create a new Broadcast Receiver as a variable in the `EmergencyResponder` Activity. The receiver should listen for incoming SMS messages and call the `requestRecieved` method when it sees SMS messages containing the "are you safe" String you defined as an external resource in Step 4.

```
BroadcastReceiver emergencyResponseRequestReceiver = new BroadcastReceiver() {

  @Override
  public void onReceive(Context _context, Intent _intent) {
    if (_intent.getAction().equals(SMS_RECEIVED)) {
      String queryString = getString(R.string.querystring);

      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) {
              if (message.getMessageBody().toLowerCase().contains(queryString)) {
                requestReceived(message.getOriginatingAddress());
              }
            }
          }
        }
      }

    };

    public void requestReceived(String _from) {}
```

8. Update the `onCreate` method of the Emergency Responder Activity to register the Broadcast Receiver created in Step 7.

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

  lock = new ReentrantLock();
  requesters = new ArrayList<String>();
  wireUpControls();

  IntentFilter filter = new IntentFilter(SMS_RECEIVED);
  registerReceiver(emergencyResponseRequestReceiver, filter);
}
```

9. Update the `requestReceived` method stub so that it adds the originating number of each status request's SMS to the "requesters" Array List.

```
public void requestReceived(String _from) {
  if (!requesters.contains(_from)) {
    lock.lock();
    requesters.add(_from);
    aa.notifyDataSetChanged();
    lock.unlock();
  }
}
```

10. The Emergency Responder Activity should now be listening for status request SMS messages and adding them to the List View as they arrive. Start the application and use the DDMS emulator control to simulate incoming SMS messages, as shown in Figure 9-4.



Figure 9-4

11. Now update the Activity to let users respond to these status requests.

Start by completing the `respond` method stub you created in Step 6. It should iterate over the Array List of status requesters and send a new SMS message to each. The SMS message text should be based on the response strings you defined as resources in Step 4. Fire the SMS using an overloaded `respond` method that you'll complete in the next step.

```
public void respond(boolean _ok, boolean _includeLocation) {
  String okString = getString(R.string.respondAllClearText);
  String notOkString = getString(R.string.respondMaydayText);

  String outString = _ok ? okString : notOkString;

  ArrayList<String> requestersCopy = (ArrayList<String>)requesters.clone();

  for (String to : requestersCopy)
    respond(to, outString, _includeLocation);
}

private void respond(String _to, String _response, boolean _includeLocation) {}
```

12. Update the `respond` method that handles the sending of each response SMS.

Start by removing each potential recipient from the "requesters" Array List before sending the SMS. If you are responding with your current location, use the Location Manager to find it before sending a second SMS with your current position as raw longitude/latitude points and a geocoded address.

```
public void respond(String _to, String _response, boolean _includeLocation) {
  // Remove the target from the list of people we need to respond to.
  lock.lock();
  requesters.remove(_to);
  aa.notifyDataSetChanged();
```

```
        lock.unlock();

        SmsManager sms = SmsManager.getDefault();

        // Send the message
        sms.sendTextMessage(_to, null, _response, null, null);

        StringBuilder sb = new StringBuilder();

        // Find the current location and send it as SMS messages if required.
        if (_includeLocation) {
          String ls = Context.LOCATION_SERVICE;
          LocationManager lm = (LocationManager)getSystemService(ls);
          Location l = lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);

          sb.append("I'm @:\n");
          sb.append(l.toString() + "\n");

          List<Address> addresses;
          Geocoder g = new Geocoder(getApplicationContext(), Locale.getDefault());
          try {
            addresses = g.getFromLocation(l.getLatitude(), l.getLongitude(), 1);
            if (addresses != null) {
              Address currentAddress = addresses.get(0);
              if (currentAddress.getMaxAddressLineIndex() > 0) {
                for (int i = 0; i < currentAddress.getMaxAddressLineIndex(); i++) {
                  sb.append(currentAddress.getAddressLine(i));
                  sb.append("\n");
                }
              }
              else {
                if (currentAddress.getPostalCode() != null)
                  sb.append(currentAddress.getPostalCode());
              }
            }
          } catch (IOException e) {}

          ArrayList<String> locationMsgs = sms.divideMessage(sb.toString());
          for (String locationMsg : locationMsgs)
            sms.sendTextMessage(_to, null, locationMsg, null, null);
        }
      }
```

At the time of production, `sendTextMessage` *required a non-null value for the fourth parameter (*`sentIntent`*). In this example, this parameter is added in Step 13, so running the application now will cause it to throw an exception.*

13. In emergencies, it's important that messages get through. Improve the robustness of the application by including auto-retry functionality. Monitor the success of your SMS transmissions so that you can re-broadcast a message if it doesn't successfully send.

13.1. Start by creating a new public static String in the Emergency Responder Activity to be used as a local "SMS Sent" action.

```
public static final String SENT_SMS = "com.paad.emergencyresponder.SMS_SENT";
```

13.2. Update the `respond` method to include a new `PendingIntent` that broadcasts the action created in the previous step when the SMS transmission has completed. The packaged Intent should include the intended recipient's number as an extra.

```
public void respond(String _to, String _response, boolean _includeLocation) {
  // Remove the target from the list of people we need to respond to.
  lock.lock();
  requesters.remove(_to);
  aa.notifyDataSetChanged();
  lock.unlock();

  SmsManager sms = SmsManager.getDefault();

  Intent intent = new Intent(SENT_SMS);
  intent.putExtra("recipient", _to);

  PendingIntent sentIntent = PendingIntent.getBroadcast(getApplicationContext(),
                                                        0,
                                                        intent,
                                                        0);

  // Send the message
  sms.sendTextMessage(_to, null, _response, sentIntent, null);

  StringBuilder sb = new StringBuilder();

  if (_includeLocation) {
    [ … existing respond method that finds the current location … ]
    ArrayList<String> locationMsgs = sms.divideMessage(sb.toString());
    for (String locationMsg : locationMsgs)
      sms.sendTextMessage(_to, null, locationMsg, sentIntent, null);
  }
}
```

13.3. Then implement a new Broadcast Receiver to listen for this broadcast Intent. Override its `onReceive` handler to confirm that the SMS was successfully delivered; if it wasn't, then add the intended recipient back on to the requesters Array List.

```
private BroadcastReceiver attemptedDeliveryReceiver = new BroadcastReceiver()
{
  @Override
  public void onReceive(Context _context, Intent _intent) {
    if (_intent.getAction().equals(SENT_SMS)) {
      if (getResultCode() != Activity.RESULT_OK) {
        String recipient = _intent.getStringExtra("recipient");
        requestReceived(recipient);
      }
    }
  }
};
```

13.4. Finally, register the new Broadcast Receiver by extending the `onCreate` method of the Emergency Responder Activity.

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

  lock = new ReentrantLock();
  requesters = new ArrayList<String>();
  wireUpControls();

  IntentFilter filter = new IntentFilter(SMS_RECEIVED);
  registerReceiver(emergencyResponseRequestReceiver, filter);

  IntentFilter attemptedDeliveryfilter = new IntentFilter(SENT_SMS);
  registerReceiver(attemptedDeliveryReceiver, attemptedDeliveryfilter);
}
```

You can now run the application. To test it, you need to open two emulator instances with the application running in one of them.

Use the DDMS emulator controls to mimic sending an "are you safe" message from one emulator to the other (using its port number as the originating number). When you press one of the response buttons, you should see a new SMS message appear in the mimicked emulator.

Automating the Emergency Responder

In the following example, you'll fill in the code behind the Set up Auto Responder button added in the previous example, to let the Emergency Responder automatically respond to status update requests.

1. Start by creating a new autoresponder.xml layout resource that will be used to lay out the automatic response configuration window. Include an `EditText` for entering a status message to send, a `Spinner` to choose the auto-response expiry time, and a `CheckBox` to let users choose if they want to include their location in the automated responses.

```
<?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:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Respond With"/>
  <EditText
    android:id="@+id/responseText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
  <CheckBox
    android:id="@+id/checkboxLocation"
```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Transmit Location"/>
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Auto Respond For"/>
  <Spinner
    android:id="@+id/spinnerRespondFor"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"/>
  <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    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="Enable"/>
    <Button
      android:id="@+id/cancelButton"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Disable"/>
  </LinearLayout>
</LinearLayout>
```

2. Update the application's string.xml resource to define a name for an application `SharedPreference` and strings to use for each of its keys.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Emergency Responder</string>

  <string name="respondAllClearButtonText">I am Safe and Well</string>
  <string name="respondMaydayButtonText">MAYDAY! MAYDAY! MAYDAY!</string>

  <string name="respondAllClearText">I am safe and well. Worry not!</string>
  <string name="respondMaydayText">Tell my mother I love her.</string>

  <string name="querystring">"are you ok?"</string>

  <string name="user_preferences">com.paad.emergencyresponder.preferences</string>
  <string name="includeLocationPref">PREF_INCLUDE_LOC</string>
  <string name="responseTextPref">PREF_RESPONSE_TEXT</string>
  <string name="autoRespondPref">PREF_AUTO_RESPOND</string>
  <string name="respondForPref">PREF_RESPOND_FOR</string>
</resources>
```

You should also take this opportunity to externalize the strings used for labels within the layout.

3. Then create a new arrays.xml resource, and create arrays to use for populating the Spinner.

```
<resources>
  <string-array name="respondForDisplayItems">
    <item>- Disabled -</item>
    <item>Next 5 minutes</item>
    <item>Next 15 minutes</item>
    <item>Next 30 minutes</item>
    <item>Next hour</item>
    <item>Next 2 hours</item>
    <item>Next 8 hours</item>
  </string-array>

  <array name="respondForValues">
    <item>0</item>
    <item>5</item>
    <item>15</item>
    <item>30</item>
    <item>60</item>
    <item>120</item>
    <item>480</item>
  </array>
</resources>
```

4. Now create a new `AutoResponder` Activity, populating it with the layout you created in Step 1.

```
package com.paad.emergencyresponder;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.res.Resources;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.BroadcastReceiver;
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.EditText;
import android.widget.Spinner;

public class AutoResponder extends Activity {

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

}
```

5. Update `onCreate` further to get references to each of the controls in the layout and wire up the Spinner using the arrays defined in Step 3. Create two new stub methods, `savePreferences` and `updateUIFromPreferences`, that will be updated to save the auto-responder settings to a named `SharedPreference` and apply the saved `SharedPreferences` to the current UI, respectively.

```
Spinner respondForSpinner;
CheckBox locationCheckbox;
EditText responseTextBox;

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

5.1. Start by getting references to each View.

```
respondForSpinner = (Spinner)findViewById(R.id.spinnerRespondFor);
locationCheckbox = (CheckBox)findViewById(R.id.checkboxLocation);
responseTextBox = (EditText)findViewById(R.id.responseText);
```

5.2. Populate the Spinner to let users select the auto-responder expiry time.

```
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(
                                     this,
                                     R.array.respondForDisplayItems,
                                     android.R.layout.simple_spinner_item);

adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
respondForSpinner.setAdapter(adapter);
```

5.3. Now wire up the OK and Cancel buttons to let users save or cancel setting changes.

```
Button okButton = (Button) findViewById(R.id.okButton);
okButton.setOnClickListener(new View.OnClickListener() {
  public void onClick(View view) {
    savePreferences();
    setResult(RESULT_OK, null);
    finish();
  }
});

Button cancelButton = (Button) findViewById(R.id.cancelButton);
cancelButton.setOnClickListener(new View.OnClickListener() {
  public void onClick(View view) {
    respondForSpinner.setSelection(-1);
    savePreferences();
    setResult(RESULT_CANCELED, null);
    finish();
  }
});
```

5.4. Finally, make sure that when the Activity starts, it updates the GUI to represent the current settings.

```
// Load the saved preferences and update the UI
updateUIFromPreferences();
```

5.5. Close off the `onCreate` method, and add the `updateUIFromPreferences` and `savePreferences` stubs.

```
}

private void updateUIFromPreferences() {}

private void savePreferences() {}
```

6. Next, complete the two stub methods from Step 5. Start with `updateUIFromPreferences`; it should read the current saved `AutoResponder` preferences and apply them to the UI.

```
private void updateUIFromPreferences() {
  // Get the saves settings
  String preferenceName = getString(R.string.user_preferences);
  SharedPreferences sp = getSharedPreferences(preferenceName, 0);

  boolean autoRespond = sp.getBoolean(getString(R.string.autoRespondPref), false);
  String respondText = sp.getString(getString(R.string.responseTextPref), "");
  boolean includeLoc = sp.getBoolean(getString(R.string.includeLocationPref),
                                     false);
  int respondForIndex = sp.getInt(getString(R.string.respondForPref), 0);

  // Apply the saved settings to the UI
  if (autoRespond)
    respondForSpinner.setSelection(respondForIndex);
  else
    respondForSpinner.setSelection(0);

  locationCheckbox.setChecked(includeLoc);
  responseTextBox.setText(respondText);
}
```

7. Complete the `savePreferences` stub to save the current UI settings to a Shared Preferences file.

```
private void savePreferences() {
  // Get the current settings from the UI
  boolean autoRespond = respondForSpinner.getSelectedItemPosition() > 0;
  int respondForIndex = respondForSpinner.getSelectedItemPosition();
  boolean includeLoc = locationCheckbox.isChecked();
  String respondText = responseTextBox.getText().toString();

  // Save them to the Shared Preference file
  String preferenceName = getString(R.string.user_preferences);
  SharedPreferences sp = getSharedPreferences(preferenceName, 0);

  Editor editor = sp.edit();
```

```
    editor.putBoolean(getString(R.string.autoRespondPref), autoRespond);
    editor.putString(getString(R.string.responseTextPref), respondText);
    editor.putBoolean(getString(R.string.includeLocationPref), includeLoc );
    editor.putInt(getString(R.string.respondForPref), respondForIndex );
    editor.commit();

    // Set the alarm to turn off the autoresponder
    setAlarm(respondForIndex);
  }

  private void setAlarm(int respondForIndex) {}
```

8. The `setAlarm` stub from Step 8 is used to create a new Alarm that fires an Intent that should result in the AutoResponder being disabled.

You'll need to create a new `Alarm` object and a `BroadcastReceiver` that listens for it before disabling the auto-responder accordingly.

8.1. Start by creating the action String that will represent the Alarm Intent.

```
public static final String alarmAction =
  "com.paad.emergencyresponder.AUTO_RESPONSE_EXPIRED";
```

8.2. Then create a new Broadcast Receiver instance that listens for an Intent that includes the action specified in Step 7. When this Intent is received, it should modify the auto-responder settings to disable the automatic response.

```
private BroadcastReceiver stopAutoResponderReceiver = new BroadcastReceiver() {
  @Override
  public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(alarmAction)) {
      String preferenceName = getString(R.string.user_preferences);
      SharedPreferences sp = getSharedPreferences(preferenceName, 0);

      Editor editor = sp.edit();
      editor.putBoolean(getString(R.string.autoRespondPref), false);
      editor.commit();
    }
  }
};
```

8.3. Finally, complete the `setAlarm` method. It should cancel the existing alarm if the auto-responder is turned off; otherwise, it should update it with the latest expiry time.

```
PendingIntent intentToFire;

private void setAlarm(int respondForIndex) {
  // Create the alarm and register the alarm intent receiver.

  AlarmManager alarms = (AlarmManager)getSystemService(ALARM_SERVICE);

  if (intentToFire == null) {
    Intent intent = new Intent(alarmAction);
```

```
      intentToFire = PendingIntent.getBroadcast(getApplicationContext(),
                                                0,intent,0);

      IntentFilter filter = new IntentFilter(alarmAction);

      registerReceiver(stopAutoResponderReceiver, filter);
    }

    if (respondForIndex < 1)
      // If "disabled" is selected, cancel the alarm.
      alarms.cancel(intentToFire);
    else {
      // Otherwise find the length of time represented by the selection and
      // and set the alarm to trigger after that time has passed.
      Resources r = getResources();
      int[] respondForValues = r.getIntArray(R.array.respondForValues);
      int respondFor = respondForValues [respondForIndex];

      long t = System.currentTimeMillis();
      t = t + respondFor*1000*60;

      // Set the alarm.
      alarms.set(AlarmManager.RTC_WAKEUP, t, intentToFire);
    }
  }
```

9. That completes the `AutoResponder`, but before you can use it, you'll need to add it to your application manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.paad.emergencyresponder">
  <application
    android:icon="@drawable/icon"
    android:label="@string/app_name">
    <activity
      android:name=".EmergencyResponder"
      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=".AutoResponder"
      android:label="Auto Responder Setup"/>
  </application>

  <uses-permission android:name="android.permission.ACCESS_GPS"/>
  <uses-permission android:name="android.permission.ACCESS_LOCATION"/>
  <uses-permission android:name="android.permission.RECEIVE_SMS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

</manifest>
```

10. To enable the auto-responder, return to the Emergency Responder Activity and update the `startAutoResponder` method stub that you created in the previous example. It should open the `AutoResponder` Activity you just created.

```
private void startAutoResponder() {
  startActivityForResult(new Intent(EmergencyResponder.this,
                                    AutoResponder.class), 0);
}
```

11. If you start your project, you should now be able to bring up the Auto Responder settings window to set the auto-response settings. It should appear as shown in Figure 9-5.

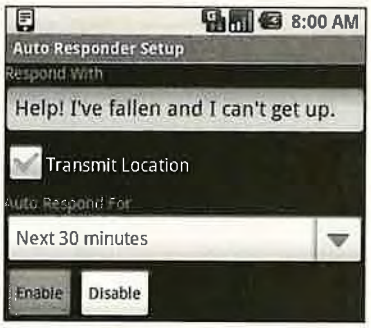


Figure 9-5

12. The final step is to update the `requestReceived` method in the Emergency Responder Activity to check if the auto-responder has been enabled.

If it has, the `requestReceived` method should automatically execute the `respond` method, using the message and location settings defined in the application's SharedPreferences.

```
public void requestReceived(String _from) {
  if (!requesters.contains(_from)) {
    lock.lock();
    requesters.add(_from);
    aa.notifyDataSetChanged();
    lock.unlock();

    // Check for auto-responder
    String preferenceName = getString(R.string.user_preferences);
    SharedPreferences prefs = getSharedPreferences(preferenceName, 0);

    String autoRespondPref = getString(R.string.autoRespondPref)
    boolean autoRespond = prefs.getBoolean(autoRespondPref, false);

    if (autoRespond) {
      String responseTextPref = getString(R.string.responseTextPref);
      String includeLocationPref = getString(R.string.includeLocationPref);

      String respondText = prefs.getString(responseTextPref, "");
```

```
                boolean includeLoc = prefs.getBoolean(includeLocationPref, false);

                respond(_from, respondText, includeLoc);
            }
        }
    }
```

You should now have a fully functional interactive and automated emergency responder.

You can test it in the same way as described in the previous example by using a second emulator instance to receive the response messages, and the emulator controls to send the requests for status updates.

Summary

Technologies like SMS and instant messaging are providing an increasingly versatile platform for person-to-person communication.

Android lets you use these text-based communication channels to create applications that let users send messages using instant messengers and SMS texts, as well as supplying an invisible data conduit for your applications to exchange data between devices.

In this chapter, you learned how to connect to IM Sessions using the GTalk Service and how to send and receive text and data messages using these sessions. You learned about presence, how to set your own presence, and how to find the presence of the contacts on the IM roster.

You also used the SMS Manager to send and receive text and data messages from your applications.

This chapter also showed you how future SDK release may allow you to:

- Add and remove instant messaging contacts.
- Block contacts and monitor the roster for changes.
- Manage group chats and chat rooms.

Chapter 10 explores access to the low-level mobile hardware.

Using the phone's telephony services, you'll initiate new calls and monitor both outgoing and incoming calls. You'll be introduced to Android's multimedia capabilities and use the media API to play back and record a variety of media resources.

You'll also learn how to interact with the Sensor Manager to access the compass and accelerometer before investigating network management using the Wi-Fi and Bluetooth APIs.

10

Accessing Android Hardware

Android's application-neutral APIs provide low-level access to the increasingly diverse hardware commonly available on mobile devices. The ability to monitor and control these hardware features provides a great incentive for application development on the Android platform.

The hardware APIs available include:

- A telephony package that provides access to calls and phone status.
- A multimedia playback and recording library.
- Access to the device camera for taking pictures and previewing video.
- Extensible support for sensor hardware.
- Accelerometer and compass APIs to monitor orientation and movement.
- Communications libraries for managing Bluetooth, network, and Wi-Fi hardware.

In this chapter, you'll take a closer look at some of these hardware APIs. In particular, you'll learn how to play and record multimedia content including audio, video, and still images, as well as use the camera to capture images and preview and capture live video.

You'll also learn how to monitor hardware sensors using the Sensor Manager. The accelerometer and compass sensors will be used to determine changes in the device orientation and acceleration — which is extremely useful for creating motion-based User Interfaces — and lets you add new dimensions to your location-based applications.

Finally, you'll take a closer look at the communication hardware by examining the telephony package for monitoring phone state and phone calls, as well as seeing what's available in the Bluetooth, networking, and Wi-Fi APIs.

Using the Media APIs

The only modern technology that can compete with mobile phones for ubiquity is the portable digital media player. As a result, the multimedia capabilities of portable devices are a significant consideration for many consumers.

Android's open platform- and provider-agnostic philosophy ensures that it offers a multimedia library capable of playing and recording a wide range of media formats, both locally and streamed.

Android exposes this library to your applications, providing comprehensive multimedia functionality including recording and playback of audio, video, and still-image media stored locally, within an application, or streamed over a data connection.

At the time of print, Android supported the following multimedia formats:

- JPEG
- PNG
- OGG
- Mpeg 4
- 3GPP
- MP3
- Bitmap

Playing Media Resources

Multimedia playback in Android is handled by the `MediaPlayer` class. You can play back media stored as application resources, local files, or from a network URI.

To play a media resource, create a new Media Player instance, and assign it a media source to play using the `setDataSource` method. Before you can start playback, you need to call `prepare`, as shown in the following code snippet:

```
String MEDIA_FILE_PATH = Settings.System.DEFAULT_RINGTONE_URI.toString();
MediaPlayer mpFile = new MediaPlayer();

try {
  mpFile.setDataSource(MEDIA_FILE_PATH);
  mpFile.prepare();
  mpFile.start();
}
catch (IllegalArgumentException e) {}
catch (IllegalStateException e) {}
catch (IOException e) {}
```

Alternatively, the static `create` methods work as shortcuts, accepting media resources as a parameter and preparing them for playback, as shown in the following example, which plays back an application resource:

```
MediaPlayer mpRes = MediaPlayer.create(context, R.raw.my_sound);
```

Note that if you use a `create` method to generate your `MediaPlayer` object, `prepare` is called for you.

Once a Media Player is prepared, call `start` as shown below to begin playback of the associated media resource.

```
mpRes.start();

mpFile.start();
```

The Android Emulator simulates audio playback using the audio output of your development platform.

The Media Player includes `stop`, `pause`, and `seek` methods to control playback, as well as methods to find the duration, position, and image size of the associated media.

To loop or repeat playback, use the `setLooping` method.

When playing video resources, `getFrame` will take a screen grab of video media at the specified frame and return a bitmap resource.

Once you've finished with the Media Player, be sure to call `release` to free the associated resources, as shown below:

```
mpRes.release();
mpFile.release();
```

Since Android only supports a limited number of simultaneous Media Player objects, not releasing them can cause runtime exceptions.

On Android devices, the Media Player always plays audio using the standard output device — the speaker or connected Bluetooth headset. It's not currently possible to play audio into a phone conversation.

Recording Multimedia

Multimedia recording is handled by the aptly named `MediaRecorder` class. To record audio or video, create a new Media Recorder object, as shown in the following code snippet:

```
MediaRecorder mediaRecorder = new MediaRecorder();
```

Before you can record any media in Android, your application needs the `RECORD_AUDIO` and / or `RECORD_VIDEO` permissions. Add `uses-permission` tags for each of them, as appropriate, in your application manifest.

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

The ability to record video has been restricted for the version 1.0 release of Android; however, Audio recording is still available.

The Media Recorder can be used to configure the video and audio sources (generally the camera and microphone), output format, video size and frame rate, and the video and audio encoders to use.

The following code snippet shows how to configure a Media Recorder to record audio from the microphone using the default format and encoder:

The emulator supports recording of audio using the microphone device attached to your development platform.

```
// Set the audio source.
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
// Set the output format.
mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.DEFAULT);
// Set the audio encoders to use.
mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
```

Once you've defined your input source and output format, assign a file to store the recorded media using the `setOutputFile` method as shown below:

```
mediaRecorder.setOutputFile("myoutputfile.mp4");
```

The `setOutputFile` method must be called before `prepare` and after `setOutputFormat` or it will throw an Illegal State Exception.

To begin recording, call `prepare` followed by the `start` method, as shown below:

```
mediaRecorder.prepare();
mediaRecorder.start();
```

When you're finished, call `stop` to end the playback, followed by `release` to free the Media Recorder resources:

```
mediaRecorder.stop();
mediaRecorder.release();
```

When recording video, it's generally considered good practice to display a preview of the recorded video in real time. Using the `setPreviewDisplay` method, you can assign a `Surface` to display the video preview.

As with any other resource, media files created by your application will be unavailable to others. As a result, it's good practice to use the Media Store Content Provider to assign metadata, select a file location, and publish the recorded media to share recordings with other applications.

To do that, after recording new media create a new `ContentValues` object to add a new record to the Media Store. The metadata you specify here can include the details including the title, time stamp, and geocoding information for your new media file, as shown in the code snippet below:

```
ContentValues content = new ContentValues(3);
content.put(Audio.AudioColumns.TITLE, "TheSoundandtheFury");
```

```
content.put(Audio.AudioColumns.DATE_ADDED,
            System.currentTimeMillis() / 1000);
content.put(Audio.Media.MIME_TYPE, "audio/amr");
```

You must also specify the absolute path of the media file being added:

```
content.put(MediaStore.Audio.Media.DATA,
            "myoutputfile.mp4");
```

Get access to the application's `ContentResolver`, and use it to insert this new row into the Media Store as shown in the following code snippet:

```
ContentResolver resolver = getContentResolver();
Uri uri = resolver.insert(Audio.Media.EXTERNAL_CONTENT_URI, content);
```

Once the media file has been inserted into the media store you should announce it's availability using a broadcast Intent as shown below:

```
sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE, uri));
```

Using the Camera

The popularity of digital cameras (particularly within phone handsets) has caused their prices to drop just as their size has shrunk dramatically. It's now becoming difficult to even find a mobile phone without a camera, and Android devices are unlikely to be exceptions.

To access the camera hardware, you need to add the `CAMERA` permission to your application manifest, as shown here:

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

This grants access to the Camera Service. The `Camera` class lets you adjust camera settings, take pictures, and manipulate streaming camera previews.

To access the Camera Service, use the static `open` method on the `Camera` class. When your application has finished with the camera, remember to relinquish your hold on the Service by calling `release` following the simple use pattern shown in the code snippet below:

```
Camera camera = Camera.open();
  [ … Do things with the camera … ]
camera.release();
```

Controlling Camera Settings

The current camera settings are available as a `Camera.Parameters` object. Call the `getParameters` method on the Camera to access the current parameters.

You can use the `set*` methods on the returned Parameters to modify the settings. To apply changes, call `setParameters`, passing in the modified values as shown below:

```
Camera.Parameters parameters = camera.getParameters();
parameters.setPictureFormat(PixelFormat.JPEG);
camera.setParameters(parameters);
```

The Camera Parameters can be used to specify the image and preview size, image format, and preview frame rate.

Using the Camera Preview

Access to the camera's streaming video means that you can incorporate live video into your applications. Some of the most exciting early Android applications have used this functionality as the basis for augmenting reality.

The camera preview can be displayed in real time onto a `Surface`, as shown in the code snippet below:

```
camera.setPreviewDisplay(mySurface);
camera.startPreview();
[ … ]
camera.stopPreview();
```

You'll learn more about Surfaces in the following chapter, although Android includes an excellent example of using a `SurfaceView` to display the camera preview in real time. This example is available in the `graphics/CameraPreview` project in the SDK API demos.

You can also assign a `PreviewCallback` to be fired for each preview frame, allowing you to manipulate or display each preview frame individually. Call the `setPreviewCallback` method on the `Camera` object, passing in a new `PreviewCallback` implementation overriding the `onPreviewFrame` method as shown here:

```
camera.setPreviewCallback(new PreviewCallback() {

  public void onPreviewFrame(byte[] _data, Camera _camera) {
   // TODO Do something with the preview image.
  }
});
```

Taking a Picture

Take a picture by calling `takePicture` on a `Camera` object, passing in a `ShutterCallback` and `PictureCallback` implementations for the RAW and JPEG-encoded images. Each picture callback will receive a byte array representing the image in the appropriate format, while the shutter callback is triggered immediately after the shutter is closed.

```
private void takePicture() {
  camera.takePicture(shutterCallback, rawCallback, jpegCallback);
}

ShutterCallback shutterCallback = new ShutterCallback() {
```

```
    public void onShutter() {
      // TODO Do something when the shutter closes.
    }
  };

  PictureCallback rawCallback = new PictureCallback() {
    public void onPictureTaken(byte[] _data, Camera _camera) {
      // TODO Do something with the image RAW data.
    }
  };

  PictureCallback jpegCallback = new PictureCallback() {
    public void onPictureTaken(byte[] _data, Camera _camera) {
      // TODO Do something with the image JPEG data.
    }
  };
```

Introducing the Sensor Manager

The Sensor Manager is used to manage the sensor hardware available on an Android device. Use `getSystemService` to get a reference to the Sensor Service as shown in the code snippet below:

```
String service_name = Context.SENSOR_SERVICE;
SensorManager sensorManager = (SensorManager)getSystemService(service_name);
```

The following sections look closely at how to use the Sensor Manager to monitor orientation and acceleration, but the pattern shown here can be used to monitor sensor results from any available hardware sensor:

```
SensorListener mySensorListener = new SensorListener() {
  public void onSensorChanged(int sensor, float[] values) {
    // TODO Deal with sensor value changes
  }

  public void onAccuracyChanged(int sensor, int accuracy) {
    // TODO Auto-generated method stub
  }
};
```

The `SensorListener` interface is used to listen for Sensor value and accuracy changes.

Implement the `onSensorChanged` method to react to value changes. The `sensor` parameter identifies the sensor that triggered the event, while the `values float` array contains the new values detected by that sensor.

Implement `onAccuracyChanged` to react to changes in a sensor's accuracy. The `sensor` parameter again identifies the sensor that triggered the event, while the `accuracy` parameter indicates the new accuracy of that sensor using one of the constants:

- `SensorManager.SENSOR_STATUS_ACCURACY_HIGH` Indicates that the sensor is reporting with the highest possible accuracy.
- `SensorManager.SENSOR_STATUS_ACCURACY_LOW` Indicates that the sensor is reporting with low accuracy and needs to be calibrated.

- `SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM` Indicates that the sensor data is of average accuracy, and that calibration might improve the readings.
- `SensorManager.SENSOR_STATUS_UNRELIABLE` Indicates that the sensor data is unreliable, meaning that either calibration is required or readings are not currently possible.

The Sensor Manager includes constants to help identify the sensor triggering the change event. The following list includes the sensors for which constants are currently defined. Some or all of these sensors will be available to your applications depending on the hardware available on the host device:

- `SensorManager.SENSOR_ACCELEROMETER` Is an accelerometer sensor that returns the current acceleration along three axes in meters per second squared (m/s^2). The accelerometer is explored in greater detail later in this chapter.
- `SensorManager.SENSOR_ORIENTATION` Is an orientation sensor that returns the current orientation on three axes in degrees. The orientation sensor is explored in greater detail later in this chapter.
- `SensorManager.SENSOR_LIGHT` Is an ambient-light sensor that returns a single value describing the ambient illumination in lux.
- `SensorManager.SENSOR_MAGNETIC_FIELD` Is a sensor used to determine the current magnetic field measured in microteslas (μT) along three axes.
- `SensorManager.SENSOR_PROXIMITY` Is a proximity sensor that returns a single value describing the distance between the device and the target object in meters (m).
- `SensorManager.SENSOR_TEMPERATURE` Is a thermometer sensor that returns the ambient temperature in degrees Celsius (°C).

To receive notifications of changes from a particular sensor, create a Sensor Listener as described previously, and register it with the Sensor Manager specifying the sensor that should trigger the Listener and the rate at which the sensor should update, as shown in the following code snippet:

```
sensorManager.registerListener(mySensorListener,
                               SensorManager.SENSOR_TRICORDER,
                               SensorManager.SENSOR_DELAY_FASTEST);
```

The Sensor Manager includes the following constants (shown in descending order of responsiveness) to let you select a suitable update rate:

- `SensorManager.SENSOR_DELAY_FASTEST` Specifies the fastest possible sensor update rate.
- `SensorManager.SENSOR_DELAY_GAME` Selects an update rate suitable for use in controlling games.
- `SensorManager.SENSOR_DELAY_NORMAL` Specifies the default update rate.
- `SensorManager.SENSOR_DELAY_UI` Specifies a rate suitable for updating UI features.

The rate you select is not binding; the Sensor Manager may return results faster or slower than you specify, though it will tend to be faster. To minimize the associated resource cost of using the sensor in your application you should try to select the slowest suitable rate.

An overloaded `registerListener` method is also available that applies the default (`SENSOR_DELAY_NORMAL`) update rate, as shown below:

```
sensorManager.registerListener(mySensorListener,
SensorManager.SENSOR_TRICORDER);
```

Using the Accelerometer and Compass

Input based on movement and orientation is an exciting innovation for mobile applications. It's a technique that has become possible thanks to the incorporation of compass and accelerometer sensors in modern devices.

Accelerometers and compasses are used to provide functionality based on changes in device orientation and movement. A recent trend is to use this functionality to provide alternative input techniques from more traditional touch-screen-, trackball-, and keyboard-based approaches. In recent years, these sensors have become increasingly common, having found their way into game controllers like the Nintendo Wii and mobile handsets like the Apple iPhone.

The availability of compass and accelerometer values depends on the hardware upon which your application runs. When available, they are exposed through the `SensorManager` class, allowing you to:

- Determine the current orientation of the hardware.
- Monitor for changes in orientation.
- Know which direction the user is facing.
- Monitor acceleration — changes in movement speed — in any direction: vertically, laterally, or longitudinally.

This opens some intriguing possibilities for your applications. By monitoring orientation, direction, and movement, you can:

- Use the compass and accelerometer to determine your speed and direction. Used with the maps and location-based services, you can create interfaces that incorporate direction and movement as well as location.
- Create User Interfaces that adjust dynamically to suit the orientation of your device. Android already alters the native screen orientation when the device is rotated from portrait to landscape or vice versa.
- Monitor for rapid acceleration to detect if a device has been dropped or thrown.
- Measure movement or vibration. For example, you could create an application that lets you lock your device; if any movement is detected while it's locked, it could send an alert IM message that includes its current location.
- Create User Interface controls that use physical gestures and movement as input.

Introducing Accelerometers

Accelerometers, as their name suggests, are used to measure acceleration.

Acceleration is defined as the rate of change of velocity, so they measure how quickly the speed of the device is changing in a given direction. Using an accelerometer, you can detect movement and, more usefully, the rate of change of the speed of that movement.

> *It's important to note that accelerometers do* not *measure velocity, so you can't measure speed directly based on a single accelerometer reading. Instead, you need to measure changes in acceleration over time.*

Generally, you'll be interested in acceleration changes relative to a rest state, or rapid movement (signified by rapid changes in acceleration) such as gestures used for user input. In the former case, you'll often need to calibrate the device to calculate the initial orientation and acceleration to take those effects into account in future results.

> *Accelerometers are unable to differentiate between acceleration due to movement and gravity. As a result, an accelerometer detecting acceleration on the Z-axis (up/down) will read –9.8 m/s² when it's at rest (this value is available as the* `SensorManager.STANDARD_GRAVITY` *constant).*

Detecting Acceleration Changes

Acceleration can be measured along three directional axes: forward–backward (longitudinal), left–right (lateral), and up–down (vertical). The Sensor Manager reports sensor changes in all three directions (as illustrated in Figure 10-1):

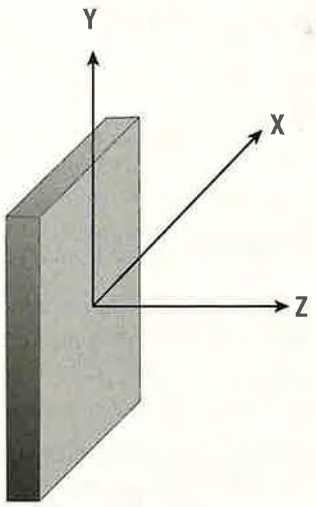


Figure 10-1

- **Vertical** Upward or downward, where positive represents upward movement such as the device being lifted up.
- **Longitudinal** Forward or backward acceleration, where forward acceleration is positive. This represents a device flat on its back, facing up, and in portrait orientation being moved along the desk in the direction of the top of the device.
- **Lateral** Sideways (left or right) acceleration, where positive values represent movement toward the right of the device, and negative values show movement toward the left. In the same configuration as described in longitudinal movement, positive lateral movement would be created by moving the device along the surface to your right.

The Sensor Manager considers the device "at rest" when it is sitting face up on a flat surface in portrait orientation.

As described previously, you can monitor changes in acceleration using Sensor Listeners. Register an extension of the `SensorListener` class with the Sensor Manager, using the `SENSOR_ACCELEROMETER` flag to request updates of accelerometer values and a sensor update rate as shown in the following code snippet:

```
SensorManager sm = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
sm.registerListener(mySensorListener,
                    SensorManager.SENSOR_ACCELEROMETER,
                    SensorManager.SENSOR_DELAY_UI);
```

Your Sensor Listener must implement the `onSensorChanged` method that will be triggered when the changes in acceleration along any of the three axes described previously are detected.

The `onSensorChanged` method receives a `float` array that contains the current acceleration along all three axes in smoothed and raw formats. The Sensor Manager includes index constants that you can use to extract the acceleration value you require, as shown in the following code snippet:

```
SensorListener mySensorListener = new SensorListener() {
  public void onSensorChanged(int sensor, float[] values) {
    if (sensor == SensorManager.SENSOR_ACCELEROMETER) {
      float xAxis = values[SensorManager.DATA_X];
      float yAxis = values[SensorManager.DATA_Y];
      float zAxis = values[SensorManager.DATA_Z];

      float raw_xAxis = values[SensorManager.RAW_DATA_X];
      float raw_yAxis = values[SensorManager.RAW_DATA_Y];
      float raw_zAxis = values[SensorManager.RAW_DATA_Z];

      // TODO apply the acceleration changes to your application.
    }
  }

  public void onAccuracyChanged(int sensor, int accuracy) { }
};
```

Creating a Speedometer

While an accelerometer won't tell you your current speed, you can calculate a rough estimate by monitoring changes in acceleration over time. In the following example, you'll create a simple speedometer using the accelerometers to determine the current speed based on acceleration changes.

The sensitivity and responsiveness of the hardware accelerometers will limit the accuracy and effectiveness of this application, but the techniques it uses should give you a better understanding of how to use the accelerometer sensors for something more useful.

Because accelerometers measure the change in velocity in a given direction, you can establish your current speed by determining how long each acceleration value has been applied. For those mathematically inclined, you're finding the second derivative of the acceleration changes.

For example, if you accelerate at a steady rate of 1 m/s^2 after 10 seconds, your speed will be 10 m/s (or 36 km/h). When your speed becomes steady, your acceleration should return to zero. In the real world, acceleration rarely stops and starts in an instant, nor does it remain constant, so you'll need to adjust your velocity calculations as the measured acceleration changes.

1. Start by creating a new `Speedometer` project with a Speedometer Activity. Modify the main.xml layout resource to display a single, centered line of large, bold text that will be used to display your current speed.

```
<?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:gravity="center"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textStyle="bold"
    android:textSize="40sp"
    android:text="CENTER"
    android:editable="false"
    android:singleLine="true"
    android:layout_margin="10px"/>
  />
</LinearLayout>
```

2. Within the Speedometer Activity, create instance variables to store references to the `TextView` and the `SensorManager`. Also create variables to record the current acceleration, velocity, and the last update time.

```
SensorManager sensorManager;
TextView myTextView;

float appliedAcceleration = 0;
float currentAcceleration = 0;
float velocity = 0;
Date lastUpdate;
```

3. Create a new `updateVelocity` method that calculates the velocity change since the last update based on the current acceleration.

```
private void updateVelocity() {
  // Calculate how long this acceleration has been applied.
  Date timeNow = new Date(System.currentTimeMillis());
  long timeDelta = timeNow.getTime()-lastUpdate.getTime();
  lastUpdate.setTime(timeNow.getTime());

  // Calculate the change in velocity at the
  // current acceleration since the last update.
  float deltaVelocity = appliedAcceleration * (timeDelta/1000);
  appliedAcceleration = currentAcceleration;

  // Add the velocity change to the current velocity.
  velocity += deltaVelocity;

  // Convert from meters per second to miles per hour.
  double mph = (Math.round(velocity / 1.6 * 3.6));

  myTextView.setText(String.valueOf(mph) + "mph");
}
```

4. Create a new `SensorListener` implementation that updates the current acceleration (and derived velocity) whenever a change in acceleration is detected.

   Because a speedometer will most likely be used while the device is mounted vertically, with the screen face perpendicular to the ground, measure negative acceleration along the Z-axis.

```
private final SensorListener sensorListener = new SensorListener() {

double calibration - Double.NAN;

  public void onSensorChanged(int sensor, float[] values) {
    double x = values[SensorManager.DATA_X];
    double y = values[SensorManager.DATA_Y];
    double z = values[SensorManager.DATA_Z];
    double a = -1*Math.sqrt(Math.pow(x, 2) +
                            Math.pow(y, 2) +
                            Math.pow(z, 2));
    if (calibration == Double.NaN)
      calibration = a;
    else {
      updateVelocity();
      currentAcceleration = (float)a;
    }
  }

  public void onAccuracyChanged(int sensor, int accuracy) {}

};
```

5. Update the `onCreate` method to register your new Listener for accelerometer updates using the `SensorManager`. Take the opportunity to get a reference to the Text View.

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

```
    super.onCreate(icicle);
    setContentView(R.layout.main);

    myTextView = (TextView)findViewById(R.id.myTextView);
    lastUpdate = new Date(System.currentTimeMillis());

    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    sensorManager.registerListener(sensorListener,
                                   SensorManager.SENSOR_ACCELEROMETER,
                                   SensorManager.SENSOR_DELAY_FASTEST);
  }
```

6. Create a new `Timer` that updates the speed based on the current acceleration every second. Because this will update a GUI element, you'll need to create a new `updateGUI` method that synchronizes with the GUI thread using a `Handler` before updating the Text View.

```
  Handler handler = new Handler();

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

    myTextView = (TextView)findViewById(R.id.myTextView);
    lastUpdate = new Date(System.currentTimeMillis());

    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    sensorManager.registerListener(sensorListener,
                                   SensorManager.SENSOR_ACCELEROMETER,
                                   SensorManager.SENSOR_DELAY_FASTEST);

    Timer updateTimer = new Timer("velocityUpdate");
    updateTimer.scheduleAtFixedRate(new TimerTask() {
                                      public void run() {
                                        updateGUI();
                                      }
                                    }, 0, 1000);
  }

  private void updateGUI()}
    //Convert from m/s to mph
    final double mph = (Math.round(100*velocity / 1.6*3.6)) / 100;

    //Update the GUI
    handler.post(new Runnable() {
      public void run() {
        myTextView.setText(String.valueOf(mph) + "mph");
      }
    });
  }
```

Once you're finished, you'll want to test this out. Given that keeping constant watch on your handset while, driving, cycling, or running is a Bad Idea, you should consider some further enhancements to the speedometer before you take it on the road.

Consider incorporating vibration or media player functionality to shake or beep with intensity proportional to your current speed, or simply log speed changes as they happen for later review.

If you're feeling particularly adventurous, consider integrating your speedometer into a map to track your speed along a journey using different colored lines to represent your speed along the way.

Determining Your Orientation

The *orientation sensors* are a combination of a built-in compass that provides the yaw (heading) and the accelerometers that help determine pitch and roll.

If you've done a bit of trigonometry, you've got the skills required to calculate the device orientation based on the accelerometer values along the three axes. If you enjoyed trig as much as I did, you'll be happy to learn that Android does these calculations for you.

The device orientation is reported along all three dimensions, as illustrated in Figure 10-2:

- **Heading** The heading (also bearing or yaw) is the direction the device is facing around the Z-axis, where 0/360 degrees is North, 90 degrees is East, 180 degrees is South, and 270 degrees is West.
- **Pitch** Pitch represents the angle of the device around the Y-axis. The tilt angle returned shows 0 degrees when the device is flat on its back, –90 degrees when standing upright (top of device pointing at the ceiling), 90 degrees when the device is upside down, and 180/–180 degrees when the device is face down.
- **Roll** The roll represents the device's sideways tilt between –90 and 90 degrees on the X-axis. The tilt is 0 degrees when the device is flat on its back, –90 degrees when the screen faces left, and 90 degrees when the screen faces right.

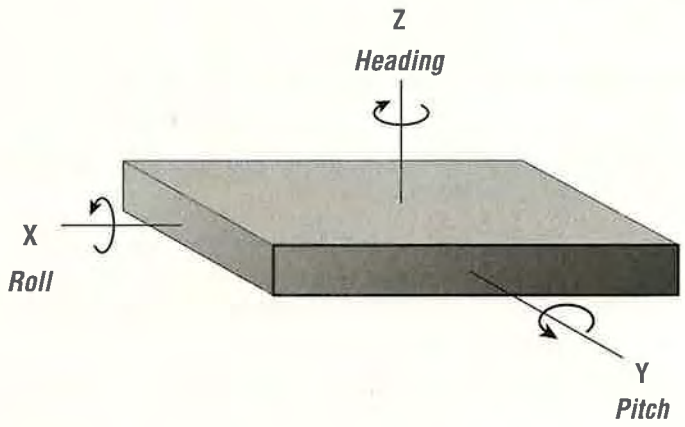


Figure 10-2

As implied by the preceding list, the Sensor Manager considers the device at rest (heading, pitch, roll at 0 degrees) when it is flat on its back. To monitor device orientation, register a Sensor Listener with the Sensor Manager, specifying the `SENSOR_ORIENTATION` flag, as shown in the following code snippet:

```
SensorManager sm = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
sm.registerListener(myOrientationListener,
                    SensorManager.SENSOR_ORIENTATION,
                    SensorManager.SENSOR_DELAY_NORMAL);
```

The `onSensorChanged` method in your `SensorListener` implementation will receive a `float` array containing the current orientation, along the three axes described above, whenever the device's orientation changes.

Within this `float` array, use the Sensor Manager constants `DATA_X`, `DATA_Y`, and `DATA_Z` to find the roll, pitch, and heading (yaw) respectively. Use the corresponding `RAW_DATA_*` constants to find the unsmoothed / unfiltered values as shown in the following code snippet:

```
SensorListener myOrientationListener = new SensorListener() {
  public void onSensorChanged(int sensor, float[] values) {
    if (sensor == SensorManager.SENSOR_ORIENTATION) {
      float rollAngle = values[SensorManager.DATA_X];
      float pitchAngle = values[SensorManager.DATA_Y];
      float headingAngle =  values[SensorManager.DATA_Z];

      float raw_rollAngle = values[SensorManager.RAW_DATA_X];
      float raw_pitchAngle= values[SensorManager.RAW_DATA_Y];
      float raw_headingAngle = values[SensorManager.RAW_DATA_Z];
      // TODO apply the orientation changes to your application.
    }
  }

  public void onAccuracyChanged(int sensor, int accuracy) { }
};
```

Creating a Compass and Artificial Horizon

In Chapter 4, you created a simple `CompassView` to experiment with owner-drawn controls. In this example, you'll extend the functionality of the `CompassView` to display the device pitch and roll, before using it to display the current device orientation.

1. Open the Compass project you created in Chapter 4. You will be making changes to the `CompassView` as well as the Compass Activity used to display it. To ensure that the View and controller remain as decoupled as possible, the `CompassView` won't be linked to the sensors directly; instead, it will be updated by the Activity.

   Start by adding field variables and get/set methods for pitch and roll to the `CompassView`.

   ```
   float pitch = 0;
   float roll = 0;


   public float getPitch() {
     return pitch;
   }
   ```

```
public void setPitch(float pitch) {
  this.pitch = pitch;
}

public float getRoll() {
  return roll;
}
public void setRoll(float roll) {
  this.roll = roll;
}
```

2. Update the `onDraw` method to include two circles that will be used to indicate the pitch and roll values.

```
@Override
protected void onDraw(Canvas canvas) {

  [ … Existing onDraw method … ]
```

2.1. Create a new circle that's half-filled and rotates in line with the sideways tilt.

```
RectF rollOval = new RectF((getMeasuredWidth()/3)-getMeasuredWidth()/7,
                           (getMeasuredHeight()/2)-getMeasuredWidth()/7,
                           (getMeasuredWidth()/3)+getMeasuredWidth()/7,
                           (getMeasuredHeight()/2)+getMeasuredWidth()/7
                          );
markerPaint.setStyle(Paint.Style.STROKE);
canvas.drawOval(rollOval, markerPaint);
markerPaint.setStyle(Paint.Style.FILL);
canvas.save();
canvas.rotate(roll, getMeasuredWidth()/3, getMeasuredHeight()/2);
canvas.drawArc(rollOval, 0, 180, false, markerPaint);

canvas.restore();
```

2.2. Create a new circle that starts half-filled and varies between full and empty based on the current pitch.

```
RectF pitchOval = new RectF((2*mMeasuredWidth/3)-mMeasuredWidth/7,
                            (getMeasuredHeight()/2)-getMeasuredWidth()/7,
                            (2*getMeasuredWidth()/3)+getMeasuredWidth()/7,
                            (getMeasuredHeight()/2)+getMeasuredWidth()/7
                           );
markerPaint.setStyle(Paint.Style.STROKE);
canvas.drawOval(pitchOval, markerPaint);
markerPaint.setStyle(Paint.Style.FILL);
canvas.drawArc(pitchOval, 0-pitch/2, 180+(pitch), false, markerPaint);
markerPaint.setStyle(Paint.Style.STROKE);
}
```

That completes the changes to the `CompassView`. If you run the application now, it should appear as shown in Figure 10-3. This screen capture was taken while the device was facing due East, with a 45-degree roll and 10-degree backward pitch.

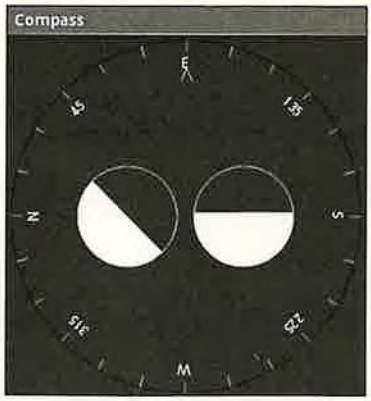


Figure 10-3

1. Now you'll be updating the `Compass` Activity to use the Sensor Manager to listen for orientation changes and pass them through to the `CompassView`. Start by adding local field variables to store the current roll, pitch, and heading as well as references to the `CompassView` and `SensorManager`.

```
float pitch = 0;
float roll = 0;
float heading = 0;

CompassView compassView;
SensorManager sensorManager;
```

2. Create a new `updateOrientation` method that takes new heading, pitch, and roll values to update the field variables and apply them to the `CompassView`.

```
private void updateOrientation(float _roll,
                               float _pitch,
                               float _heading) {
  heading = _heading;
  pitch = _pitch;
  roll = _roll;

  if (compassView!= null) {
    compassView.setBearing(heading);
    compassView.setPitch(pitch);
    compassView.setRoll(roll);
    compassView.invalidate();
  }
}
```

3. Update the `onCreate` method to get references to the `CompassView` and `SensorManager`, as well as initializing the heading, pitch, and roll values.

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

```
    setContentView(R.layout.main);

    compassView = (CompassView)this.findViewById(R.id.compassView);
    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    updateOrientation(0, 0, 0);
  }
```

4. Create a new field variable that instantiates a new `SensorListener` implementation that calls the `updateOrientation` method.

```
  private final SensorListener sensorListener = new SensorListener() {

    public void onSensorChanged(int sensor, float[] values) {
      updateOrientation(values[SensorManager.DATA_X],
                        values[SensorManager.DATA_Y],
                        values[SensorManager.DATA_Z]);
    }

    public void onAccuracyChanged(int sensor, int accuracy) {}

  };
```

5. Then override the `onResume` method to register the `SensorListener` to listen for orientation changes when the Activity is visible. Also override `onStop` to prevent updates when the Activity has been suspended.

```
  @Override
  protected void onResume()
  {
    super.onResume();
    sensorManager.registerListener(sensorListener,
                                   SensorManager.SENSOR_ORIENTATION,
                                   SensorManager.SENSOR_DELAY_FASTEST);
  }

  @Override
  protected void onStop()
  {
    sensorManager.unregisterListener(sensorListener);
    super.onStop();
  }
```

If you run the application now, you should see the three face dials update dynamically when the orientation of the device changes. Unfortunately, it's not currently possible to emulate the sensor hardware, so this application will only update when running on supported hardware.

Android Telephony

The telephony APIs let your applications access the underlying telephone hardware, making it possible to create your own dialer — or integrate call handling and phone state monitoring into your applications.

Due to security concerns, the current Android SDK does not allow you to create your own "in call" application — the screen that is displayed when an incoming call is received or an outgoing call has been initiated.

Rather than creating a new dialer implementation, the following sections focus on how to monitor and control phone, service, and cell events in your applications to augment and manage the native phone-handling functionality.

Making Phone Calls

The best practice is to use Intents to launch a dialer application to initiate new phone calls. There are two Intent actions you can use to dial a number; in both cases, you should specify the number to dial using the `tel:` schema as the data component of the Intent:

- `Intent.ACTION_CALL` Automatically initiates the call, displaying the in-call application. You should only use this action if you are replacing the native dialer, otherwise you should use the `ACTION_DIAL` action as described below. Your application must have the `CALL_PHONE` permission granted to broadcast this action.
- `Intent.ACTION_DIAL` Rather than dial the number immediately, this action starts a dialer application, passing in the specified number but allowing the dialer application to manage the call initialization (the default dialer asks the user to explicitly initiate the call). This action doesn't require any permissions and is the standard way applications should initiate calls.

The following skeleton code shows the basic technique for dialing a number:

```
Intent intent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel:1234567"));
startActivity(intent);
```

By using Intents to announce your intention to dial a number, your application can remain fully decoupled from a particular hardware implementation used to initiate the call. For example, should you replace the existing dialer with a hybrid that allows IP-based telephony, using Intents to dial a number from other applications will let you leverage that new functionality without needing to change each application.

Monitoring Phone State and Phone Activity

The Android telephony API lets you monitor phone state, retrieve incoming phone numbers, and manage calls. Access to the telephony APIs is managed by the Telephony Manager, accessible using the `getSystemService` method as shown below:

```
String srvcName = Context.TELEPHONY_SERVICE;
TelephonyManager telephonyManager =
  (TelephonyManager)getSystemService(srvcName);
```

In order to monitor and manage phone state, your application will need a `READ_PHONE_STATE` permission, as shown in the following XML code snippet:

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

Changes to the phone state are announced to other components using the `PhoneStateListener` class. Extend the `PhoneStateListener` to listen for, and respond to, phone state change events including call state (ringing, off hook, etc.), cell location changes, voice-mail and call-forwarding status, phone service changes, and changes in mobile signal strength.

To react to phone state change events, create a new Phone State Listener implementation, and override the event handlers of the events you want to react to. Each handler receives parameters that indicate the new phone state, such as the current cell location, call state, or signal strength.

The following code highlights the available state change handlers in a skeleton Phone State Listener implementation:

```
PhoneStateListener phoneStateListener = new PhoneStateListener() {
  public void onCallForwardingIndicatorChanged(boolean cfi) {}
  public void onCallStateChanged(int state, String incomingNumber) {}
  public void onCellLocationChanged(CellLocation location) {}
  public void onDataActivity(int direction) {}
  public void onDataConnectionStateChanged(int state) {}
  public void onMessageWaitingIndicatorChanged(boolean mwi) {}
  public void onServiceStateChanged(ServiceState serviceState) {}
  public void onSignalStrengthChanged(int asu) {}
};
```

Once you've created your own Phone State Listener, register it with the Telephony Manager using a bit-mask to indicate the events you want to listen for, as shown in the following code snippet:

```
telephonyManager.listen(phoneStateListener,
                        PhoneStateListener.LISTEN_CALL_FORWARDING_INDICATOR |
                        PhoneStateListener.LISTEN_CALL_STATE |
                        PhoneStateListener.LISTEN_CELL_LOCATION |
                        PhoneStateListener.LISTEN_DATA_ACTIVITY |
                        PhoneStateListener.LISTEN_DATA_CONNECTION_STATE |
                        PhoneStateListener.LISTEN_MESSAGE_WAITING_INDICATOR |
                        PhoneStateListener.LISTEN_SERVICE_STATE |
                        PhoneStateListener.LISTEN_SIGNAL_STRENGTH);
```

To unregister a listener, call `listen` and pass in `PhoneStateListener.LISTEN_NONE` as the bit field parameter, as shown below:

```
telephonyManager.listen(phoneStateListener, PhoneStateListener.LISTEN_NONE);
```

In the following sections, you'll learn how to use the Phone State Listener to monitor incoming calls, track cell location changes, and monitor service changes.

Monitoring Phone Calls

One of the most popular reasons for monitoring phone state changes is to detect, and react to, incoming and outgoing phone calls.

Calls can be detected through changes in the phone's call state. Override the `onCallStateChanged` method in a Phone State Listener implementation, and register it as shown below to receive notifications when the call state changes:

```
PhoneStateListener callStateListener = new PhoneStateListener() {
  public void onCallStateChanged(int state, String incomingNumber) {
    // TODO React to incoming call.
  }
};

telephonyManager.listen(callStateListener,
PhoneStateListener.LISTEN_CALL_STATE);
```

The `onCallStateChanged` handler receives the phone number associated with incoming calls, and the state parameter represents the current call state as one of the following three values:

- `TelephonyManager.CALL_STATE_IDLE` When the phone is neither ringing nor in a call
- `TelephonyManager.CALL_STATE_RINGING` When the phone is ringing
- `TelephonyManager.CALL_STATE_OFFHOOK` If the phone is currently on a call

Tracking Cell Location Changes

You can get notifications whenever the current cell location changes by overriding `onCellLocationChanged` on a Phone State Listener implementation. Before you can register to listen for cell location changes, you need to add the `ACCESS_COARSE_LOCATION` permission to your application manifest.

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

The `onCellLocationChanged` handler receives a `CellLocation` object that includes methods for extracting the cell ID (`getCid`) and the current LAC (`getLac`).

The following code snippet shows how to implement a Phone State Listener to monitor cell location changes, displaying a Toast that includes the new location's cell ID:

```
PhoneStateListener cellLocationListener = new PhoneStateListener() {
  public void onCellLocationChanged(CellLocation location) {
    GsmCellLocation gsmLocation = (GsmCellLocation)location;
    Toast.makeText(getApplicationContext(),
                   String.valueOf(gsmLocation.getCid()),
                   Toast.LENGTH_LONG).show();
  }
};
telephonyManager.listen(cellLocationListener,
                        PhoneStateListener.LISTEN_CELL_LOCATION);
```

Tracking Service Changes

The `onServiceStateChanged` handler tracks the service details for the device's cell service. Use the `ServiceState` parameter to find details of the current service state.

The `getState` method on the Service State object returns the current service state as one of:

- `ServiceState.STATE_IN_SERVICE` Normal phone service is available.
- `ServiceState.STATE_EMERGENCY_ONLY` Phone service is available but only for emergency calls.
- `ServiceState.STATE_OUT_OF_SERVICE` No cell phone service is currently available.
- `ServiceState.STATE_POWER_OFF` The phone radio is turned off (usually when airplane mode is enabled).

A series of `getOperator*` methods is available to retrieve details on the operator supplying the cell phone service, while `getRoaming` tells you if the device is currently using a roaming profile.

The following example shows how to register for service state changes and displays a Toast showing the operator name of the current phone service:

```
PhoneStateListener serviceStateListener = new PhoneStateListener() {
  public void onServiceStateChanged(ServiceState serviceState) {
    if (serviceState.getState() == ServiceState.STATE_IN_SERVICE) {
      String toastText = serviceState.getOperatorAlphaLong();
      Toast.makeText(getApplicationContext(), toastText, Toast.LENGTH_SHORT);
    }
  }
};

telephonyManager.listen(serviceStateListener,
                        PhoneStateListener.LISTEN_SERVICE_STATE);
```

Monitoring Data Connectivity and Activity

As well as voice and service details, you can monitor changes in mobile data connectivity and mobile data transfer by implementing a `PhoneStateListener`.

The Phone State Listener includes two event handlers for monitoring the device data connection. Override `onDataActivity` to track data transfer activity, and `onDataConnectionStateChanged` to request notifications for data connection state changes.

The following skeleton code shows both handlers overridden, with switch statements demonstrating each of the possible values for the `state` and `direction` parameters passed in to each event:

```
PhoneStateListener dataStateListener = new PhoneStateListener() {
  public void onDataActivity(int direction) {
    switch (direction) {
      case TelephonyManager.DATA_ACTIVITY_IN : break;
      case TelephonyManager.DATA_ACTIVITY_OUT : break;
      case TelephonyManager.DATA_ACTIVITY_INOUT : break;
      case TelephonyManager.DATA_ACTIVITY_NONE : break;
    }
  }

  public void onDataConnectionStateChanged(int state) {
```

```
        switch (state) {
          case TelephonyManager.DATA_CONNECTED : break;
          case TelephonyManager.DATA_CONNECTING : break;
          case TelephonyManager.DATA_DISCONNECTED : break;
          case TelephonyManager.DATA_SUSPENDED : break;
        }
      }
    };

    telephonyManager.listen(dataStateListener,
                            PhoneStateListener.LISTEN_DATA_ACTIVITY |
                            PhoneStateListener.LISTEN_DATA_CONNECTION_STATE);
```

Accessing Phone Properties and Status

The Telephony Manager also provides access to several static phone properties. You can obtain the current value of any of the phone state details described previously. The following code snippet shows how to extract the current incoming call number if the phone is ringing:

```
String incomingCall = null;
if (telephonyManager.getCallState() == TelephonyManager.CALL_STATE_RINGING)
  incomingCall = telephonyManager.getLine1Number();
```

You can also access SIM and network operator details, network information, and voice-mail details. The following code snippet shows the framework used to access the current network details:

```
String srvcName = Context.TELEPHONY_SERVICE;
TelephonyManager telephonyManager = (TelephonyManager)getSystemService(srvcName);

String networkCountry = telephonyManager.getNetworkCountryIso();
String networkOperatorId = telephonyManager.getNetworkOperator();
String networkName = telephonyManager.getNetworkOperatorName();
int networkType = telephonyManager.getNetworkType();
```

Controlling the Phone

There are times when you need access to the underlying phone hardware to effect changes rather than simply monitoring them.

> **Access to the underlying Phone hardware was removed shortly before the release of Android SDK version 1. It is expected that basic phone interaction including answering and hanging up the phone will be available in a subsequent API release.**
>
> **The following section is based on an earlier API release that included phone hardware interaction support. It has been included to serve as a guide for likely future implementations.**

The `Phone` class in Android provides this interface, letting you control hardware settings, handle incoming calls, initiate new outgoing calls, hang up calls in progress, handle conference calls, and a variety of other Phone functionalities.

Replacing the basic phone functionality is a complex process. Rather than delve into this in detail, this section focuses on some of the more useful functions that can be particularly powerful when used within applications that augment rather than replace the existing functionality.

To access the phone hardware, use the `Phone` class, available through the Telephony Manager using the `getPhone` method as shown in the following code snippet:

```
Phone phone = telephonyManager.getPhone();
```

Once you have a reference to the `Phone` object, you can initiate calls using the `call` or `dial` method or end them by calling `endCall`.

Answering, Dismissing, and Ending Calls

The following code snippet shows how to use the Phone State Listener to listen for incoming calls and reject them if they're from a particular place:

```
final String badPrefix = "+234";

PhoneStateListener callBlockListener = new PhoneStateListener() {
  public void onCallStateChanged(int state, String incomingNumber) {
    if (state == TelephonyManager.CALL_STATE_RINGING) {
      Phone phone = telephonyManager.getPhone();
      if (incomingNumber.startsWith(badPrefix)) {
        phone.endCall();
      }
    }
  }
};

telephonyManager.listen(callBlockListener,
PhoneStateListener.LISTEN_CALL_STATE);
```

Using Bluetooth

In this section, you'll learn how to interact directly with Bluetooth devices including other phones and Bluetooth headsets. Using Bluetooth, you can pair with other devices within range, initiate an `RFCOMMSocket`, and transmit and receive streams of data from or for your applications.

> **The Bluetooth libraries have been removed for the Android version 1.0 release. The following sections are based on earlier SDK releases and are included as a guide to functionality that is expected to be made available in subsequent releases.**

Introducing the Bluetooth Service

The Android Bluetooth service is represented by the `BluetoothDevice` class.

Bluetooth is a system service accessed using the `getSystemService` method. Get a reference to the `BluetoothDevice` by passing in the `Context.BLUETOOTH` constant as the service name parameter, as shown in the following code snippet:

```
String context = Context.BLUETOOTH_SERVICE;
final BluetoothDevice bluetooth = (BluetoothDevice)getSystemService(context);
```

To use the Bluetooth Service, your application needs to have the `BLUETOOTH` permission as shown here:

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

Controlling the Local Bluetooth Device

The Bluetooth Device offers several methods that let you control the Bluetooth hardware.

The `enable` and `disable` methods let you enable or disable the Bluetooth adapter. The `getName` and `setName` methods let you modify the local device name, and `getAddress` can be used to determine the local device address. You can find and change the discovery mode and discovery time-out settings using the `getMode` and `getDiscoverableTimeout` methods and their setter equivalents.

The following code snippet enables the Bluetooth adapter and waits until it has connected before changing the device name and setting the mode to "discoverable":

```
bluetooth.enable(new IBluetoothDeviceCallback.Stub() {

  public void onCreateBondingResult(String _address, int _result)
              throws RemoteException
  {
    String friendlyName = bluetooth.getRemoteName(_address);
  }

  public void onEnableResult(int _result) throws RemoteException {
    if (_result == BluetoothDevice.RESULT_SUCCESS) {
      bluetooth.setName("BLACKFANG");
      bluetooth.setMode(BluetoothDevice.MODE_DISCOVERABLE);
    }
  }

});
```

Discovering and Bonding with Bluetooth Devices

Before you can establish a data communications socket, the local Bluetooth device must first discover, connect, and bond with the remote device.

Discovery

Looking for remote devices to connect to is called *discovery*. For other devices to discover your handset, you need to set the mode to "discoverable" or "connectable" using the `setMode` method as shown previously.

To discover other devices, initiate a discovery session using the `startDiscovery` or `startPeriodicDiscovery` methods, as shown below:

```
if (discoverPeriodically)
  bluetooth.startPeriodicDiscovery();
else
  bluetooth.startDiscovery(true);
```

Both of these calls are asynchronous, broadcasting a `REMOTE_DEVICE_FOUND_ACTION` whenever a new remote Bluetooth device is discovered.

To get a list of the remote devices that have been discovered, call `listRemoteDevices` on the Bluetooth device object. The returned String array contains the address of each remote device found. You can find their "friendly" names by passing in the device address to `getRemoteName`.

Bonding

Bonding, also known as *pairing*, lets you create an authenticated connection between two Bluetooth devices using a four-digit PIN. This ensures that your Bluetooth connections aren't hijacked.

Android requires you to bond with remote devices before you can establish application-layer communication sessions such as RFCOMM.

To bond with a remote device, call the `createBonding` method on the Bluetooth Device after using `setPin` to set the unique identifier PIN to use for this pairing request. To abort the bonding attempt, call `cancelBonding` and use `cancelPin` to re-set the PIN if required.

Once a remote device has been bonded, it will be added to the native database and will automatically bond with the local device if it is discovered in the future.

In the following code snippet, the Bluetooth Service is queried for a list of all the available remote devices. The list is then checked to see if any of these devices are not yet bonded with the local Bluetooth service, in which case, pairing is initiated.

```
String[] devices = bluetooth.listRemoteDevices();

for (String device : devices) {
  if (!bluetooth.hasBonding(device)) {
    // Set the pairing PIN. In real life it's probably a smart
    // move to make this user enterable and dynamic.
    bluetooth.setPin(device, new byte[] {1,2,1,2});
    bluetooth.createBonding(device, new IBluetoothDeviceCallback.Stub() {

      public void onCreateBondingResult(String _address, int _result)
                  throws RemoteException {
        if (_result == BluetoothDevice.RESULT_SUCCESS) {
          String connectText = "Connected to " + bluetooth.getRemoteName(_address);
          Toast.makeText(getApplicationContext(), connectText, Toast.LENGTH_SHORT);
        }
```

```
        }

        public void onEnableResult(int _result) throws RemoteException {}

      });
    }
  }
```

To listen for remote-device bonding requests, implement and register a `BroadcastListener` that filters for the `ACTION_PAIRING_REQUEST` Intent.

Managing Bluetooth Connections

Calling `listRemoteDevices` returns a list of the currently discovered devices, while `listBondings` returns the address of each remote device currently bonded to the local device. As a shortcut, `hasBonding` lets you specify a device address and returns `true` if you have bonded with it.

Further details on each device can be found using the `lastSeen` and `lastUsed` methods. These methods return the last time a device was seen (through discovery) or accessed.

Use `removeBonding` to sever a bond with a remote device. This will also close any application layer communications sockets you've established.

Communication with Bluetooth

The most likely reason for bonding to a remote Bluetooth device is to communicate with it.

Bluetooth data transfer is handled using the `RfcommSocket` class, which provides a wrapper for the Bluetooth radiofrequency communications (RFCOMM) protocol that supports RS232 serial communication over an underlying Logical Link Control and Adaptation Protocol (L2CAP) layer.

In practice, this alphabet soup provides a mechanism for opening communication sockets between two paired Bluetooth devices.

In order for an RFCOMM communication channel to be established, a listening port on one device must be connected to an outgoing port on the other. As a result, for bidirectional communication, two socket connections must be established.

Opening a Socket Connection

Before you can transfer data between Bluetooth devices, you need to open a new `RFCommSocket`.

Start by creating a new `RFCommSocket` object and calling its `create` method. This constructs a new socket for you to use on your Bluetooth device, returning a `FileDescriptor` for transferring data.

The `FileDescriptor` is the lowest-level representation of an I/O source. You can create any of the I/O class objects (`FileStream`, `DataOutputStream`, etc.) using a `FileDescriptor` as a constructor parameter. Later you'll use one of these I/O classes to transfer data with a remote device.

The following skeleton code shows the basic RFCommSocket implementation that creates a new socket connection ready to either initiate or respond to communications requests:

```
FileDescriptor localFile;
RfcommSocket localSocket = new RfcommSocket();
try {
  localFile = localSocket.create();
} catch (IOException e) { }
```

Once the socket has been created, you need to either bind it to the local device to listen for connection requests or initiate a connection with a remote device.

Listening for Data

To listen for incoming data, use the `bind` method to create a socket to use as a listening port for the local device. If this is successful, use the `listen` method to open the socket to start listening for incoming data transfer requests.

Once you've initialized your listener socket, use the `accept` method to check for, and respond to, any incoming socket connection requests.

The `accept` method takes a new `RfcommSocket` object that will be used to represent the remote socket connection, and a time-out for a connection request to be received. If a successful connection is made, `accept` returns a `FileDescriptor` that represents the input stream. Use this File Descriptor to process the incoming data stream from the remote device.

The following skeleton code shows how to configure a new socket that listens for, and accepts, an incoming socket connection:

```
FileDescriptor localFile;
FileDescriptor remoteFile;

RfcommSocket localSocket = new RfcommSocket();
try {
  localFile = localSocket.create();
  localSocket.bind(null);
  localSocket.listen(1);
  RfcommSocket remotesocket = new RfcommSocket();
  remoteFile = localSocket.accept(remotesocket, 10000);
}
catch (IOException e) { }
```

If no connection request is made within the time-out period, `accept` returns null.

Transmitting Data

To transmit data using an `RfcommSocket`, use the `connect` method to specify a bonded remote device address and port number to transmit data to. The connection request can also be made asynchronously using the `connectAsync` method to initiate the connection; `waitForAsyncConnect` can then be used to block on a separate thread until a response is received.

Once a connection has been established, you can transmit data with any of the I/O output classes using the local socket's `FileDescriptor` as a constructor parameter, as shown in the following code snippet:

```
FileDescriptor localFile;
String remoteAddress = bluetooth.listBondings()[0];

RfcommSocket localSocket = new RfcommSocket();
try {
  localFile = localSocket.create();
  // Select an unused port
  if (localSocket.connect(remoteAddress, 0)) {
    FileWriter output = new FileWriter(localFile);
    output.write("Hello, Android");
    output.close();
  }
} catch (IOException e) { }
```

Using a Bluetooth Headset

Wireless headsets are one of the most common uses of Bluetooth on mobile phones.

The `BluetoothHeadset` class provides specialized support for interacting with Bluetooth headsets. In this context, a headset includes any headset or hands-free device.

To use the Bluetooth headset API, create a new `BluetoothHeadset` object on your application context, as shown in the code snippet below:

```
BluetoothHeadset headset = new BluetoothHeadset(this);
```

This object will act as a proxy to the Bluetooth Headset Service that services any Bluetooth headsets bonded with the system.

Android only supports a single headset connection at a time, but you can change the connected headset using this API. Call `connectHeadset`, passing in the address of the headset to connect to, as shown in the code snippet below:

```
headset.connectHeadset(address, new IBluetoothHeadsetCallback.Stub() {

  public void onConnectHeadsetResult(String _address, int _resultCode)
              throws RemoteException {
    if (_resultCode == BluetoothHeadset.RESULT_SUCCESS) {
      // Connected to a new headset device.
    }
  }

});
```

The Headset Service is not guaranteed to be connected to a headset at all times, so it's good practice to use the `getState` method to confirm a valid connection before performing any actions, as shown in the code snippet below:

```
if (headset.getState() == BluetoothHeadset.STATE_CONNECTED) {
  // TODO Perform actions on headset.
}
```

When you've finished interacting with the headset, you should always call `close` on the `BluetoothHeadset` object to let the proxy unbind from the underlying service:

```
BluetoothHeadset headset = new BluetoothHeadset(this);
// [ … Perform headset actions … ]
headset.close();
```

Managing Network and Wi-Fi Connections

The incredible growth of Internet services and the ubiquity of mobile devices has made mobile Internet access an increasingly prevalent feature on mobile phones.

With the speed, reliability, and cost of Internet connectivity dependent on the network technology being used (Wi-Fi, GPRS, 3G), letting your applications know and manage these connections can help to ensure that they run efficiently and responsively.

Android provides access to the underlying network state, broadcasting Intents to notify application components of changes in network connectivity and offering control over network settings and connections.

Android networking is principally handled using the `ConnectivityManager`, a Service that lets you monitor the connectivity state, set your preferred network connection, and manage connectivity failover.

Later you'll learn how to use the `WifiManager` to monitor and control the device's Wi-Fi connectivity specifically. The Wi-Fi Manager lets you see and modify the configured Wi-Fi networks, manage the active connection, and perform access point scans.

Monitoring and Managing Your Internet Connectivity

The `ConnectivityManager` represents the Network Connectivity Service. It's used to monitor the state of network connections, configure failover settings, and control the network radios.

To access the Connectivity Manager, call `getSystemService`, passing in `Context.CONNECTIVITY_SERVICE` as the service name, as shown in the code snippet below:

```
String service = Context.CONNECTIVITY_SERVICE;
ConnectivityManager connectivity = (ConnectivityManager)getSystemService(service);
```

Before it can use the Connectivity Manager, your application will need Read and Write network state access permissions to be added to the manifest, as shown below:

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

Managing Active Connections

The Connectivity Manager provides a high-level view of the available network connections. Using the `getActiveNetworkInfo` or `getNetworkInfo` methods, you query `NetworkInfo` objects that include details on the currently active network or on an inactive network of the type specified.

In both cases, the `NetworkInfo` returned includes methods that indicate the connection status and network type of the specified network.

Configuring Network Preferences and Controlling Hardware

The Connectivity Manager can also be used to control network hardware and configure failover preferences.

Android will attempt to connect to the preferred network whenever an authorized application requests an Internet connection. You can set the preferred network using the `setNetworkPreference` method and specifying the network you would prefer to connect to, as shown in the code snippet below:

```
connectivity.setNetworkPreference(NetworkPreference.PREFER_WIFI);
```

If the preferred connection is unavailable, or connectivity on this network is lost, Android will automatically attempt to connect to the secondary network.

You can control the availability of the network types, using the `setRadio` method. This method lets you set the state of the radio associated with a particular network (Wi-Fi, mobile, etc.). For example, in the following code snippet, the Wi-Fi radio is turned off and the mobile radio is turned on:

```
connectivity.setRadio(NetworkType.WIFI, false);
connectivity.setRadio(NetworkType.MOBILE, true);
```

Monitoring Network Connectivity

One of the most useful functions of the Connectivity Manager is to notify applications of changes in network connectivity.

To monitor network connectivity, create your own Broadcast Receiver implementation that listens for `ConnectivityManager.CONNECTIVITY_ACTION` Intents. Such Intents include several extras that provide additional details on the change to the connectivity state:

- `ConnectivityManager.EXTRA_IS_FAILOVER` Is a Boolean that returns `true` if the current connection is the result of a failover from a preferred network.
- `ConnectivityManager.EXTRA_NO_CONNECTIVITY` Is a Boolean that returns `true` if the device is not connected to any network.

- `ConnectivityManager.EXTRA_REASON` If this broadcast represents a connection failure, this string value includes a description of why the connection attempt failed.
- `ConnectivityManager.EXTRA_NETWORK_INFO` This returns a `NetworkInfo` object containing more fine-grained details on the network associated with the current connectivity event.
- `ConnectivityManager.EXTRA_OTHER_NETWORK_INFO` After a network disconnection, this value will return a `NetworkInfo` object populated with the details for the possible failover network connection.
- `ConnectivityManager.EXTRA_EXTRA_INFO` Contains additional network-specific extra connection details.

Android SDK beta 0.9 included a NetworkConnectivityListener that encapsulated this functionality. This class has been removed for version 1.0.

Managing Your Wi-Fi

The `WifiManager` represents the Android Wi-Fi Connectivity Service. It can be used to configure Wi-Fi network connections, manage the current Wi-Fi connection, scan for access points, and monitor changes in Wi-Fi connectivity.

As with the Connectivity Manager, access the Wi-Fi Manager using the `getSystemService` method, passing in the `Context.WIFI_SERVICE` constant, as shown in the following code snippet:

```
String service = Context.WIFI_SERVICE;
final WifiManager wifi = (WifiManager)getSystemService(service);
```

To use the Wi-Fi Manager, your application must have `uses-permissions` for Read/Write Wi-Fi state access included in its manifest.

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

You can use the Wi-Fi Manager to enable or disable your Wi-Fi hardware using the `setWifiEnabled` method, or request the current Wi-Fi state using the `getWifiState` or `isWifiEnabled` methods as shown in the code snippet below:

```
if (!wifi.isWifiEnabled())
  if (wifi.getWifiState() != WifiManager.WIFI_STATE_ENABLING)
    wifi.setWifiEnabled(true);
```

The following sections begin with tracking the current Wi-Fi connection status and monitoring changes in signal strength. Later you'll also learn how to scan for and connect to specific access points.

While these functions may be sufficient for most developers, the `WifiManager` also provides low-level access to the Wi-Fi network configurations, giving you full control over each configuration setting and allowing you to completely replace the native Wi-Fi management application. Later in the section, you'll be introduced to the API used to create, delete, and modify network configurations.

Monitoring Wi-Fi Connectivity

The Wi-Fi Manager broadcasts Intents whenever the connectivity status of the Wi-Fi network changes, using the following actions:

- `WifiManager.WIFI_STATE_CHANGED_ACTION` Indicates that the Wi-Fi hardware status has changed, moving between enabling, enabled, disabling, disabled, and unknown. It includes two extra values keyed on `EXTRA_WIFI_STATE` and `EXTRA_PREVIOUS_STATE` that provide the previous and new Wi-Fi states.
- `WifiManager.SUPPLICANT_CONNECTION_CHANGE_ACTION` This Intent is broadcast whenever the connection state with the active supplicant (access point) changes. It is fired when a new connection is established or an existing connection is lost using the `EXTRA_NEW_STATE` Boolean extra that returns `true` in the former case.
- `WifiManager.NETWORK_STATE_CHANGED_ACTION` The network state change broadcast is fired whenever the Wi-Fi connectivity state changes. This Intent includes two extras — the first `EXTRA_NETWORK_INFO` includes a `NetworkInfo` object that details the current network state, while the second `EXTRA_BSSID` includes the BSSID of the access point you're connected to.
- `WifiManager.RSSI_CHANGED_ACTION` You can monitor the current signal strength of the connected Wi-Fi network by listening for the `RSSI_CHANGED_ACTION` Intent. This Broadcast Intent includes an integer extra, `EXTRA_NEW_RSSI`, that holds the current signal strength. To use this signal strength, you should use the `calculateSignalLevel` static method on the Wi-Fi Manager to convert it to an integer value on a scale you specify.

Creating and Managing Wi-Fi Connections and Configurations

You can use the Wi-Fi Manager to manage the configured network settings and control which networks to connect to. Once connected, you can interrogate the active network connection to get additional details of its configuration and settings.

Get a list of the current network configurations using `getConfiguredNetworks`. The list of `WifiConfiguration` objects returned includes the network ID, SSID, and other details for each configuration.

To use a particular network configuration, use the `enableNetwork` method, passing in the network ID to use and specifying `true` for the `disableAllOthers` parameter as shown below:

```
// Get a list of available configurations
List<WifiConfiguration> configurations = wifi.getConfiguredNetworks();
// Get the network ID for the first one.
if (configurations.size() > 0) {
  int netID = configurations.get(0).networkId;
  // Enable that network.
  boolean disableAllOthers = true;
  wifi.enableNetwork(netID, disableAllOtherstrue);
}
```

Once an active network connection has been established, use the `getConnectionInfo` method to return information on the active connection's status. The returned `WifiInfo` object includes the BSSID, Mac address, and IP address of the current access point, as well as the current link speed and signal strength.

The following code snippet queries the currently active Wi-Fi connection and displays a Toast showing the connection speed and signal strength:

```
WifiInfo info = wifi.getConnectionInfo();
if (info.getBSSID() != null) {
  int strength = WifiManager.calculateSignalLevel(info.getRssi(), 5);
  int speed = info.getLinkSpeed();
  String units = WifiInfo.LINK_SPEED_UNITS;
  String ssid = info.getSSID();

  String toastText = String.format("Connected to {0} at {1}{2}. Strength {3}/5",
                                   ssid, speed, units, strength);

  Toast.makeText(this, toastText, Toast.LENGTH_LONG);
}
```

Scanning for Hotspots

You can use the Wi-Fi Manager to conduct access point scans using the `startScan` method.

An Intent with the `SCAN_RESULTS_AVAILABLE_ACTION` action will be broadcast to asynchronously announce that the scan is complete and results are available.

Call `getScanResults` to get those results as a list of `ScanResult` objects.

Each `ScanResult` includes the details retrieved for each access point detected, including link speed, signal strength, SSID, and the authentication techniques supported.

The following skeleton code shows how to initiate a scan for access points that displays a Toast indicating the total number of access points found and the name of the Access Point with the strongest signal:

```
// Register a broadcast receiver that listens for scan results.
registerReceiver(new BroadcastReceiver() {

  @Override
  public void onReceive(Context context, Intent intent) {
    List<ScanResult> results = wifi.getScanResults();
    ScanResult bestSignal = null;
    for (ScanResult result : results) {
      if (bestSignal == null ||
          WifiManager.compareSignalLevel(bestSignal.level, result.level) < 0)
        bestSignal = result;
    }

    String toastText = String.format("{0} networks found. {1} is the strongest.",
                                     results.size(), bestSignal.SSID);

    Toast.makeText(getApplicationContext(), toastText, Toast.LENGTH_LONG);
  }

}, new IntentFilter(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));

// Initiate a scan.
wifi.startScan();
```

Managing Wi-Fi Network Configurations

To connect to a Wi-Fi network, you need to create and register a configuration. Normally your users would do this using the native Wi-Fi configuration settings, but there's no reason you can't expose the same functionality within your own applications, or for that matter replace the native Wi-Fi configuration Activity entirely.

Network configurations are stored as `WifiConfiguration` objects. The following is a non-exhaustive list of some of the public fields available for each Wi-Fi Configuration:

- `BSSID` Specifies the BSSID for an access point.
- `SSID` The SSID for a particular network
- `networkId` A unique identifier used to identify this network configuration on the current device
- `priority` The priority of each network configuration when choosing which of several access points to connect to
- `status` The current status of this network connection, will be one of `WifiConfiguration.Status.ENABLED`, `WifiConfiguration.Status.DISABLED`, or `WifiConfiguration.Status.CURRENT`

The configuration object also contains the supported authentication technique, as well as the keys used previously to authenticate with this access point.

The `addNetwork` method lets you specify a new configuration to add to the current list; similarly, `updateNetwork` lets you update a network configuration by passing in a `WifiConfiguration` that's sparsely populated with a network ID and the values you want to change.

You can also use `removeNetwork`, passing in a network ID, to remove a configuration.

To persist any changes made to the network configurations, you must call `saveConfiguration`.

Controlling Device Vibration

In Chapter 8, you learned how to create Notifications that can trigger vibration to provide additional user feedback when signaling events. In some circumstances, you may wish to vibrate the device independently of Notifications. Vibrating the device is an excellent way to provide haptic user feedback and is particularly popular as a feedback mechanism for games.

To control device vibration, your applications need the `VIBRATE` permission. Add this to your application manifest using the code snippet below:

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

Device vibration is controlled through the `Vibrator` class, accessible using the `getSystemService` method, as shown in the following code snippet:

```
String vibratorService = Context.VIBRATOR_SERVICE;
Vibrator vibrator = (Vibrator)getSystemService(vibratorService);
```

Call `vibrate` to start device vibration; you can pass in either a vibration duration or pattern of alternating vibration/pause sequences along with an optional index parameter that will repeat the pattern starting at the index specified. Both techniques are demonstrated below:

```
long[] pattern = {1000, 2000, 4000, 8000, 16000 };
vibrator.vibrate(pattern, 0);
vibrator.vibrate(1000); // Vibrate for 1 second
```

To cancel vibration, you can call `cancel`. Alternatively, exiting your application will automatically cancel any vibration it has initiated.

Summary

In this chapter, you learned how to monitor and control some of the hardware services available on Android devices.

Beginning with the media APIs, you learned about Android's multimedia capabilities including playback and recording using the Media Player and Media Recorder classes.

You were introduced to the Camera APIs that can be used to change camera settings, initiate live camera previews, and take photos.

With the Sensor Manager, you used the accelerometer and compass hardware to determine the device's orientation and acceleration, as well as monitoring and interpreting changes to detect device movement.

Finally, you examined the underlying communications hardware APIs available in Android. This included an introduction to the telephony APIs and an overview of the Bluetooth, network, and Wi-Fi managers for monitoring and controlling device connectivity.

This chapter also included:

- Monitoring phone state information, including cell location, phone state, and service state.
- Controlling the Bluetooth device to discover, bond, and transmit information between local and remote Bluetooth devices.
- Managing Bluetooth headsets.
- Managing Wi-Fi configurations, searching for access points, and managing Wi-Fi connections.
- Controlling device vibration to provide haptic feedback.

In the final chapter, you'll be introduced to some of the advanced Android features. You'll learn more about security and how to use AIDL to facilitate interprocess communication. You'll learn about Android's User Interface and graphics capabilities by exploring animations and advanced Canvas drawing techniques. Finally, you'll be introduced to the `SurfaceView` and touch-screen input functionality.



11

Advanced Android Development

In this chapter, you'll be returning to some of the possibilities touched on in previous chapters and explore some of the topics that deserve more attention.

In the first six chapters, you learned the fundamentals of creating mobile applications for Android devices. In Chapters 7 through 10, you were introduced to some of the more powerful optional APIs, including location-based services, maps, instant messaging, and hardware monitoring and control.

You'll start this chapter by taking a closer look at security, in particular, how permissions work and how to use them to secure your own applications.

Next you'll examine the Android Interface Definition Language (AIDL) and learn how to create rich application interfaces that support full object-based interprocess communication (IPC) between Android applications running in different processes.

You'll then take a closer look at the rich toolkit available for creating User Interfaces for your Activities. Starting with animations, you'll learn how to apply tweened animations to Views and View Groups, and construct frame-by-frame cell-based animations.

Next is an in-depth examination of the possibilities available with Android's raster graphics engine. You'll be introduced to the drawing primitives available before learning some of the more advanced possibilities available with Paint. You'll learn how to use transparency and create gradient Shaders and bitmap brushes. You'll be introduced to mask and color filters, as well as Path Effects and the possibilities of using different transfer modes.

You'll then delve a little deeper into the design and execution of more complex User Interface Views, learning how to create three-dimensional and high frame-rate interactive controls using the Surface View, and how to use the touch screen, trackball, and device keys to create intuitive input possibilities for your UIs.

Paranoid Android

Much of Android's security is native to the underlying Linux kernel. Resources are sandboxed to their owner applications, making them inaccessible from other applications. Android provides broadcast Intents, Services, and Content Providers to let you relax these strict process boundaries, using the permission mechanism to maintain application-level security.

You've already used the permission system to request access to native system services — notably the location-based services and contacts Content Provider — for your applications.

The following sections provide a more detailed look at the security available. For a comprehensive view, the Android documentation provides an excellent resource that describes the security features in depth at `code.google.com/android/devel/security.html`.

Linux Kernel Security

Each Android package has a unique Linux userID assigned to it during installation. This has the effect of sandboxing the process and the resources it creates, so that it can't affect (or be affected by) other applications.

Because of this kernel-level security, you need to take additional steps to communicate between applications. Enter Content Providers, broadcast Intents, and AIDL interfaces. Each of these mechanisms opens a tunnel for information to flow between applications. Android permissions act as border guards at either end to control the traffic allowed through these tunnels.

Introducing Permissions

Permissions are an application-level security mechanism that lets you restrict access to application components. Permissions are used to prevent malicious applications from corrupting data, gaining access to sensitive information, or making excessive (or unauthorized) use of hardware resources or external communication channels.

As you've learned in earlier chapters, many of Android's native components have permission requirements. The native permission strings used by native Android Activities and Services can be found as static constants in the `android.Manifest.permission` class.

To use permission-protected components, you need to add `uses-permission` tags to application manifests, specifying the permission string that each application requires.

When an application package is installed, the permissions requested in its manifest are analyzed and granted (or denied) by checks with trusted authorities and user feedback.

Unlike many existing mobile platforms, all Android permission checks are done at installation. Once an application is installed, the user will not be prompted to reevaluate those permissions.

> *There's no guarantee that your application will be granted the permissions it requests, so it's good practice to write defensive code that ensures it fails gracefully in these circumstances.*

Declaring and Enforcing Permissions

Before you can assign a permission to an application component, you need to define it within your manifest using the `permission` tag as shown in the following code snippet:

```
<permission
  android:name="com.paad.DETONATE_DEVICE"
  android:protectionLevel="dangerous"
  android:label="Self Destruct"
  android:description="@string/detonate_description">
</permission>
```

Within the permission tag, you can specify the level of access that the permission will permit (`normal`, `dangerous`, `signature`, `signatureOrSystem`), a label, and an external resource containing the description that explains the risks of granting this permission.

To include permission requirements for your own application components, use the `permission` attribute in the application manifest. Permission constraints can be enforced throughout your application, most usefully at application interface boundaries, for example:

- **Activities** Add a permission to limit the ability of other applications to launch an Activity.
- **Broadcast Receivers** Control which applications can send broadcast Intents to your receiver.
- **Content Providers** Limit Read access and Write operations on Content Providers.
- **Services** Limit the ability of other applications to start, or bind to, a Service.

In each case, you can add a `permission` attribute to the application component in the manifest, specifying a required permission string to access each component, as shown below in a manifest excerpt that shows a permission requirement for an Activity:

```
<activity
  android:name=".MyActivity"
  android:label="@string/app_name"
  android:permission="com.paad.DETONATE_DEVICE">
</activity>
```

Content Providers let you set `readPermission` and `writePermission` attributes to offer a more granular control over Read/Write access.

Enforcing Permissions with Broadcasting Intents

As well as requiring permissions for Intents to be received by your Broadcast Receivers, you can also attach a permission string to each Intent you broadcast.

When calling `sendIntent`, you can supply a permission string required by Broadcast Receivers before they can receive the Intent. This process is shown below:

```
sendBroadcast(myIntent, REQUIRED_PERMISSION);
```

Using AIDL to Support IPC for Services

One of the more interesting possibilities of Services is the idea of running independent background processes to supply processing, data lookup, or other useful functionality to multiple independent applications.

In Chapter 8, you learned how to create Services for your applications. Here, you'll learn how to use the Android Interface Definition Language (AIDL) to support interprocess communication (IPC) between Services and application components. This will give your Services the ability to support multiple applications across process boundaries.

To pass objects between processes, you need to deconstruct them into OS-level primitives that the underlying operating system (OS) can then marshal across application boundaries.

AIDL is used to simplify the code that lets your processes exchange objects. It's similar to interfaces like COM or Corba in that it lets you create public methods within your Services that can accept and return object parameters and return values between processes.

Implementing an AIDL Interface

AIDL supports the following data types:

- Java language primitives (`int`, `boolean`, `float`, `char`, etc.)
- `String` and `CharSequence` values
- `List` (including generic) objects, where each element is a supported type. The receiving class will always receive the List object instantiated as an `ArrayList`.
- `Map` (*not* including generic) objects in which each key and element is a supported type. The receiving class will always receive the `Map` object instantiated as a `HashMap`.
- AIDL-generated interfaces (covered later). An import statement is always needed for these.
- Classes that implement the `Parcelable` interface (covered next). An import statement is always needed for these.

The following sections demonstrate how to make your application classes AIDL-compatible by implementing the `Parcelable` interface, before creating an AIDL interface definition and implementing that interface within your Service.

Passing Custom Class Objects

To pass non-native objects between processes, they must implement the `Parcelable` interface. This lets you decompose your objects into primitive types stored within a `Parcel` that can be marshaled across process boundaries.

Implement the `writeToParcel` method to decompose your class object, then implement the public static `Creator` field (which implements a new `Parcelable.Creator` class), which will create a new object based on an incoming Parcel.

The following code snippet shows a basic example of using the `Parcelable` interface for the `Quake` class you've been using in the ongoing Earthquake example:

```
package com.paad.earthquake;

import java.util.Date;
import android.location.Location;
import android.os.Parcel;
import android.os.Parcelable;

public class Quake implements Parcelable {
  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;
  }

  private Quake(Parcel in) {
    date.setTime(in.readLong());
    details = in.readString();
    magnitude = in.readDouble();
    Location location = new Location("gps");
    location.setLatitude(in.readDouble());
    location.setLongitude(in.readDouble());
    link= in.readString();
  }

  public void writeToParcel(Parcel out, int flags) {
    out.writeLong(date.getTime());
    out.writeString(details);
    out.writeDouble(magnitude);
```

```
    out.writeDouble(location.getLatitude());
    out.writeDouble(location.getLongitude());
    out.writeString(link);
  }

  public static final Parcelable.Creator<Quake> CREATOR =
    new Parcelable.Creator<Quake>() {
      public Quake createFromParcel(Parcel in) {
        return new Quake(in);
      }

      public Quake[] newArray(int size) {
       return new Quake[size];
      }
    };

  public int describeContents() {
    return 0;
  }

}
```

Now that you've got a Parcelable class, you need to create an AIDL definition to make it available when defining your Service's AIDL interface.

The following code snippet shows the contents of the Quake.aidl file you need to create for the `Quake` class defined above:

```
package com.paad.earthquake;

parcelable Quake;
```

Remember that when you're passing class objects between processes, the client process must understand the definition of the object being passed.

Creating the AIDL Definition

In this section, you will be defining a new AIDL interface definition for a Service you'd like to use across processes.

Start by creating a new .aidl file within your project. This will define the methods and fields to include in an Interface that your Service will implement.

The syntax for creating AIDL definitions is similar to that used for standard Java interface definitions.

Start by specifying a fully qualified package name, then `import` all the packages required. Unlike normal Java interfaces, AIDL definitions need to import packages for any class or interface that isn't a native Java type even if it's defined in the same project.

Define a new `interface`, adding the properties and methods you want to make available.

Methods can take zero or more parameters and return void or a supported type. If you define a method that takes one or more parameters, you need to use a directional tag to indicate if the parameter is a value or reference type using the `in`, `out`, and `inout` keywords.

> *Where possible, you should limit the direction of each parameter, as marshaling parameters is an expensive operation.*

The following sample shows a basic AIDL definition for the IEarthquakeService.aidl file:

```
package com.paad.earthquake;

import com.paad.earthquake.Quake;

interface IEarthquakeService {
  List<Quake> getEarthquakes();

  void refreshEarthquakes();
}
```

Implementing and Exposing the IPC Interface

If you're using the ADT plug-in, saving the AIDL file will automatically code-generate a Java `interface` file. This interface will include an inner `Stub` class that implements the interface as an abstract class.

Have your Service extend the `Stub` and implement the functionality required. Typically, this will be done using a private field variable within the Service whose functionality you'll be exposing.

The following code snippet shows an implementation of the `IEarthquakeService` AIDL definition created above:

```
IBinder myEarthquakeServiceStub = new IEarthquakeService.Stub() {
  public void refreshEarthquakes() throws RemoteException {
    EarthquakeService.this.refreshEarthquakes();
  }

  public List<Quake> getEarthquakes() throws RemoteException {
    ArrayList<Quake> result = new ArrayList<Quake>();

    ContentResolver cr = EarthquakeService.this.getContentResolver();
    Cursor c = cr.query(EarthquakeProvider.CONTENT_URI, null, null, null, null);
    if (c.moveToFirst())
      do {

        Double lat = c.getDouble(EarthquakeProvider.LATITUDE_COLUMN);
        Double lng = c.getDouble(EarthquakeProvider.LONGITUDE_COLUMN);
        Location location = new Location("dummy");
        location.setLatitude(lat);
        location.setLongitude(lng);

        String details = c.getString(EarthquakeProvider.DETAILS_COLUMN);
```

```
        String link =  c.getString(EarthquakeProvider.LINK_COLUMN);

        double magnitude = c.getDouble(EarthquakeProvider.MAGNITUDE_COLUMN);

        long datems =  c.getLong(EarthquakeProvider.DATE_COLUMN);
        Date date = new Date(datems);

        result.add(new Quake(date, details, location, magnitude, link));
      } while(c.moveToNext());

    return result;
  }
};
```

There are several considerations when implementing these methods:

- All exceptions will remain local to the implementing process; they will not be propagated to the calling application.
- All IPC calls are synchronous. If you know that the process is likely to be time-consuming, you should consider wrapping the synchronous call in an asynchronous wrapper or moving the processing on the receiver side onto a background thread.

With the functionality implemented, you need to expose this interface to client applications. Expose the IPC-enabled Service interface by overriding the `onBind` method within our service implementation to return an instance of the interface.

The code snippet below demonstrates the `onBind` implementation for the `EarthquakeService`:

```
@Override
public IBinder onBind(Intent intent) {
  return myEarthquakeServiceStub;
}
```

To use the IPC Service from within an Activity, you must bind it as shown in the following code snippet taken from the `Earthquake` Activity:

```
IEarthquakeService earthquakeService = null;

private void bindService() {
  bindService(new Intent(IEarthquakeService.class.getName()),
              serviceConnection, Context.BIND_AUTO_CREATE);
}

private ServiceConnection serviceConnection = new ServiceConnection() {
  public void onServiceConnected(ComponentName className,
                                 IBinder service) {
    earthquakeService = IEarthquakeService.Stub.asInterface(service);
  }

  public void onServiceDisconnected(ComponentName className) {
```

```
        earthquakeService = null;
    }
};
```

Using Internet Services

Software as a service, or cloud computing, is becoming increasingly popular as companies try to reduce the cost overheads associated with installation, upgrades, and maintenance of deployed software. The result is a range of rich Internet services with which you can build thin mobile applications that enrich online services with the personalization available from your mobile.

The idea of using a middle tier to reduce client-side load is not a novel one, and happily there are many Internet-based options to supply your applications with the level of service you need.

The sheer volume of Internet services available makes it impossible to list them all here (let alone look at them in any detail), but the following list shows some of the more mature and interesting Internet services currently available:

- **Google's gData Services** As well as the native Google applications, Google offers Web APIs for access to their calendar, spreadsheet, Blogger, and Picasaweb platforms. These APIs collectively make use of Google's standardized gData framework, a form of Read/Write XML data communication.
- **Yahoo! Pipes** Yahoo! Pipes offers a graphical web-based approach to XML feed manipulation. Using pipes, you can filter, aggregate, analyze, and otherwise manipulate XML feeds and output them in a variety of formats to be consumed by your applications.
- **The Google App Engine** Using the Google App Engine, you can create cloud-hosted web services that shift complex processing away from your mobile client. Doing so reduces the load on your system resources but comes at the price of Internet-connection dependency.
- **Amazon Web Services** Amazon offers a range of cloud-based services, including a rich API for accessing its media database of books, CDs, and DVDs. Amazon also offers a distributed storage solution (S3) and an elastic compute cloud (EC2).

Building Rich User Interfaces

Mobile phone User Interfaces have improved dramatically in recent years, thanks not least of all to the iPhone's innovative take on mobile UI.

In this section, you'll learn how to use more advanced UI visual effects like Shaders, translucency, animations, touch screens, and OpenGL to add a level of polish to your Activities and Views.

Working with Animations

Back in Chapter 3, you learned how to define animations as external resources. Now, eight chapters later, you get the opportunity to put them to use.

Android offers two kinds of animation:

- **Frame-by-Frame Animations** Traditional cell-based animations in which a different Drawable is displayed in each frame. Frame-by-frame animations are displayed within a View, using its Canvas as a projection screen.
- **Tweened Animations** Tweened animations are applied to Views, letting you define a series of changes in position, size, rotation, and opacity that animate the View contents.

Both animation types are restricted to the original bounds of the View they're applied to. Rotations, translations, and scaling transformations that extend beyond the original boundaries of the View will result in the contents being clipped.

Introducing Tweened Animations

Tweened Animations offer a simple way to provide depth, movement, or feedback to your users at a minimal resource cost.

Using animations to apply a set of orientation, scale, position, and opacity changes is much less resource-intensive than manually redrawing the Canvas to achieve similar effects, not to mention far simpler to implement.

Tweened animations are commonly used to:

- Transition between Activities.
- Transition between layouts within an Activity.
- Transition between different content displayed within the same View.
- Provide user feedback such as:
  - A rotating hourglass View to indicate progress

    or
  - "Shaking" an input box to indicate an incorrect or invalid data entry.

Creating Tweened Animations

Tweened animations are created using the `Animation` class. The following list explains the animation types available:

- `AlphaAnimation` Lets you animate a change in the Views transparency (opacity or alpha blending).
- `RotateAnimation` Lets you spin the selected View canvas in the *XY* plane.
- `ScaleAnimation` Allows you to zoom in to or out from the selected View.
- `TranslateAnimation` Lets you move the selected View around the screen (although it will only be drawn within its original bounds).

Android offers the `AnimationSet` class to group and configure animations to be run as a set. You can define the start time and duration of each animation used within a set to control the timing and order of the animation sequence.

It's important to set the start offset and duration for each child animation, or they will all start and complete at the same time.

The following code and XML snippets demonstrate how to create the same animation sequence in code or as an external resource:

```
// Create the AnimationSet
AnimationSet myAnimation = new AnimationSet(true);

// Create a rotate animation.
RotateAnimation rotate = new RotateAnimation(0, 360,
                                             RotateAnimation.RELATIVE_TO_SELF,0.5f,
                                             RotateAnimation.RELATIVE_TO_SELF,0.5f
                                            );
rotate.setFillAfter(true);
rotate.setDuration(1000);

// Create a scale animation
ScaleAnimation scale = new ScaleAnimation(1, 0,
                                          1, 0,
                                          ScaleAnimation.RELATIVE_TO_SELF, 0.5f,
                                          ScaleAnimation.RELATIVE_TO_SELF, 0.5f
                                         );
scale.setFillAfter(true);
scale.setDuration(500);
scale.setStartOffset(500);

// Create an alpha animation
AlphaAnimation alpha = new AlphaAnimation(1, 0);
scale.setFillAfter(true);
scale.setDuration(500);
scale.setStartOffset(500);

// Add each animation to the set
myAnimation.addAnimation(rotate);
myAnimation.addAnimation(scale);
myAnimation.addAnimation(alpha);
```

The code snippet above implements the same animation sequence shown in the following XML snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<set
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:shareInterpolator="true">
  <rotate
    android:fromDegrees="0"
    android:toDegrees="360"
    android:pivotX="50%"
    android:pivotY="50%"
    android:startOffset="0"
    android:duration="1000" />
  <scale
    android:fromXScale="1.0"
    android:toXScale="0.0"
```

```
        android:fromYScale="1.0"
        android:toYScale="0.0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:startOffset="500"
        android:duration="500" />
      <alpha
        android:fromAlpha="1.0"
        android:toAlpha="0.0"
        android:startOffset="500"
        android:duration="500" />
    </set>
```

As you can see, it's generally both easier and more intuitive to create your animation sequences using an external animation resource.

Applying Tweened Animations

Animations can be applied to any View by calling its `startAnimation` method and passing in the Animation or Animation Set to apply.

Animation sequences will run once and then stop, unless you modify this behavior using the `setRepeatMode` and `setRepeatCount` methods on the Animation or Animation Set. You can force an animation to loop or ping-pong by setting the repeat mode of `RESTART` or `REVERSE`. Setting the repeat count controls the number of times the animation will repeat.

The following code snippet shows an Animation that repeats indefinitely:

```
myAnimation.setRepeatMode(Animation.RESTART);
myAnimation.setRepeatCount(Animation.INFINITE);

myView.startAnimation(myAnimation);
```

Using Animation Listeners

The `AnimationListener` lets you create an event handler that's fired when an animation begins or ends. This lets you perform actions before or after an animation has completed, such as changing the View contents or chaining multiple animations.

Call `setAnimationListener` on an Animation object, and pass in a new implementation of `AnimationListener`, overriding `onAnimationEnd`, `onAnimationStart`, and `onAnimationRepeat` as required.

The following skeleton code shows the basic implementation of an Animation Listener:

```
myAnimation.setAnimationListener(new AnimationListener() {

  public void onAnimationEnd(Animation _animation) {
   // TODO Do something after animation is complete.
  }


  public void onAnimationRepeat(Animation _animation) {
```

```
    // TODO Do something when the animation repeats.
  }

  public void onAnimationStart(Animation _animation) {
    // TODO Do something when the animation starts.
  }

});
```

Animated Sliding User Interface Example

In this example, you'll create a new Activity that uses an Animation to smoothly change the content of the User Interface based on the direction pressed on the D-pad.

1. Start by creating a new ContentSlider project featuring a `ContentSlider` Activity.

```
package com.paad.contentslider;

import android.app.Activity;
import android.view.KeyEvent;
import android.os.Bundle;
import android.view.animation.Animation;
import android.view.animation.Animation.AnimationListener;
import android.view.animation.AnimationUtils;
import android.widget.TextView;

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

2. Next, modify the main.xml layout resource. It should contain a single `TextView` with the text bold, centered, and relatively large.

```
<?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="fill_parent"
    android:gravity="center"
    android:textStyle="bold"
    android:textSize="30sp"
    android:text="CENTER"
    android:editable="false"
    android:singleLine="true"
    android:layout_margin="10px"
  />
</LinearLayout>
```

3. Then create a series of animations that slides the current View out of, and the next View into, the frame for each direction: left, right, up, and down. Each animation should have its own file.

3.1. Create `slide_bottom_in.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromYDelta="-100%p"
    android:toYDelta="0"
    android:duration="700"
  />
</set>
```

3.2. Create `slide_bottom_out.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromYDelta="0"
    android:toYDelta="100%p"
    android:duration="700"
  />
</set>
```

3.3. Create `slide_top_in.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromYDelta="100%p"
    android:toYDelta="0"
    android:duration="700"
  />
</set>
```

3.4. Create `slide_top_out.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromYDelta="0"
    android:toYDelta="-100%p"
    android:duration="700"
  />
</set>
```

3.5. Create `slide_left_in.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromXDelta="100%p"
    android:toXDelta="0"
    android:duration="700"
  />
</set>
```

3.6. Create `slide_left_out.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromXDelta="0"
    android:toXDelta="-100%p"
    android:duration="700"
  />
</set>
```

3.7. Create `slide_right_in.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromXDelta="-100%p"
    android:toXDelta="0"
    android:duration="700"
  />
</set>
```

3.8. Create `slide_right_out.xml`.

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:fromXDelta="0"
    android:toXDelta="100%p"
    android:duration="700"
  />
</set>
```

4. Return to the `ContentSlider` Activity and get references to the TextView and each of the animations you created in Step 3.

```
Animation slideInLeft;
Animation slideOutLeft;
Animation slideInRight;
Animation slideOutRight;
Animation slideInTop;
Animation slideOutTop;
Animation slideInBottom;
Animation slideOutBottom;
TextView myTextView;

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

  slideInLeft = AnimationUtils.loadAnimation(this, R.anim.slide_left_in);
  slideOutLeft = AnimationUtils.loadAnimation(this, R.anim.slide_left_out);
  slideInRight = AnimationUtils.loadAnimation(this, R.anim.slide_right_in);
  slideOutRight = AnimationUtils.loadAnimation(this, R.anim.slide_right_out);
  slideInTop = AnimationUtils.loadAnimation(this, R.anim.slide_top_in);
  slideOutTop = AnimationUtils.loadAnimation(this, R.anim.slide_top_out);
```

```
    slideInBottom = AnimationUtils.loadAnimation(this, R.anim.slide_bottom_in);
    slideOutBottom = AnimationUtils.loadAnimation(this,
                                                  R.anim.slide_bottom_out);

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

Each screen transition consists of two animations chained together: sliding out the old text before sliding in the new text. Rather than create multiple Views, you can change the value of the View once it's "off screen" before sliding it back in from the opposite side.

5. Create a new method that applies a slide-out animation and waits for it to complete before modifying the text and initiating the slide-in animation.

```
private void applyAnimation(Animation _out, Animation _in, String _newText) {
  final String text = _newText;
  final Animation in = _in;

  // Ensure the text stays out of screen when the slide-out
  // animation has completed.
  _out.setFillAfter(true);

  // Create a listener to wait for the slide-out animation to complete.
  _out.setAnimationListener(new AnimationListener() {

    public void onAnimationEnd(Animation _animation) {
      // Change the text
      myTextView.setText(text);
      // Slide it back in to view
      myTextView.startAnimation(in);
    }

    public void onAnimationRepeat(Animation _animation) {}

    public void onAnimationStart(Animation _animation) {}
  });

  // Apply the slide-out animation
  myTextView.startAnimation(_out);
}
```

6. The text displayed can represent nine positions. To keep track of the current location, create an `enum` for each position and an instance variable to track it.

```
TextPosition textPosition = TextPosition.Center;
enum TextPosition { UpperLeft, Top, UpperRight,
                    Left, Center, Right,
                    LowerLeft, Bottom, LowerRight };
```

7. Create a new method `movePosition` that takes the current position, and the direction to move, and calculates the new position. It should then execute the appropriate animation sequence created in Step 5.

```
private void movePosition(TextPosition _current,
                          TextPosition _directionPressed) {
```

```
    Animation in;
    Animation out;
    TextPosition newPosition;

    if (_directionPressed == TextPosition.Left){
      in = slideInLeft;
      out = slideOutLeft;
    }
    else if (_directionPressed == TextPosition.Right){
      in = slideInRight;
      out = slideOutRight;
    }
    else if (_directionPressed == TextPosition.Top){
      in = slideInTop;
      out = slideOutTop;
    }
    else {
      in = slideInBottom;
      out = slideOutBottom;
    }

    int newPosValue = _current.ordinal();
    int currentValue = _current.ordinal();

    // To simulate the effect of 'tilting' the device moving in one
    // direction should make text for the opposite direction appear.
    // Ie. Tilting right should make left appear.
    if (_directionPressed == TextPosition.Bottom)
      newPosValue = currentValue - 3;
    else if (_directionPressed == TextPosition.Top)
      newPosValue = currentValue + 3;
    else if (_directionPressed == TextPosition.Right) {
      if (currentValue % 3 != 0)
        newPosValue = currentValue - 1;
    }
    else if (_directionPressed == TextPosition.Left) {
      if ((currentValue+1) % 3 != 0)
        newPosValue = currentValue + 1;
    }

    if (newPosValue != currentValue &&
        newPosValue > -1 &&
        newPosValue < 9){
        newPosition = TextPosition.values()[newPosValue];

      applyAnimation(in, out, newPosition.toString());
      textPosition = newPosition;
    }
  }
```

8. Wire up the D-pad by overriding the Activity's `onKeyDown` handler to listen for key presses and trigger `movePosition` accordingly.

```
@Override
public boolean onKeyDown(int _keyCode, KeyEvent _event) {
```

```
    if (super.onKeyDown(_keyCode, _event))
      return true;

    if (_event.getAction() == KeyEvent.ACTION_DOWN){
      switch (_keyCode) {
        case (KeyEvent.KEYCODE_DPAD_LEFT):
          movePosition(textPosition, TextPosition.Left); return true;
        case (KeyEvent.KEYCODE_DPAD_RIGHT):
          movePosition(textPosition, TextPosition.Right); return true;
        case (KeyEvent.KEYCODE_DPAD_UP):
          movePosition(textPosition, TextPosition.Top); return true;
        case (KeyEvent.KEYCODE_DPAD_DOWN):
          movePosition(textPosition, TextPosition.Bottom); return true;
      }
    }
    return false;
  }
```

Running the application now will show a screen displaying "Center"; pressing any of the four directions will slide out this text and display the appropriate new position.

As an extra step, you could wire up the accelerometer sensor rather than relying on pressing the D-pad.

Animating Layouts and View Groups

A `LayoutAnimation` is used to animate View Groups, applying a single Animation (or Animation Set) to each child View in a predetermined sequence.

Use a `LayoutAnimationController` to specify an Animation (or Animation Set) that's applied to each child View in a View Group. Each View it contains will have the same animation applied, but you can use the Layout Animation Controller to specify the order and start time for each View.

Android includes two `LayoutAnimationController` classes.

- `LayoutAnimationController` Lets you select the start offset of each View (in milliseconds) and the order (`forward`, `reverse`, and `random`) to apply the animation to each child View.
- `GridLayoutAnimationController` Is a derived class that lets you assign the animation sequence of the child Views using grid row and column references.

Creating Layout Animations

To create a new Layout Animation, start by defining the Animation to apply to each child view. Then create a new `LayoutAnimation`, either in code or as an external animation resource, that references the animation to apply and defines the order and timing in which to apply it.

The following XML snippets show the definition of a simple animation stored as popin.xml in the res/anim folder, and a layout animation stored as popinlayout.xml.

The Layout Animation applies a simple "pop-in" animation randomly to each child View of any View Group it's assigned to.

res/anim/popin.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:interpolator="@android:anim/accelerate_interpolator">
  <scale
    android:fromXScale="0.0" android:toXScale="1.0"
    android:fromYScale="0.0" android:toYScale="1.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:duration="400"
  />
</set>
```

res/anim/popinlayout.xml

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
                 android:delay="0.5"
                 android:animationOrder="random"
                 android:animation="@anim/popin" />
```

Using Layout Animations

Once you've defined a Layout Animation, you can apply it to a ViewGroup either in code or in the layout XML resource. In XML this is done using the `android:layoutAnimation` tag in the layout definition, as shown in the following XML snippet:

```
android:layoutAnimation="@anim/popinlayout"
```

To set a Layout Animation in code, call `setLayoutAnimation` on the View Group, passing in a reference to the `LayoutAnimation` object you want to apply.

In both cases, the Layout Animation will execute once, when the View Group is first laid out. You can force it to execute again by calling `scheduleLayoutAnimation` on the `ViewGroup` object. The animation will be executed the next time the View Group is laid out.

Layout Animations also support Animation Listeners.

In the following code snippet, a ViewGroup's animation is re-run with a listener attached to trigger additional actions once it's complete:

```
aViewGroup.setLayoutAnimationListener(new AnimationListener() {

  public void onAnimationEnd(Animation _animation) {
    // TODO: Actions on animation complete.
  }

  public void onAnimationRepeat(Animation _animation) {}

  public void onAnimationStart(Animation _animation) {}
});

aViewGroup.scheduleLayoutAnimation();
```

Creating and Using Frame-by-Frame Animations

Frame-by-frame animations are akin to traditional cell-based cartoons where an image is chosen for each frame. Where tweened animations use the target View to supply the content of the animation, frame-by-frame animations let you specify a series of `Drawable` objects that are used as the background to a View.

The `AnimationDrawable` class is used to create a new frame-by-frame animation presented as a `Drawable` resource.

You can define your Animation Drawable resource as an external resource in your project's `drawable` folder using XML. Use the `animation-list` tag to group a collection of `item` tags, each of which uses a `drawable` attribute to define an image to display, and a `duration` attribute to specify the time (in milliseconds) to display it.

The following XML snippet shows how to create a simple animation that displays a rocket taking off (rocket images not included). The file is stored as res/drawable/animated_rocket.xml:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
                android:oneshot="false">
  <item android:drawable="@drawable/rocket1" android:duration="500" />
  <item android:drawable="@drawable/rocket2" android:duration="500" />
  <item android:drawable="@drawable/rocket3" android:duration="500" />
 </animation-list>
```

To display your animation, set it as the background to a View using the `setBackgroundResource` method, as shown in the following code snippet:

```
ImageView image = (ImageView)findViewById(R.id.my_animation_frame);
image.setBackgroundResource(R.drawable.animated_rocket);
```

Alternatively, use the `setBackgroundDrawable` to use a Drawable instance instead of a resource reference. Run the animation calling its `start` method, as shown in the code snippet below:

```
AnimationDrawable animation = (AnimationDrawable)image.getBackground();
animation.start();
```

Using Themes to Skin Your Applications

The multifunction nature of a mobile device means users will be running and switching between many applications created by a range of different developers. Themes are a way of ensuring that your applications present a consistent look and feel.

To apply a theme, set the `android:theme` attribute on either the `application` or an individual `activity` tag in the manifest, as shown in the code snippet below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.google.android.home">
  <application
     android:theme="@android:style/Theme.Light" >
     <activity
        android:theme="@android:style/Theme.Black" >
```

```
        </activity>
    </application>
</manifest>
```

Android includes several predefined themes as part of the base package, including:

- `Theme.Black` Features a black background with white foreground controls and text.
- `Theme.Light` Features a white background with dark borders and text.
- `Theme.Translucent` Features a partially transparent Form.

You can set the theme of an Activity at run time, but it's generally not recommended, as Android uses your Activity's theme for intra-Activity animations, which happens before your application is loaded. If you do apply a theme programmatically, be sure to do so before you lay out the Activity as shown in the code snippet below:

```
protected void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setTheme(android.R.style.Theme_Translucent);
  setContentView(R.layout.main);
}
```

If you don't apply a theme to your application or any of its Activities, it will use the default theme `android.R.style.Theme`.

Advanced Canvas Drawing

You were introduced to the `Canvas` class in Chapter 4, where you learned how to create your own Views. The Canvas was also used in Chapter 7 to annotate Overlays for `MapViews`.

The concept of the Canvas is a common metaphor used in graphics programming and generally consists of three basic drawing components:

- `Canvas` Supplies the draw methods that paint drawing primitives onto the underlying bitmap.
- `Paint` Also referred to as a "brush," `Paint` lets you specify how a primitive is drawn on the bitmap.
- `Bitmap` Is the surface being drawn on.

Most of the advanced techniques described in this chapter involve variations and modifications to the `Paint` object that let you add depth and texture to otherwise flat raster drawings.

The Android drawing API supports translucency, gradient fills, rounded rectangles, and anti-aliasing. Unfortunately, owing to resource limitations, it does not yet support vector graphics; instead, it uses traditional raster-style repaints.

The result of this raster approach is improved efficiency, but changing a `Paint` object will not affect primitives that have already been drawn; it will only affect new elements.

If you've got a Windows development background, the two-dimensional (2D) drawing capabilities of Android are roughly equivalent to those available in GDI+.

What Can You Draw?

The `Canvas` class wraps up the bitmap that's used as a surface for your artistic endeavors; it also exposes the `draw*` methods used to implement your designs.

Without going into detail on each of the `draw` methods, the following list provides a taste of the primitives available:

- `drawARGB / drawRGB / drawColor` Fill the canvas with a single color.
- `drawArc` Draws an arc between two angles within an area bounded by a rectangle.
- `drawBitmap` Draws a bitmap on the Canvas. You can alter the appearance of the target bitmap by specifying a target size or using a matrix to transform it.
- `drawBitmapMesh` Draws a bitmap using a mesh that lets you manipulate the appearance of the target by moving points within it.
- `drawCircle` Draws a circle of a specified radius centered on a given point.
- `drawLine(s)` Draws a line (or series of lines) between two points.
- `drawOval` Draws an oval bounded by the rectangle specified.
- `drawPaint` Fills the entire Canvas with the specified Paint.
- `drawPath` Draws the specified Path. A `Path` object is often used to hold a collection of drawing primitives within a single object.
- `drawPicture` Draws a `Picture` object within the specified rectangle.
- `drawPosText` Draws a text string specifying the offset of each character.
- `drawRect` Draws a rectangle.
- `drawRoundRect` Draws a rectangle with rounded edges.
- `drawText` Draws a text string on the Canvas. The text font, size, color, and rendering properties are all set in the `Paint` object used to render the text.
- `drawTextOnPath` Draws text that follows along a specified path.
- `drawVertices` Draws a series of tri-patches specified as a series of vertex points.

Each of these drawing methods lets you specify a `Paint` object to render it. In the following sections, you'll learn how to create and modify `Paint` objects to get the most out of your drawing.

Getting the Most from Your Paint

The `Paint` class represents a paint brush and palette. It lets you choose how to render the primitives you draw onto the canvas using the draw methods described above. By modifying the `Paint` object, you can control the color, style, font, and special effects used when drawing. Most simply, `setColor` lets you select the color of a Paint while the style of a `Paint` object (controlled using `setStyle`) lets you decide if you want to draw only the outline of a drawing object (`STROKE`), just the filled portion (`FILL`), or both (`STROKE_AND_FILL`).

Beyond these simple controls, the `Paint` class also supports transparency and can also be modified using a variety of Shaders, filters, and effects to provide a rich palette of complex paints and brushes.

The Android SDK includes several excellent projects that demonstrate most of the features available in the `Paint` class. They are available in the graphics subfolder of the API demos at

```
[sdk root folder]\samples\ApiDemos\src\com\android\samples\graphics
```

In the following sections, you'll learn what some of these features are and how to use them. These sections outline what can be achieved (such as gradients and edge embossing) without exhaustively listing all possible alternatives.

Using Translucency

All colors in Android include an opacity component (alpha channel).

You define an alpha value for a color when you create it using the `argb` or `parseColor` methods shown below:

```
// Make color red and 50% transparent
int opacity = 127;
int intColor = Color.argb(opacity, 255, 0, 0);
int parsedColor = Color.parseColor("#7FFF0000");
```

Alternatively, you can set the opacity of an existing `Paint` object using the `setAlpha` method:

```
// Make color 50% transparent
int opacity = 127;
myPaint.setAlpha(opacity);
```

Creating a paint color that's not 100 percent opaque means that any primitive drawn with it will be partially transparent — making whatever is drawn beneath it partially visible.

You can use transparency effects in any class or method that uses colors including Paint colors, Shaders, and Mask Filters.

Introducing Shaders

Extensions of the `Shader` class let you create Paints that fill drawn objects with more than a single solid color.

The most common use of Shaders is to define gradient fills; gradients are an excellent way to add depth and texture to 2D drawings. Android includes three gradient Shaders as well as a Bitmap Shader and a Compose Shader.

Trying to describe painting techniques seems inherently futile, so have a look at Figure 11-1 to get an idea of how each of the Shaders works. Represented from left to right are `LinearGradient`, `RadialGradient`, and `SweepGradient`.

Not included in the image in Figure 11-1 is the `ComposeShader`*, which lets you create a composite of multiple Shaders and the* `BitmapShader` *that lets you create a paint brush based on a bitmap image.*

To use a Shader when drawing, apply it to a Paint using the `setShader` method, as shown in the following code snippet:

```
Paint shaderPaint = new Paint();
shaderPaint.setShader(myLinearGradient);
```

Anything you draw with this Paint will be filled with the Shader you specified rather than the paint color.

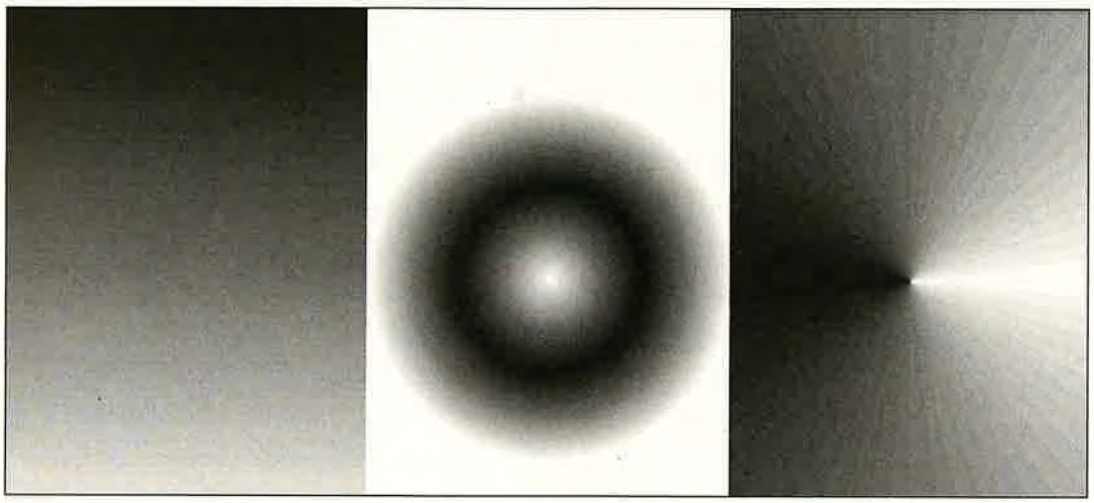


Figure 11-1

Defining Gradient Shaders

As shown above, using gradient Shaders lets you fill drawings with an interpolated color range; you can define the gradient as either a simple transition between two colors, as shown in the `LinearGradientShader` in the following code snippet:

```
int colorFrom = Color.BLACK;
int colorTo = Color.WHITE;

LinearGradient linearGradientShader = new LinearGradient(x1, y1, x2, y2,
                                                         colorFrom,
                                                         colorTo,
                                                         TileMode.CLAMP);
```

or as a more complex series of colors distributed at set proportions, as shown in the following example of a `RadialGradientShader`:

```
int[] gradientColors = new int[3];
gradientColors[0] = Color.GREEN;
```

```
gradientColors[1] = Color.YELLOW;
gradientColors[2] = Color.RED;

float[] gradientPositions = new float[3];
gradientPositions[0] = 0.0f;
gradientPositions[1] = 0.5f;
gradientPositions[2] = 1.0f;

RadialGradient radialGradientShader = new RadialGradient(centerX, centerY, radius,
                                                         gradientColors,
                                                         gradientPositions,
                                                         TileMode.CLAMP);
```

Each of the gradient Shaders (linear, radial, and sweep) lets you define the gradient fill using either of these techniques.

Using Shader Tile Modes

The brush sizes of the gradient Shaders are defined using explicit bounding rectangles or center points and radius lengths; the Bitmap Shader implies a brush size through its bitmap size.

If the area defined by your Shader brush is smaller than the area being filled, the `TileMode` determines how the remaining area will be covered.

- `CLAMP` Uses the edge colors of the Shader to fill the extra space.
- `MIRROR` Flips the Shader image horizontally and vertically so that each image seams with the last.
- `REPEAT` Repeats the Shader image horizontally and vertically, but doesn't flip it.

Using `MaskFilters`

The `MaskFilter` classes let you assign edge effects to your Paint.

Extensions to `MaskFilter` apply transformations to the alpha-channel of a Paint along its outer edge. Android includes the following Mask Filters:

- `BlurMaskFilter` Specifies a blur style and radius to feather the edges of your Paint.
- `EmbossMaskFilter` Specifies the direction of the light source and ambient light level to add an embossing effect.

To apply a Mask Filter, use the `setMaskFilter` method, passing in a `MaskFilter` object. The following code snippet applies an `EmbossMaskFilter` to an existing Paint:

```
// Set the direction of the light source
float[] direction = new float[]{ 1, 1, 1 };
// Set the ambient light level
float light = 0.4f;
// Choose a level of specularity to apply
float specular = 6;
```

```
// Apply a level of blur to apply to the mask
float blur = 3.5f;
EmbossMaskFilter emboss = new EmbossMaskFilter(direction, light, specular, blur);

// Apply the mask
myPaint.setMaskFilter(emboss);
```

The `FingerPaint` API demo included in the SDK is an excellent example of how to use `MaskFilters`. It demonstrates the effect of both the emboss and blur filters.

Using `ColorFilters`

Where `MaskFilters` are transformations of the alpha-channel of a Paint, a `ColorFilter` applies a transformation to each of the RGB channels. All `ColorFilter`-derived classes ignore the alpha-channel when performing their transformations.

Android includes three Color Filters:

- `ColorMatrixColorFilter` Lets you specify a 4 × 5 `ColorMatrix` to apply to a Paint. `ColorMatrixes` are commonly used to perform image processing programmatically and are useful as they support chaining transformations using matrix multiplication.
- `LightingColorFilter` Multiplies the RGB channels by the first color before adding the second. The result of each transformation will be clamped to between 0 and 255.
- `PorterDuffColorFilter` Lets you use any one of the 16 Porter-Duff rules for digital image compositing to apply a specified color to the Paint.

Apply `ColorFilters` using the `setColorFilter` method as shown below:

```
myPaint.setColorFilter(new LightingColorFilter(Color.BLUE, Color.RED));
```

There is an excellent example of using a Color Filter and Color Matrixes in the `ColorMatrixSample` API example.

Using `PathEffects`

The effects so far have affected the way the Paint *fills* a drawing; `PathEffects` are used to control how its outline (or stroke) is drawn.

Path Effects are particularly useful for drawing Path primitives, but they can be applied to any Paint to affect the way the stroke is drawn.

Using Path Effects, you can change the appearance of a shape's corners and control the appearance of the outline. Android includes several Path Effects including:

- `CornerPathEffect` Lets you smooth sharp corners in the shape of a primitive by replacing sharp edges with rounded corners.
- `DashPathEffect` Rather than drawing a solid outline, you can use the `DashPathEffect` to create an outline of broken lines (dashes/dots). You can specify any repeating pattern of solid/empty line segments.

- `DiscretePathEffect` Similar to the `DashPathEffect`, but with added randomness. Specifies the length of each segment and a degree of deviation from the original path to use when drawing it.
- `PathDashPathEffect` This effect lets you define a new shape (path) to use as a stamp to outline the original path.

The following effects let you combine multiple Path Effects to a single Paint.

- `SumPathEffect` Adds two effects to a path in sequence, such that each effect is applied to the original path and the two results are combined.
- `ComposePathEffect` Compose applies first one effect and then applies the second effect to the result of the first.

Path Effects that modify the shape of the object being drawn will change the area of the affected shape. This ensures that any fill effects being applied to the same shape are drawn within the new bounds.

Path Effects are applied to `Paint` objects using the `setPathEffect` method as shown below:

```
borderPaint.setPathEffect(new CornerPathEffect(5));
```

The Path Effects API sample gives an excellent guide to how to apply each of these effects.

Changing the `Xfermode`

Change a Paint's `Xfermode` to affect the way it paints new colors on top of what's already on the Canvas.

Under normal circumstances, painting on top of an existing drawing will layer the new shape on top. If the new Paint is fully opaque, it will totally obscure the paint underneath; if it's partially transparent, it will tint the colors underneath.

The following `Xfermode` subclasses let you change this behavior:

- `AvoidXfermode` Specifies a color and tolerance to force your Paint to avoid drawing over (or only draw over) it.
- `PixelXorXfermode` Applies a simple pixel XOR operation when covering existing colors.
- `PorterDuffXfermode` This is a very powerful transfer mode with which you can use any of the 16 Porter-Duff rules for image composition to control how the paint interacts with the existing canvas image.

To apply transfer modes, use the `setXferMode` method as shown in the sample below:

```
AvoidXfermode avoid = new AvoidXfermode(Color.BLUE, 10, AvoidXfermode.Mode.AVOID);
borderPen.setXfermode(avoid);
```

Improving Paint Quality with Anti-Aliasing

When you create a new `Paint` object, you can pass in several flags that affect the way the Paint will be rendered. One of the most interesting is the `ANTI_ALIAS_FLAG`, which ensures that diagonal lines drawn with this paint are anti-aliased to give a smooth appearance (at the cost of performance).

Anti-aliasing is particularly important when drawing text, as anti-aliased text can be significantly easier to read. To create even smoother text effects, you can apply the `SUBPIXEL_TEXT_FLAG`, which will apply subpixel anti-aliasing.

You can also set both of these flags manually using the `setSubpixelText` and `setAntiAlias` methods, as shown below:

```
myPaint.setSubpixelText(true);
myPaint.setAntiAlias(true);
```

Hardware Acceleration for 2D Graphics

In a boon for 2D graphics enthusiasts everywhere, Android lets you request that your application always be rendered using hardware acceleration.

If hardware acceleration is available on the device, setting this flag will cause every View within the Activity to be rendered using hardware. This has the side effect of dramatically improving the speed of your graphics while reducing the load on the system processor.

Turn it on by applying the `Window.FEATURE_OPENGL` flag to your Activity using the `requestWindowFeature` method, as shown below:

```
myActivity.requestWindowFeature(Window.FEATURE_OPENGL);
```

Unfortunately, Good Things seldom come for free, and this is no exception.

Not all the 2D drawing primitives available in Android are supported by hardware (notably most of the Path Effects described previously).

Also, as your entire Activity is being effectively rendered as a single Canvas, invalidate requests on any View will cause the whole Activity to be redrawn.

Canvas Drawing Best Practice

2D owner-draw operations tend to be expensive in terms of processor use; inefficient drawing routines can block the GUI thread and have a detrimental effect on application responsiveness. This is particularly true in a resource-constrained environment with a single, limited processor.

You need to be aware of the resource drain and CPU-cycle cost of your `onDraw` methods, to ensure you don't end up with an attractive application that's completely unresponsive.

A lot of techniques exist to help minimize the resource drain associated with owner-drawn controls. Rather than focus on general principles, I'll describe some Android specific considerations for ensuring that you can create activities that look good and remain interactive (note that this list is not exhaustive):

- **Consider Hardware Acceleration** OpenGL hardware acceleration support for 2D graphics is a Good Thing, so you should always consider if it's suitable for your Activity. Good candidates are Activities consisting of a single View with rapid, time-consuming updates. Be sure that the primitives you use are supported by hardware.

- **Consider Size and Orientation** When you're designing your Views and Overlays, be sure to consider (and test!) how they will look at different resolutions and sizes.
- **Create Static Objects Once** Object creation in Android is particularly expensive. Where possible, create drawing objects like `Paint` objects, Paths, and Shaders once, rather than recreating them each time the View is invalidated.
- **Remember `onDraw` Is Expensive** Performing the `onDraw` method is an expensive process that forces Android to perform several image composition and bitmap construction operations. Many of the following points suggest ways to modify the appearance of your Canvas without having to redraw it:
  - **Use Canvas Transforms** Use canvas transforms like `rotate` and `translate` to simplify complex relational positioning of elements on your canvas. For example, rather than positioning and rotating each text element around a clock face, simply rotate the canvas 22.5 degrees, and draw the text in the same place.
  - **Use Animations** Consider using Animations to perform pre-set transformations of your View rather than manually redrawing it. Scale, rotation, and translation Animations can be performed on any View within an Activity and provide a resource-efficient way to provide zoom, rotate, or shake effects.
  - **Consider Using Bitmaps and 9 Patches** If your Views feature static backgrounds, you should consider using a Drawable like a bitmap or scalable 9 patch rather than manually drawing it.

Advanced Compass Face Example

Early in Chapter 4, you created a simple compass. In the last chapter, you returned to it, extending it to display the pitch and roll using the accelerometer hardware.

The UI of the View used in those examples was kept simple to keep the code in those chapters as clear as possible.

In the following example, you'll make some significant changes to the `CompassView`'s `onDraw` method to change it from a simple, flat compass into a dynamic artificial horizon, as shown in Figure 11-2.



Figure 11-2

As the previous image is limited to black and white, you'll need to create the control in order to see the full effect.

1. Start by modifying the `colors.xml` resource file to include color values for the border gradient, the glass compass shading, the sky, and the ground. Also update the colors used for the border and the face markings.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="text_color">#FFFF</color>
    <color name="background_color">#F000</color>
    <color name="marker_color">#FFFF</color>
    <color name="shadow_color">#7AAA</color>

    <color name="outer_border">#FF444444</color>
    <color name="inner_border_one">#FF323232</color>
    <color name="inner_border_two">#FF414141</color>
    <color name="inner_border">#FFFFFFFF</color>

    <color name="horizon_sky_from">#FFA52A2A</color>
    <color name="horizon_sky_to">#FFFFC125</color>
    <color name="horizon_ground_from">#FF5F9EA0</color>
    <color name="horizon_ground_to">#FF00008B</color>
</resources>
```

2. The `Paint` and `Shader` objects used for the sky and ground in the artificial horizon are created based on the size of the current View, so they're not static like the `Paint` objects you created in Chapter 4. Instead of creating `Paint` objects, construct the gradient arrays and colors they use.

```
int[] borderGradientColors;
float[] borderGradientPositions;

int[] glassGradientColors;
float[] glassGradientPositions;

int skyHorizonColorFrom;
int skyHorizonColorTo;
int groundHorizonColorFrom;
int groundHorizonColorTo;
```

3. Update the `CompassView`'s `initCompassView` method to initialize the variables created in Step 2 using the resources from Step 1. The existing method code can be left largely intact, with some changes to the `textPaint`, `circlePaint`, and `markerPaint` variables, as highlighted below:

```
protected void initCompassView() {
  setFocusable(true);
  // Get external resources
  Resources r = this.getResources();

  circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  circlePaint.setColor(R.color.background_color);
  circlePaint.setStrokeWidth(1);
  circlePaint.setStyle(Paint.Style.STROKE);

  northString = r.getString(R.string.cardinal_north);
```

```
eastString = r.getString(R.string.cardinal_east);
southString = r.getString(R.string.cardinal_south);
westString = r.getString(R.string.cardinal_west);

textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
textPaint.setColor(r.getColor(R.color.text_color));
textPaint.setFakeBoldText(true);
textPaint.setSubpixelText(true);
textPaint.setTextAlign(Align.LEFT);

textHeight = (int)textPaint.measureText("yY");

markerPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
markerPaint.setColor(r.getColor(R.color.marker_color));
markerPaint.setAlpha(200);
markerPaint.setStrokeWidth(1);
markerPaint.setStyle(Paint.Style.STROKE);
markerPaint.setShadowLayer(2, 1, 1, r.getColor(R.color.shadow_color));
```

3.1. Create the color and position arrays that will be used by a radial Shader to paint the outer border.

```
borderGradientColors = new int[4];
borderGradientPositions = new float[4];

borderGradientColors[3] = r.getColor(R.color.outer_border);
borderGradientColors[2] = r.getColor(R.color.inner_border_one);
borderGradientColors[1] = r.getColor(R.color.inner_border_two);
borderGradientColors[0] = r.getColor(R.color.inner_border);
borderGradientPositions[3] = 0.0f;
borderGradientPositions[2] = 1-0.03f;
borderGradientPositions[1] = 1-0.06f;
borderGradientPositions[0] = 1.0f;
```

3.2. Now create the radial gradient color and position arrays that will be used to create the semitransparent "glass dome" that sits on top of the View to give it the illusion of depth.

```
glassGradientColors = new int[5];
glassGradientPositions = new float[5];

int glassColor = 245;
glassGradientColors[4] = Color.argb(65, glassColor, glassColor, glassColor);
glassGradientColors[3] = Color.argb(100, glassColor, glassColor, glassColor);
glassGradientColors[2] = Color.argb(50, glassColor, glassColor, glassColor);
glassGradientColors[1] = Color.argb(0, glassColor, glassColor, glassColor);
glassGradientColors[0] = Color.argb(0, glassColor, glassColor, glassColor);
glassGradientPositions[4] = 1-0.0f;
glassGradientPositions[3] = 1-0.06f;
glassGradientPositions[2] = 1-0.10f;
glassGradientPositions[1] = 1-0.20f;
glassGradientPositions[0] = 1-1.0f;
```

3.3. Finally, get the colors you'll use to create the linear gradients that will represent the sky and the ground in the artificial horizon.

```
    skyHorizonColorFrom = r.getColor(R.color.horizon_sky_from);
    skyHorizonColorTo = r.getColor(R.color.horizon_sky_to);

    groundHorizonColorFrom = r.getColor(R.color.horizon_ground_from);
    groundHorizonColorTo = r.getColor(R.color.horizon_ground_to);
  }
```

4. Before you start drawing the face, create a new `enum` that stores each of the cardinal directions.

```
private enum CompassDirection { N, NNE, NE, ENE,
                                E, ESE, SE, SSE,
                                S, SSW, SW, WSW,
                                W, WNW, NW, NNW }
```

Now you need to completely replace the existing `onDraw` method. You'll start by figuring out some size-based values including the center of the View, the radius of the circular control, and the rectangles that will enclose the outer (heading) and inner (tilt and roll) face elements.

```
@Override
protected void onDraw(Canvas canvas) {
```

1. Calculate the width of the outer (heading) ring based on the size of the font used to draw the heading values.

```
    float ringWidth = textHeight + 4;
```

2. Then calculate the height and width of the View, and use those values to establish the radius of the inner and outer face dials, as well as create the bounding boxes for each face.

```
    int height = getMeasuredHeight();
    int width =getMeasuredWidth();

    int px = width/2;
    int py = height/2;
    Point center = new Point(px, py);

    int radius = Math.min(px, py)-2;

    RectF boundingBox = new RectF(center.x - radius,
                                  center.y - radius,
                                  center.x + radius,
                                  center.y + radius);

    RectF innerBoundingBox = new RectF(center.x - radius + ringWidth,
                                       center.y - radius + ringWidth,
                                       center.x + radius - ringWidth,
                                       center.y + radius - ringWidth);

    float innerRadius = innerBoundingBox.height()/2;
```

3. With the dimensions of the View established, it's time to start drawing the faces.

   Start from the bottom layer at the outside, and work your way in and up, starting with the outer face (heading). Create a new `RadialGradient` Shader using the colors and positions you defined in Step 3.2 in the previous code sample, and assign that Shader to a new Paint before using it to draw a circle.

```
RadialGradient borderGradient = new RadialGradient(px, py, radius,
                                                   borderGradientColors,
                                                   borderGradientPositions,
                                                   TileMode.CLAMP);

Paint pgb = new Paint();
pgb.setShader(borderGradient);

Path outerRingPath = new Path();
outerRingPath.addOval(boundingBox, Direction.CW);

canvas.drawPath(outerRingPath, pgb);
```

4. Next you need to draw the artificial horizon. The horizon is created by dividing the circular face into two sections, one representing the sky and the other the ground. The proportion of each section depends on the current pitch.

   Start by creating the `Shader` and `Paint` objects that will be used to draw the sky and earth.

```
LinearGradient skyShader = new LinearGradient(center.x,
                                              innerBoundingBox.top,
                                              center.x,
                                              innerBoundingBox.bottom,
                                              skyHorizonColorFrom,
                                              skyHorizonColorTo,
                                              TileMode.CLAMP);
Paint skyPaint = new Paint();
skyPaint.setShader(skyShader);

LinearGradient groundShader = new LinearGradient(center.x,
                                                 innerBoundingBox.top,
                                                 center.x,
                                                 innerBoundingBox.bottom,
                                                 groundHorizonColorFrom,
                                                 groundHorizonColorTo,
                                                 TileMode.CLAMP);
Paint groundPaint = new Paint();
groundPaint.setShader(groundShader);
```

5. Now normalize the pitch and roll values to clamp them within ±90 degrees and ±180 degrees, respectively.

```
float tiltDegree = pitch;
while (tiltDegree > 90 || tiltDegree < -90)
{
  if (tiltDegree > 90) tiltDegree = -90 + (tiltDegree - 90);
    if (tiltDegree < -90) tiltDegree = 90 - (tiltDegree + 90);
}

float rollDegree = roll;
```

```
while (rollDegree > 180 || rollDegree < -180)
{
  if (rollDegree > 180) rollDegree = -180 + (rollDegree - 180);
   if (rollDegree < -180) rollDegree = 180 - (rollDegree + 180);
}
```

6. Create paths that will fill each segment of the circle (ground and sky). The proportion of each segment should be related to the clamped pitch.

```
Path skyPath = new Path();
skyPath.addArc(innerBoundingBox,
               -tiltDegree,
               (180 + (2 * tiltDegree)));
```

7. Spin the canvas around the center in the opposite direction to the current roll, and draw the sky and ground paths using the Paints you created in Step 4.

```
canvas.rotate(-rollDegree, px, py);
canvas.drawOval(innerBoundingBox, groundPaint);
canvas.drawPath(skyPath, skyPaint);
canvas.drawPath(skyPath, markerPaint);
```

8. Next is the face marking. Start by calculating the start and end points for the horizontal horizon markings.

```
int markWidth = radius / 3;
int startX = center.x - markWidth;
int endX = center.x + markWidth;
```

9. To make the horizon values easier to read, you should ensure that the pitch scale always starts at the current value. The following code calculates the position of the interface between the ground and sky on the horizon face:

```
double h = innerRadius*Math.cos(Math.toRadians(90-tiltDegree));
double justTiltY = center.y - h;
```

10. Find the number of pixels that represents each degree of tilt.

```
float pxPerDegree = (innerBoundingBox.height()/2)/45f;
```

11. Now iterate over 180 degrees, centered on the current tilt value, to give a sliding scale of possible pitch.

```
for (int i = 90; i >= -90; i -= 10)
{
  double ypos = justTiltY + i*pxPerDegree;

  // Only display the scale within the inner face.
  if ((ypos < (innerBoundingBox.top + textHeight)) ||
      (ypos > innerBoundingBox.bottom - textHeight))
    continue;

  // Draw a line and the tilt angle for each scale increment.
  canvas.drawLine(startX, (float)ypos, endX, (float)ypos, markerPaint);
  int displayPos = (int)(tiltDegree - i);
```

```
        String displayString = String.valueOf(displayPos);
        float stringSizeWidth = textPaint.measureText(displayString);
        canvas.drawText(displayString,
                        (int)(center.x-stringSizeWidth/2),
                        (int)(ypos)+1,
                        textPaint);
    }
```

12. Now draw a thicker line at the earth/sky interface. Change the stroke thickness of the `markerPaint` object before drawing the line (then set it back to the previous value).

```
markerPaint.setStrokeWidth(2);
canvas.drawLine(center.x - radius / 2,
                (float)justTiltY,
                center.x + radius / 2,
                (float)justTiltY,
                markerPaint);
markerPaint.setStrokeWidth(1);
```

13. To make it easier to read the exact roll, you should draw an arrow and display a text string that shows the exact value.

Create a new `Path`, and use the `moveTo` / `lineTo` methods to construct an open arrow that points straight up. Draw the path and a text string that shows the current roll.

```
// Draw the arrow
Path rollArrow = new Path();
rollArrow.moveTo(center.x - 3, (int)innerBoundingBox.top + 14);
rollArrow.lineTo(center.x, (int)innerBoundingBox.top + 10);
rollArrow.moveTo(center.x + 3, innerBoundingBox.top + 14);
rollArrow.lineTo(center.x, innerBoundingBox.top + 10);
canvas.drawPath(rollArrow, markerPaint);
// Draw the string
String rollText = String.valueOf(rollDegree);
double rollTextWidth = textPaint.measureText(rollText);
canvas.drawText(rollText,
                (float)(center.x - rollTextWidth / 2),
                innerBoundingBox.top + textHeight + 2,
                textPaint);
```

14. Spin the canvas back to upright so that you can draw the rest of the face markings.

```
canvas.restore();
```

15. Draw the roll dial markings by rotating the canvas 10 degrees at a time to draw either a mark or a value. When you've completed the face, restore the canvas to its upright position.

```
canvas.save();
canvas.rotate(180, center.x, center.y);
for (int i = -180; i < 180; i += 10)
{
  // Show a numeric value every 30 degrees
  if (i % 30 == 0) {
    String rollString = String.valueOf(i*-1);
    float rollStringWidth = textPaint.measureText(rollString);
```

```
      PointF rollStringCenter = new PointF(center.x-rollStringWidth / 2,
                                           innerBoundingBox.top+1+textHeight);

      canvas.drawText(rollString,
                      rollStringCenter.x, rollStringCenter.y,
                      textPaint);
    }
    // Otherwise draw a marker line
    else {
      canvas.drawLine(center.x, (int)innerBoundingBox.top,
                      center.x, (int)innerBoundingBox.top + 5,
                      markerPaint);
    }

    canvas.rotate(10, center.x, center.y);
  }
  canvas.restore();
```

16. The final step in creating the face is drawing the heading markers around the outside edge.

```
canvas.save();
canvas.rotate(-1*(bearing), px, py);

double increment = 22.5;

for (double i = 0; i < 360; i += increment)
{
  CompassDirection cd = CompassDirection.values()[(int)(i / 22.5)];
  String headString = cd.toString();

  float headStringWidth = textPaint.measureText(headString);
  PointF headStringCenter = new PointF(center.x - headStringWidth / 2,
                                       boundingBox.top + 1 + textHeight);

  if (i % increment == 0)
    canvas.drawText(headString,
                    headStringCenter.x, headStringCenter.y,
                    textPaint);
  else
    canvas.drawLine(center.x, (int)boundingBox.top,
                    center.x, (int)boundingBox.top + 3,
                    markerPaint);

  canvas.rotate((int)increment, center.x, center.y);
}
canvas.restore();
```

17. With the face complete, you get to add some finishing touches.

Start by adding a "glass dome" over the top to give the illusion of a watch face. Using the radial gradient array you constructed earlier, create a new `Shader` and `Paint` object. Use them to draw a circle over the inner face that makes it look like it's covered in glass.

```
RadialGradient glassShader = new RadialGradient(px, py,
                                                (int)innerRadius,
```

```
                                                    glassGradientColors,
                                                    glassGradientPositions,
                                                    TileMode.CLAMP);
        Paint glassPaint = new Paint();
        glassPaint.setShader(glassShader);

        canvas.drawOval(innerBoundingBox, glassPaint);
```

18. All that's left is to draw two more circles as clean borders for the inner and outer face boundaries. Then restore the canvas to upright, and finish the `onDraw` method.

```
        // Draw the outer ring
        canvas.drawOval(boundingBox, circlePaint);

        // Draw the inner ring
        circlePaint.setStrokeWidth(2);
        canvas.drawOval(innerBoundingBox, circlePaint);

        canvas.restore();
    }
```

Bringing Map Overlays to Life

In Chapter 7, you learned how to use Overlays to add annotation layers to MapViews. The Canvas used for annotating MapView Overlays is the same class as the one used to draw new View controls. As a result, all of the advanced features described so far in this section can be used to enhance map Overlays.

That means you can use any of the draw methods, transparency, Shaders, Color Masks, and Filter Effects to create rich Overlays using the Android graphics framework.

Interacting with Overlays

Touch-screen interaction in MapViews is handled individually by each of its Overlays. To handle map taps within an Overlay, override the `onTap` event.

The following code snippet shows an `onTap` implementation that receives the map coordinates of the tap and the `MapView` on which the tap occurred:

```
@Override
public boolean onTap(GeoPoint point, MapView map) {
  // Get the projection to convert to and from screen coordinates
  Projection projection = map.getProjection();

  // Return true if we handled this onTap()
  return [ … hit test passed … ];
}
```

The `MapView` can be used to obtain the Projection of the map when it was tapped. Used in conjunction with the `GeoPoint` parameter, you can determine the position on screen of the real-world longitude and latitude pressed.

The `onTap` method of an Overlay derived class should return `true` if it has handled the tap (and `false` otherwise). If none of the Overlays assigned to a MapView return true, the tap event will be handled by the MapView itself, or failing that, by the Activity.

Introducing `SurfaceView`

Under normal circumstances, your applications' Views are all drawn on the same GUI thread. This main application thread is also used for all user interaction (such as button clicks or text entry).

In Chapter 8, you learned how to move blocking processes onto background threads. Unfortunately, you can't do this with the `onDraw` method of a View, as modifying a GUI element from a background thread is explicitly disallowed.

When you need to update the View's UI rapidly, or the rendering code blocks the GUI thread for too long, the `SurfaceView` class is the answer. A Surface View wraps a `Surface` object rather than a `Canvas`. This is important because Surfaces can be drawn onto from background threads. This is particularly useful for resource-intensive operations, or where rapid updates or high frame rates are required, such as when using 3D graphics, creating games, or previewing the camera in real time.

The ability to draw independently of the GUI thread comes at the price of additional memory consumption, so while it's a useful — sometimes necessary — way to create custom Views, Surface Views should be used with caution.

When Should You Use the `SurfaceView`?

A `SurfaceView` can be used in exactly the same way as any View-derived class. You can apply animations and place them in layouts as you would any other View.

The `Surface` encapsulated by the SurfaceView supports drawing, using most of the standard Canvas methods described previously in this chapter, and also supports the full OpenGL ES library.

Using OpenGL, you can draw any supported 2D or 3D object onto the Surface, relying on hardware acceleration (where available) to significantly improve performance compared to simulating the same effects on a 2D canvas.

SurfaceViews are particularly useful for displaying dynamic 3D images, such as those featured in applications like Google Earth, or featured in interactive games that provide immersive experiences. It's also the best choice for displaying real-time camera previews.

Creating a New `SurfaceView` Control

To create a new `SurfaceView`, create a new class that extends `SurfaceView` and implements `SurfaceHolder.Callback`.

The `SurfaceHolder` callback notifies the View when the underlying Surface is created and destroyed, passing a reference to the `SurfaceHolder` object that contains the currently valid Surface.

A typical Surface View design pattern includes a `Thread`-derived class that accepts a reference to the current `SurfaceHolder` and independently updates it.

The following skeleton code shows a Surface View implementation for drawing using the Canvas. A new Thread-derived class is created within the Surface View control, and all UI updates are handled within this new class.

```
import android.content.Context;
import android.graphics.Canvas;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class MySurfaceView extends SurfaceView implements SurfaceHolder.Callback {

  private SurfaceHolder holder;
  private MySurfaceViewThread mySurfaceViewThread;
  private boolean hasSurface;

  MySurfaceView(Context context) {
    super(context);
    init();
  }

  private void init() {
    // Create a new SurfaceHolder and assign this class as its callback.
    holder = getHolder();
    holder.addCallback(this);
    hasSurface = false;
  }

  public void resume() {
    // Create and start the graphics update thread.
    if (mySurfaceViewThread == null) {
      mySurfaceViewThread = new MySurfaceViewThread();

      if (hasSurface == true)
        mySurfaceViewThread.start();
    }
  }

  public void pause() {
    // Kill the graphics update thread
    if (mySurfaceViewThread != null) {
      mySurfaceViewThread.requestExitAndWait();
      mySurfaceViewThread = null;
    }
  }

  public void surfaceCreated(SurfaceHolder holder) {
    hasSurface = true;
    if (mySurfaceViewThread != null)
      mySurfaceViewThread.start();
  }

  public void surfaceDestroyed(SurfaceHolder holder) {
    hasSurface = false;
```

```
    pause();
  }

  public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    if (mySurfaceViewThread != null)
      mySurfaceViewThread.onWindowResize(w, h);
  }

  class MySurfaceViewThread extends Thread {
    private boolean done;

    MySurfaceViewThread() {
      super();
      done = false;
    }

    @Override
    public void run() {
      SurfaceHolder surfaceHolder = holder;

      // Repeat the drawing loop until the thread is stopped.
      while (!done) {
        // Lock the surface and return the canvas to draw onto.
        Canvas canvas = surfaceHolder.lockCanvas();

        // TODO: Draw on the canvas!

        // Unlock the canvas and render the current image.
        surfaceHolder.unlockCanvasAndPost(canvas);
      }
    }

    public void requestExitAndWait() {
      // Mark this thread as complete and combine into
      // the main application thread.
      done = true;
      try {
        join();
      } catch (InterruptedException ex) { }
    }

    public void onWindowResize(int w, int h) {
      // Deal with a change in the available surface size.
    }
  }
}
```

Creating 3D Controls with `SurfaceView`

Android includes full support for the OpenGL ES 3D rendering framework including support for hardware acceleration on devices that offer it. The `SurfaceView` control provides a Surface onto which you can render your OpenGL scenes.

OpenGL is commonly used in desktop applications to provide dynamic 3D interfaces and animations. Resource-constrained devices don't have the capacity for polygon handling that's available on desktop PCs and gaming devices that feature dedicated 3D graphics processors. Within your applications, consider the load your 3D `SurfaceView` will be placing on your processor, and attempt to keep the total number of polygons being displayed, and the rate at which they're updated, as low as possible.

Creating a Doom clone for Android is well out of the scope of this book, so I'll leave it to you to test the limits of what's possible in a mobile 3D User Interface. Check out the `GLSurfaceView` API Demo example included in the SDK distribution to see an example of the OpenGL ES framework in action.

Creating Interactive Controls

Anyone who's used a mobile phone will be painfully aware of the challenges associated with designing intuitive User Interfaces for mobile devices. Touch screens have been available on mobiles for many years, but it's only recently that touch-enabled devices have been designed to be used by fingers rather than styluses.

Full physical keyboards have also become common, with the compact size of the slide-out or flip-out keyboard introducing its own challenges.

As an open framework, Android is expected to be available on a wide variety of devices featuring many different permutations of input technologies including touch screens, D-pads, trackballs, and keyboards.

The challenge for you as a developer is to create intuitive User Interfaces that make the most of the whatever input hardware is available, while introducing as little hardware dependence as possible.

The techniques described in this section show how to listen for (and react to) user input from key presses, trackball events, and touch-screen taps using the following event handlers in Views and Activities:

- `onKeyDown` Called when any hardware key is pressed.
- `onKeyUp` Called when any hardware key is released.
- `onTrackballEvent` Triggered by movement on the trackball.
- `onTouchEvent` The touch-screen event handler, triggered when the touch screen is touched, released, or dragged.

Using the Touch Screen

Mobile touch screens have existed since the days of the Apple Newton and the Palm Pilot, although their usability has had mixed reviews. Recently this technology has enjoyed a popular resurgence, with devices like the Nintendo DS and the Apple iPhone using touch screens in innovative ways.

Modern mobiles are all about finger input — a design principle that assumes users will be using their fingers rather than a specialized stylus to touch the screen.

Finger-based touch makes interaction less precise and is often based more on movement than simple contact. Android's native applications make extensive use of finger-based touch-screen interfaces, including the use of dragging motions to scroll lists or perform actions.

To create a View or Activity that uses touch-screen interaction, override the `onTouchEvent` handler as shown in the skeleton code below:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
  return super.onTouchEvent(event);
}
```

Return `true` if you have handled the screen press; otherwise, return `false` to pass events through a stack of Views and Activities until the touch has been successfully handled.

Processing Touch Events

The `onTouchEvent` handler is fired when the user first touches the screen, once each time the position changes, and again when the contact ends.

The action property of the `MotionEvent` parameter reflects which of these event types has triggered the handler. To determine the initiating touch action, call `getAction` on the Motion Event parameter, and compare it to the static `MotionEvent` action values, as shown in the following skeleton code:

```
@Override
public boolean onTouchEvent(MotionEvent event) {

  int action = event.getAction();

  switch (action) {
    case (MotionEvent.ACTION_DOWN)   : // Touch screen pressed
                                       break;
    case (MotionEvent.ACTION_UP)     : // Touch screen touch ended
                                       break;
    case (MotionEvent.ACTION_MOVE)   : // Contact has moved across screen
                                       break;
    case (MotionEvent.ACTION_CANCEL) : // Touch event cancelled
                                       break;

  }

  return super.onTouchEvent(event);
}
```

The Motion Event also includes the coordinates of the current screen contact. You can access these coordinates using the `getX` and `getY` methods. These methods return the coordinate relative to the responding View or Activity; alternatively, `getRawX` and `getRawY` return the absolute screen coordinates. Both techniques are shown in the following code snippet:

```
// Relative screen coordinates.
int xPos = (int)event.getX();
int yPos = (int)event.getY();
```

```
// Absolute screen coordinates.
int xPosRaw = (int)event.getRawX();
int yPosRaw = (int)event.getRawY();
```

The Motion Event parameter also includes the pressure being applied to the screen using `getPressure`, a method that returns a value usually between 0 (no pressure) and 1 (normal pressure).

Depending on the calibration of the hardware, it's possible to return values greater than 1.

Finally, you can also determine the normalized size of the current contact area using the `getSize` method. This method returns a value between 0 and 1, where 0 suggests a very precise measurement and 1 indicates a possible "fat touch" event in which the user may not have intended to press anything.

Tracking Movement

Whenever the current touch contact position, pressure, or size changes, a new `onTouchEvent` is triggered with an `ACTION_MOVE` action.

As well as the fields described previously, the Motion Event parameter can include historical values. This history represents all the movement events that have occurred between the previous `onTouchEvent` and this one, allowing Android to buffer rapid movement changes to provide fine-grained capture of movement data.

You can find the size of the history by calling `getHistorySize`, which returns the number of movement positions available for the current event. You can then obtain the times, pressures, sizes, and positions of each of the historical events, using a series of `getHistorical*` methods and passing in the position index, as shown in the following code snippet:

```
int historySize = event.getHistorySize();

for (int i = 0; i < historySize; i++) {
  long time = event.getHistoricalEventTime(i);
  float pressure = event.getHistoricalPressure(i);
  float x = event.getHistoricalX(i);
  float y = event.getHistoricalY(i);
  float size = event.getHistoricalSize(i);

  // TODO: Do something with each point
}
```

The normal pattern used for handling movement events is to process each of the historical events first, followed by the current Motion Event values, as shown in the following code snippet:

```
@Override
public boolean onTouchEvent(MotionEvent event) {

  int action = event.getAction();

  switch (action) {
    case (MotionEvent.ACTION_MOVE)
    {
      int historySize = event.getHistorySize();
```

```
        for (int i = 0; i < historySize; i++) {
          float x = event.getHistoricalX(i);
          float y = event.getHistoricalY(i);
          processMovement(x, y);
        }

        float x = event.getX();
        float y = event.getY();
        processMovement(x, y);

        return true;
      }
    }

    return super.onTouchEvent(event);
  }

  private void processMovement(float _x, float _y) {
    // Todo: Do something on movement.
  }
```

Android includes two excellent examples of using the touch screen in the `Fingerpaint` and `Touch Paint` API Demos.

Using the `OnTouchListener`

You can listen for touch events without subclassing an existing View by attaching an `OnTouchListener` to any `View` object, using the `setOnTouchListener` method. The following code snippet demonstrates how to assign a new `OnTouchListener` implementation to an existing View within an Activity:

```
myView.setOnTouchListener(new OnTouchListener() {

  public boolean onTouch(View _view, MotionEvent _event) {
    // TODO Respond to motion events
    return false;
  }

});
```

Using the Device Keys and Buttons (Including D-Pad)

Button and key-press events for all hardware keys are handled by the `onKeyDown` and `onKeyUp` handlers of the active Activity or the focused view. This includes keyboard keys, D-pad, volume, back, dial, and hang-up buttons. The only exception is the Home key, which is reserved to ensure that users can never get locked within an application.

To have your View or Activity react to button presses, override the `onKeyUp` and `onKeyDown` event handlers as shown in the following skeleton code:

```
@Override
public boolean onKeyDown(int _keyCode, KeyEvent _event) {
  // Perform on key pressed handling, return true if handled
```

```
    return false
}

@Override
public boolean onKeyUp(int _keyCode, KeyEvent _event) {
  // Perform on key released handling, return true if handled
  return false;
}
```

The `keyCode` parameter contains the value of the key being pressed; compare it to the static key code values available from the `KeyEvent` class to perform key-specific processing.

The `KeyEvent` parameter also includes the `isAltPressed`, `isShiftPressed`, and `isSymPressed` methods to determine if the function, shift, and symbol metakeys are also being held. The static `isModifierKey` method accepts the `keyCode` and determines if this key event was triggered by the user pressing one of these modifier keys.

Using the `OnKeyListener`

To respond to key presses within Views in your Activity, implement an `OnKeyListener`, and assign it to a View using the `setOnKeyListener` method. Rather than implementing a separate method for key-press and key-release events, the `OnKeyListener` uses a single `onKey` event, as shown below:

```
myView.setOnKeyListener(new OnKeyListener() {

  public boolean onKey(View v, int keyCode, KeyEvent event)
  {
    // TODO Process key press event, return true if handled
    return false;
  }
});
```

Use the `keyCode` parameter to find the key pressed. The `KeyEvent` parameter is used to determine if the key has been pressed or released, where `ACTION_DOWN` represents a key press, and `ACTION_UP` signals its release.

Using the Trackball

Many mobile devices offer a trackball as a useful alternative (or addition) to the touch screen and D-pad. Trackball events are handled by overriding the `onTrackballEvent` method in your View or Activity.

Like touch events, trackball movement is included in a `MotionEvent` parameter. In this case, the `MotionEvent` contains the relative movement of the trackball since the last trackball event, normalized so that 1 represents the equivalent movement caused by the user pressing the D-pad key.

Vertical change can be obtained using the `getY` method, and horizontal scrolling is available through the `getX` method, as shown in the following skeleton code:

```
@Override
public boolean onTrackballEvent(MotionEvent _event) {
```

```
    float vertical = _event.getY();
    float horizontal = _event.getX();

    // TODO: Process trackball movement.

    return false;
}
```

Summary

This final chapter has served as a catch-all for some of the more complex Android features that were glossed over in earlier chapters.

You learned more about Android's security mechanisms, in particular, examining the permissions mechanism used to control access to Content Providers, Services, Activities, Intent Receivers, and broadcast Intents.

You explored the possibilities of interprocess communication using the Android Interface Definition Language to create rich interfaces between application components.

Much of the last part of the chapter focused on the `Canvas` class, as some of the more complex features available in the 2D drawing library were exposed. This included an examination of the drawing primitives available and a closer look at the possibilities of the `Paint` class.

You learned to use transparency and create gradient Shaders before looking at Mask Filters, Color Filters, and Path Effects. You also learned how to use hardware acceleration on 2D canvas-based Views, as well as some Canvas drawing best-practice techniques.

You were then introduced to the `SurfaceView` — a graphical control that lets you render graphics onto a surface from a background thread. This led to an introduction of rendering 3D graphics using the OpenGL ES framework and using the Surface View to provide live camera previews.

Finally, you learned the details for providing interactivity within your Activities and View by listening for and interpreting touch screen, track ball, and key press events.

In particular, you learned:

- Some of the possibilities of using the Internet as a data source, or processing middle tier, to keep your applications lightweight and information-rich.
- How to animate Views and ViewGroups using tweened animations.
- How to create frame-by-frame animations.
- Which drawing primitives you can use to draw on a canvas.
- How to get the most out of the `Paint` object using translucency, Shaders, Mask Filters, Color Filters, and Path Effects.
- Some of the best-practice techniques for drawing on the canvas.
- That you can apply hardware acceleration to 2D graphics drawing.

Index

SYMBOLS AND NUMBERS

@ (at), 61
| (pipe), 65
- (hyphen), 63
2D (two-dimensional) graphics
of Android, 8
hardware acceleration for, 380
3D (three-dimensional) graphics
of Android, 8
with SurfaceView , 392–393
9 (nine) patches, 381

A

AAPT (Android Asset Packaging Tool), 42
AbsoluteLayout class, 79
acceleration, 324
accelerometers
Android features, 6
animating sliding user interface example, 370
constants, 322
creating Speedometer, 326–329
defined, 323
using, 324–325
access. *See also* security
Content Provider, 189
exposing data source, 195–197
files in Content Providers, 192
hardware. See hardware APIs
Location Manager, 213
permission tags, 48
phone properties, 338
retrieving Shared Preferences, 161–162
scanning for Wi-Fi hotspots, 349
accuracy
finding Location Providers based on, 212–213
Sensor Manager, 321–322
actions
anonymous, 130–132
making phone calls, 334
monitoring Wi-Fi connectivity, 348
native Activity, 120–121
native broadcast, 135–136
passing responsibility, 124
strings, 132–133
tag, 121
transmitting data messages, 290
active connections, 346–347
active lifetime, 72–73
active processes, 52
active state, 69
Activities
binding to background Services, 258–259
classes, 73
creating, 66–67
creating Compass and artificial horizon, 330–333
creating earthquake viewer, 148–156
creating UIs with Views, 77–78
defined, 46, 76
Dialog-themed, 144
Emergency Responder, 297–314
environment considerations, 34
launching with Intents, 114–121
life-cycle, 68–73
map-based, 224–226
permissions, 355
runtime configuration changes, 64–65
saving state, 162–165
sliding user interface example, 365–370
Speedometer, 326–329
"Where Am I?" example, 214–216
Activity base class, 27
Activity Manager, 35–36
Activity Menus. *See also* menus
adding to to-do list example, 107–112
Context Menus, 105–107
defined, 101–104
activity tags, 47
activityCreator, 42
ActivityGroup, 73
adapters
Bluetooth, 339–345
database, 177–179
introducing, 136–141
summary, 157
ADB (Android Debug Bridge), 42–43
adding Overlays, 234
adding rows, 182–183
address geocoding, 220–223
ADT (Android Developer Tool) plug-in, 21–24
advanced Android development. *See* Android, advanced development
AIDL (Android Interface Definition Language), 356–361
Alarms
automating Emergency Responder, 311–312
background Services, 273–274
updating earthquake example with, 274–276
AlertDialog class
creating earthquake viewer, 148–156
defined, 145–146
alerts
Notifications, 265–273
proximity, 219–220
alpha, 57–58
Amazon Web Services, 361
Android, 1–17
applications. See applications
background, 2–3
Content Providers, 192–194
database design considerations, 180
developing for, 9–11
development framework, 11–16
hardware APIs. See hardware APIs
menu system, 99–101
native applications, 4–5
OHA, 8–9
open platform for mobile development, 4
overview, 1–2
running, 9
SDK features, 5–8
summary, 17
what it isn't, 3
Android, advanced development, 353–398
animating layouts and View Groups, 370–372
animating sliding user interface example, 365–370
animations, 361–365
Canvas drawing, 373–374
Canvas drawing, best practices, 380–381
compass face example, 381–389
hardware acceleration for 2D graphics, 380
improving paint quality with anti-aliasing, 379–380
interactive controls, 393–398
Internet Services, 361
IPC support with AIDL, 356–361
map Overlays, 389–390
overview, 353
painting, 374–379
security, 354–355
skinning applications with themes, 372–373
summary, 398
SurfaceView, 390–393

Android, getting started, 19–44
application types, 29–30
creating first activity, 24–29
developing for, 30–37
development tools, 42–43
overview, 19–20
summary, 44
to-do list example, 37–41
what you need to begin, 20–24
Android Asset Packaging Tool (AAPT), 42
Android Debug Bridge (ADB), 42–43
Android Developer Tool (ADT) plug-in, 21–24
Android Dialog, 143–147
Android Interface Definition Language (AIDL), 356–361
animations
advanced development, 361–365
animating layouts and View Groups, 370–372
animating sliding user interface example, 365–370
Canvas drawing, best practices, 381
externalizing, 57–59
skinning applications with themes, 372–373
annotations. See Overlays
anonymous actions, 130–132
answering calls, 339
anti-aliasing, 379–380
APIs (Application Programming Interfaces)
Android libraries, 15–16
hardware. See hardware APIs
SDK contents, 12
application layers
vs. Android, 3
decoupling with Content Providers, 189
in software stack, 14
application manifest, 134
Application Programming Interfaces (APIs). See APIs (Application Programming Interfaces)
Application Unresponsive dialog, 133
applications, 45–73
Activity classes, 73
Activity creation, 66–67
Activity life cycle, 68–73
application manifest, 46–49
architecture, 14–15
data storage, retrieval and sharing. See data storage, retrieval and sharing
defined, 46
externalizing resources, 52–59
life-cycle, 50–51
Manifest Editor, 49–50
overview, 45
priority and process states, 51–52
resources for different languages and hardware, 63–64
running/debugging, 26–27
runtime configuration changes, 64–65
skinning with themes, 372–373
summary, 73
to-do list example, 37–41
to-do list resources example, 62–63
types, 44
using resources, 59–62
applying tweened animations, 364
arc drawing, 374
Array List of Strings, 300
ArrayAdapter
creating earthquake viewer, 152
defined, 136–141
Emergency Responder example, 300–301
setting preferences, 168–169
artificial horizons
creating, 330–333
updating compass example, 385
at (@), 61
attributes
application manifest, 47–49
Intent Filter, 121–122
managing with Manifest Editor, 50
referencing resources, 61
runtime configuration changes, 65
audio alert Notifications, 270–271
audio recording, 317–319
authority, Content Provider, 190, 197
auto-retry functionality, 304–306

B

background animations, 372
background processes, 52
background Services, 249–277
Alarms, 273–274
of Android, 6–7
binding Activities to, 258–259
creating and controlling, 250–252
defined, 29–30
earthquake monitoring service example, 252–258
environment considerations, 34
notifications, 265–273
overview, 249–250
summary, 276–277
toasts, 262–264
unique Android features, 10
updating earthquake example with Alarms, 274–276
worker threads, 259–262
bandwidth, 142
base classes, 27
binding
Activities to Services, 258–259
adapters, 136–141
exposing IPC interface, 360–361
to GTalk Service, 281–282
bitmaps
Bitmap object, 373
Canvas drawing, best practices, 381
externalizing, 56
black theme, 373
blocking contacts, 285
Bluetooth, 339–345
blurring Paint, 377–378
bonding Bluetooth devices, 341–342
Broadcast Intents
communicating with Services, 259
defined, 132–136
enforcing permissions, 355
instant messaging, 280
listening for SMS messages, 294–296
P2P communication, 289–291
Broadcast Receivers
Alarms and, 273
automating Emergency Responder, 311
defined, 46
Emergency Responder example, 301–302, 305–306
listening for broadcasts with, 133–135
listening for SMS messages, 295–296
monitoring network connectivity, 346–347
OTA Intents, 289–291
permissions, 355
receiver tags, 48
summary, 157
tracking SMS messages, 292–294
Browser Content Provider, 192
button interaction, 396–397

C

caching
Internet resources, 142
mobile device storage, 31
callbacks, 320–321
CallLog Content Provider, 192
calls, phone
Android telephony, 333–339
using Bluetooth, 339–345
cameras
Android features, 6
using, 319–321
canceling Alarms, 274
Canvas drawing
Android, advanced development, 373–374
best practice, 380–381
compass face example, 381–389
controls, 89–90
hardware acceleration for 2D graphics, 380
improving paint quality with anti-aliasing, 379–380
map Overlays, 389–390
painting, 374–379
SurfaceView , 390–392
categories, Intent, 121
cell location, 336
Chat Listeners
defined, 281
managing group chats, 288–289
receiving text messages, 287
chat rooms, 287–288

Chat Sessions, 281, 286–289
checkboxes
menu items, 102–103
preferences, 168–169
child threads, 259–262
circle drawing, 374
CLAMP, 377
classes
Activity, 73
animating layouts and View Groups, 370
Animation, 362–364
BluetoothDevice, 339–345
CAMERA, 319–321
Canvas. See Canvas drawing
Content Providers, 192–193
Cursor, 176–177
Dialog, 144–147
extending Activity, 66–67
layout, 79
Linkify, 116–117
map, 224
Overlay. See Overlays
passing custom objects, 356–358
Phone, 338–339
Service. See Services
Vibrator, 350–351
View and SurfaceView, 88
clear text functionality, 86–87
click handling. See Overlays
click listeners
contact example, 127–128
menu items, 103
coarse permissions, 213–214
code
registering Broadcast Receivers in, 134–135
using resources in, 59–60
ColorFilters, 378
colors
creating simple values, 55
updating compass example, 382–384
using translucency, 375
column definitions, 176
communication
Bluetooth, 342–344
interapplication, 7
P2P communication. See P2P (peer-to-peer) communication
compasses
Canvas drawing example, 381–389
Compass View example, 93–98
creating, 330–333
defined, 323
determining orientation, 329–330
orientation sensors, 329–330
components, application, 46–49
compound controls, 85–87
condensed titles, 103
configurations, Wi-Fi, 350
confirmation of SMS delivery, 292–294
connectivity
Bluetooth, 339–345
GTalk Connections, 281, 282–286
Internet resources, 141–143
mobile device limitations, 32–33
monitoring mobile data, 337–338
network and Wi-Fi, 345–350
consistency
externalizing strings to maintain, 54
with themes, 372–373
contacts
Content Provider, 192–194
example, 124–129
roster, 283–286
Content Providers
contact roster, 283–286
creating earthquake, 197–205
creating new, 194–197
defined, 7, 46, 160
introducing, 189–192
Media Store, 318
native, 192–194
permissions, 355
provider tags, 48
updating with Services, 256–258
Content Resolvers
creating new Content Provider, 195
defined, 190
using earthquake Content Provider, 203–204
Content Values, 176–177
ContentSlider Activity, 365–370
Context Menus
defined, 105–107
to-do list example, 107–112
Controller, Map, 227–228
controls
Android widgets, 78–79
background Services, 250–252
Bluetooth, 340
creating compound, 85–87
creating custom, 88
device vibration, 350–351
interactive, 393–398
Map View, 224–226
phone, 338–339
specialist Dialog boxes, 146
SurfaceView , 390–393
using custom, 98–99
Views. See Views
costs
finding Location Providers based on, 212–213
mobile device limitations, 33
Criteria
finding Location Providers based on, 212–213
updating location example, 217–218
current themes, 62
Cursors
binding Views to, 136, 141
Content Values and, 176–177
extracting results, 181–182
querying Content Providers, 190
querying databases, 181
custom actions, 290
custom class objects, 356–358
custom controls
Compass View example, 93–98
creating, 88
using, 98–99
custom link strings, 116–117
custom Toasts, 263–264

D

Dalvik, 11–12
Dalvik Debug Monitoring Service (DDMS)
ADT features, 22
defined, 42–43
Dalvik Virtual Machine (VM), 14
data
AIDL support types, 356
binding, 136–141
monitoring connectivity, 337–338
tags, 122
transmitting with Bluetooth, 343–344
data messaging
with Google Talk, 8
handling SMS, 296–297
sending and receiving, 289–291
sending SMS, 294
data sources
Content Providers as, 189
exposing access to, 195–197
playing media, 316–317
data storage, retrieval and sharing, 159–206
Content Providers, 189–192
Content Providers, creating earthquake, 197–205
Content Providers, creating new, 194–197
Cursors and content values, 176–177
database design considerations, 180
database querying, 181
databases, 175
databases, working with, 177–179
developing for mobile devices, 31
extracting Cursor results, 181–182
native Content Providers, 192–194
overview, 159–160
preferences, creating and saving, 161
preferences, creating for earthquake viewer, 165–174
retrieving Shared Preferences, 161–162
rows, adding/updating/removing, 182–183
saving Activity state, 162–165
saving and loading files, 174–175
saving application data, 160–161
saving techniques, 160
saving to-do list example, 183–189
SQLite, 7, 176
SQLiteOpenHelper, 179–180
summary, 205–206

databases
creating earthquake Content Provider, 199–200
Cursors and Content Values, 176–177
design considerations, 180
extracting Cursor results, 181–182
introducing, 175
querying, 181
rows, adding/updating/removing, 182–183
saving to-do list example, 183–189
SQLite, 176
SQLiteOpenHelper, 179–180
working with, 177–179
DatePickerDialog, 146
DDMS (Dalvik Debug Monitoring Service)
ADT features, 22
defined, 42–43
debugging
ADT features, 22
Android applications, 26–27
launch configurations, 25–26
SDK tools, 42–43
simulating SMS messages, 296
to-do list example, 38
declaring permissions, 355
definitions, AIDL, 358–359
deleteFile, 175
deleting
Content Providers, 191
creating earthquake Content Provider, 201–203
rows, 183
delivery, SMS message, 292–294, 305
design
database considerations, 180
hardware-imposed considerations, 30–31
UI, 76
development
advanced Android. See Android, advanced development
for Android, 9–11, 30–37
Android as open platform for mobile, 4
Android framework, 11–16
tools, 42–43
device vibration control, 350–351
dialers, 333–334
Dialog boxes
creating earthquake viewer, 154–156
introducing, 143–147
toasts, 262–264
using background worker threads, 259–262
digital cameras
Android features, 6
using, 319–321
dimensions
creating simple values, 55
determining orientation, 329–330
direction monitoring, 323
discovering Bluetooth devices, 340–341
dismissing calls, 339
documentation, SDK, 12
downloading what you need to begin, 20–24
D-pad
interactive controls, 396–397
wiring up, 369–370
drawables
externalizing, 56
frame-by-frame animations, 59, 372
drawing
Canvas. See Canvas drawing
controls, 89–90
current location, 235–236
customizing to-do list, 84–85
on Overlay Canvas, 233
dx, 42

E

earthquake example
adding Notifications, 267–270
creating Content Providers, 197–205
creating preference page for, 165–174
creating viewer, 148–156
IPC support with AIDL, 356–361
mapping, 242–247
monitoring service example, 252–258
updating with alarms, 274–276
Eclipse IDE (integrated development environment)
ADT plug-in, 21–24
developing with, 21
getting started, 19
SDK and, 12
editors, 161
efficiency, 31, 35
element access, 28–29
embossing Paint, 377–378
Emergency Responder example
automating, 306–314
creating SMS application, 297–306
creating Speedometer, 326–329
empty processes, 52
Emulators
ADT features, 22
audio playback, 317
defined, 42–43
Emergency Responder testing, 306
mobile device limitations, 32–33
recording media, 318
SDK contents, 12
setting up with Test Providers, 208–211
SMS messages, 296
vibrating Notifications, 271
ending calls, 339
enforcing permissions, 355
environment considerations, 33–34
event broadcasting, 132–136
event handlers
Context Menu selections, 106–107
Dialog boxes, 146–147
interactive controls, 393–398
map taps, 234
menu item selections, 104
MenuItemClickListener, 103
modifying existing Views, 81–82
monitoring state changes, 69–71
phone state, 335
saving and restoring instance states, 162–163
saving application data, 160–161
Shared Preferences, 170
sub-Activity results, 119–120
user interaction, 92–93
events, system
broadcasting with Intents, 114
native broadcast actions, 135–136
ExpandableListActivity, 73
expanded menu, 100–101
explicit Intents
defined, 114
starting new Activities, 115
exposing IPC interface, 358–359
extensibility
using Intent Filters for, 130–132
using SQLiteOpenHelper, 179–180
eXtensible Markup Language (XML) layouts, 79–80
eXtensible Markup Language (XML) resources
Android project, 28
using, 59–62
Extensible Messaging and Presence Protocol (XMPP), 7
externalizing resources, 52–59
extras
defined, 133
transmitting data messages, 290

F

feedback with vibration, 350–351
fileList, 175
files
accessing in Content Providers, 192
database design considerations, 180
loading and saving, 174–175
media APIs, 316–319
saving data, 160
FILL, 374
filters
Intent. See Intent Filters
Mask and Color, 377–378
Match and Transform, 117
fine permissions, 213–214
finger input, 393–396
flashing light Notifications, 272
fontscale, 65
foreground Activities
defined, 29–30
environment considerations, 34
format support, 316
Forward Geocoding, 220–222
frame-by-frame animations
creating and using, 372
defined, 362
externalizing, 57, 59

FrameLayout class, 79
Fry, Stephen, 75
full Activity lifetime, 71–72
functions
  Cursor class, 176–177
  geocoding, 220–223
future of Android, 3

G

gData Services, 361
geocoding, 6, 220–223
global positioning services (GPS). ***See also*** **LBS (location-based Services)**
  Android features, 6
  "Where Am I?" example, 214–216
Google
  Android and, 3
  Internet Services, 361
Google Maps
  in Android, 6
  map-based Activities, 224–226
  unique Android features, 10
Google Talk. ***See*** **GTalk Service**
GPS (global positioning services). *See also* LBS (location-based Services)
  Android features, 6
  "Where Am I?" example, 214–216
GPX (GPS Exchange Format), 209
gradient Shaders, 376–377
graphical user interfaces (GUIs)
  creating Speedometer, 328
  synchronizing threads for, 260–261
graphics
  Android, 8
  controls, 32
  hardware acceleration for 2D, 380
  with SurfaceView , 392–393
gravity and acceleration, 324
group chats
  GTalk Service, 281
  managing, 287–289
Groups, View. ***See*** **View Groups**
GTalk Connections
  defined, 281
  GTalk Service, 282–286
GTalk Service, 7–8
  binding to, 281–282
  Chat Sessions, 286–289
  GTalk Connections, 282–286
  introducing, 280
  sending and receiving data messages, 289–291
  using, 280–281
GUIs (graphical user interfaces)
  creating Speedometer, 328
  synchronizing threads for, 260–261

H

handling events. ***See*** **event handlers**
handsets, 9
haptic feedback, 350–351
hardware
  acceleration for 2D graphics, 380
  Android features, 6
  Android security, 36
  imposed design considerations, 30–31
  resources for different, 63–64
  runtime configuration changes, 64–65
hardware APIs, 315–351
  accelerometers, 324–325
  accelerometers and compasses, 323
  Bluetooth, 339–345
  cameras, 319–321
  creating compass and artificial horizon, 330–333
  creating Speedometer, 326–329
  device vibration control, 350–351
  media APIs, 316–319
  network and Wi-Fi connections, 345–350
  orientation sensors, 329–330
  overview, 315
  Sensor Manager, 321–323
  summary, 351
  telephony, 333–339
heading
  adding to Compass View, 332
  compass face example, 388
  orientation sensors, 329
headsets, Bluetooth, 344–345
Hello World template, 27–29
helper classes
  creating for SQLite, 177–179
  using SQLiteOpenHelper, 179–180
history of Android, 2–3
horizons, artificial
  creating, 330–333
  updating compass example, 385
hotspots, 349
hyperlinks
  adding to earthquake viewer, 156
  creating with Linkify, 116–117
hyphen (-), 63

I

icons
  creating Notifications, 266–267
  menu, 99–100
  menu items, 103
  resources, 62–63
IDE (integrated development environment), Eclipse. ***See*** **Eclipse IDE (integrated development environment)**
identifiers
  dimension, 55
  menu items, 102–104
  URIs, See URIs (Uniform Resource Identifiers)
idle phones, 336
IM (instant messaging), 7–8
  binding to GTalk Service, 281–282
  Chat Sessions, 286–289
  GTalk Service, 280–281
  sending and receiving data messages, 289–291
  starting sessions, 282–286
images
  creating, 62
  externalizing, 56
  taking pictures, 320–321
implicit Intents
  defined, 114
  late runtime binding and, 115
  servicing with Intent Filters, 121–129
inactive state, 69
inflating layouts, 86
input
  Android security, 36
  defining sources, 318
  touch screen, 393–396
inserting
  creating earthquake Content Provider, 201–203
  methods, 191
  rows, 182
insistent Notifications, 272–273
instances
  binding Activities to Services, 258–259
  creating, 60
  getting provider, 212
  saving and restoring state, 162–163
instant messaging (IM). ***See*** **IM (instant messaging)**
instrumentation classes, 48–49
integrated development environment (IDE), Eclipse. *See* Eclipse IDE (integrated development environment)
Intent Filters
  listening for SMS messages, 296
  populating Context Menus, 106
  servicing implicit Intents with, 121–129
  using for plug-ins and extensibility, 130–132
Intents
  broadcasting events, 132–136
  broadcasting OTA, 289–291
  broadcasting with Services, 256
  communicating with Services, 259
  creating Activity, 67
  defined, 7, 46
  enforcing permissions, 355
  instant messaging, 280
  making phone calls, 334
  menu items, 104
  monitoring Wi-Fi connectivity, 348
  proximity alerts, 219–220
  servicing implicit, 121–129
  summary, 157
  using Alarms to fire, 273–274
  using to launch Activities, 114–121
interactive controls, 393–398

interapplication communication, 7
interfaces
APIs. See APIs (Application Programming Interfaces)
GTalk Service, 280–281
implementing AIDL, 356–361
user. See UIs (user interfaces)
intermittent Activities, 29–30
Internet
monitoring connectivity, 345–346
services, 361
Internet resources
creating earthquake viewer, 153
introducing, 141–143
summary, 157
interprocess communication (IPC)
support with AIDL, 356–361
uniqueness of Android, 11
Invitation Listener, 288
IPC (interprocess communication)
support with AIDL, 356–361
uniqueness of Android, 11
iPhone, 3
ItemizedOverlays, 224, 239–240

J

Java, 20
Java IDE (integrated development environment), 21
Java ME, 3
Java MIDlets, 2–3
JDK (Java Development Kit)
downloading, 20
getting started, 19

K

keyboardHidden, 65
keys
interactive controls, 396–397
shortcut, 103
KML (Keyhole Markup Language), 209

L

languages
resources for different, 63–64
runtime configuration changes, 64–65
latency, 32–33
lateral sensor changes, 325
latitude
extracting, 215–216
geocoding, 220–223
MapController, 227
launch configurations, 25–26
Layout Managers, 79
layouts
animating, 370–372
compound controls, 86
creating Activity, 66–67
creating Activity UIs, 77–78
creating UIs, 79–80
externalizing resources with, 52, 57
Map Views, 225–226
referencing resources, 61
using adapters, 136–141
using custom controls, 98–99
LBS (location-based Services), 207–247
in Android, 6
creating map-based Activity, 224–226
finding location, 213–219
Geocoder, 220–223
ItemizedOverlays/OverlayItems, 239–240
Map Controller, 227–228
Map Views, 226–227
mapping earthquakes example, 242–247
mapping "Where Am I?", 228–231
maps-based Activities, 224
My Location Overlay, 239
Overlays, 231–238
overview, 207–208
pinning Views to maps, 240–242
proximity alerts, 219–220
selecting Location Provider, 212–213
setting up Emulator with Test Providers, 208–211
summary, 247
LED (light-emitting diode) Notifications, 272
leverage, 143
libraries
Android APIs, 15–16
Android media support, 8
GTalk Service, 280–281
software stack, 12–13
SQLite, 176
life-cycles
Activities, 68–73
application, 50–51
light Notifications, 272
light sensors, 322
light theme, 373
light-emitting diode (LED) Notifications, 272
line drawing, 374
LinearLayout, 57, 79
Linkify class, 116–117
Linux Kernel
security, 354
software stack, 12–13
Linux Phone Standards Forum (LiPS), 3
LiPS (Linux Phone Standards Forum), 3
ListActivity, 73
listeners
adding to Compass View, 333
animation, 364–365
Animation, 371
Bluetooth, 343
Chat. See Chat Listeners
click, 103, 127–128
creating Speedometer, 327
Location, 216–217
monitoring acceleration, 325
monitoring network connectivity, 346–347
OnKeyListener, 397
OnTouchListener, 396
orientation sensors, 330
phone state, 335–339
Sensor Manager, 321–322
SMS messages, 294–297
live video, 320
loading files, 174–175
Locale
geocoding, 221–222
locale, 65
location
runtime configuration changes, 64–65
tracking cell, 336
Location Manager, 208
Location Providers
defined, 208
managing Test, 209–211
selecting, 212–213
location-based Services (LBS). See LBS (location-based services); LBS (location-based Services)
longitude
extracting, 215–216
geocoding, 220–223
MapController, 227
sensor changes, 325

M

magnetic field sensor, 322
magnitude filter, 167, 172–173
Managers
Activity, 35–36
Layout, 79
Location, 208
Notification, 266
Sensor. See Sensor Manager
Telephony, 334–339
Wi-Fi, 347–350
Window, 35–36
Manifest Editor, 49–50
Map Views
configuring and using, 226–227
defined, 224
Overlays. See Overlays
MapActivity, 73, 224
MapController
defined, 224
using, 227–228
maps
based Activities, 224
creating Activity, 224–226
earthquakes example, 242–247
geocoding, 220–223
ItemizedOverlays/OverlayItems, 239–240
Map Controller, 227–228
Map Views, 226–227
My Location Overlay, 239
Overlays, 231–238
pinning Views to, 240–242
"Where Am I?", 228–231

markers
creating with Overlays, 239–240
pinning Views to maps, 240–242
MaskFilters, 377–378
Match Filter, 117
matching URIs, 195, 200
measuring Views, 90–92
media
Android support, 8
APIs, 316–319
Media Player, 316–317
Media Store Content Provider
defined, 193
recording media, 318
memory
optimization and management, 8
speed and efficiency, 35
MenuItemClickListener, 103
menus
Activity, 101–104
adding preferences, 171
Android, 99–101
creating earthquake viewer, 153–154
dynamic updating and handling selections, 104
mapping earthquakes example, 243–244
populating with Intent Filters, 130–132
submenus and context menus, 105–107
to-do list example, 107–112
mesh drawing, 374
messaging
instant. *See* IM (instant messaging)
with Intents. *See* Intents
P2P communication. *See* P2P (peer-to-peer) communication
SMS. *See* SMS (short messaging service)
Toasts, 262–264
metadata, 47–49
methods
Activity lifetimes, 71–73
adding hyperlinks, 116–117
adding menus to to-do list example, 107–112
advanced development. *See* Android, advanced development
background Services. *See* background Services
broadcasting with Intents, 132–136
creating earthquake viewer, 148–156
creating new visual interface, 88–92
creating submenus and Context Menus, 105–107
data storage, retrieval and sharing. *See* data storage, retrieval and sharing
defining Activity menus, 101–104
Dialog class, 144–147
hardware. *See* hardware APIs
of location-based Services. *See* LBS (location-based Services)
modifying existing Views, 81–82
P2P communication. *See* P2P (peer-to-peer) communication
returning results from Activities, 117–120
runtime configuration changes, 65
understanding Hello World, 27–29
using resources in code, 60
MIDlets, 2–3
MIRROR, 377
MkSDCard, 42
mobile development, 4
mobile devices, 30–37
mobile phone handsets, 3
mobile phones, 1–2
mode values, 91
monitoring phone calls, 335–336
monitoring SMS messages, 294
movement, tracking. *See* tracking movement
multimedia
APIs, 316–319
Media Store Content Provider, 193
MyLocationOverlay, 224, 239
MyView, 67

N

name/value pair (NVP) mechanism, 160
native actions
Activity, 120–121
broadcast, 135–136
native applications
of Android, 4–5
Internet resources, 141–143
uniqueness of Android, 11
native Content Providers, 192–194
native Internet resources, 142
native link types, 116
navigation
configuration changes, 65
menu, 99–101
nested submenus, 105
network connections. *See also* connectivity
Internet resources, 142–143
managing, 345–350
mobile device limitations, 32–33
Network Providers, 212
networks, phone, 338
Nine Patches, 381
NinePatch (stretchable PNG) images, 56
Notification Manager, 266
notifications
background Services, 265–273
defined, 7, 46
Sensor Manager, 322
NVP (name/value pair) mechanism, 160

O

objects
binding Views to arrays, 136–141
creating static, 381
Cursors and Content Values, 176–177
drawing, 374
passing custom class, 356–358
off the hook phones, 336
OHA (Open Handset Alliance), 8–9
OMA (Open Mobile Alliance), 3
ongoing Notifications, 272–273
OnKeyListener, 397
online support, 12
OnTouchListener, 396
opacity, 375
Open Handset Alliance (OHA), 8–9
Open Mobile Alliance (OMA), 3
open platforms, 4
opening databases, 179–180
opening socket connections, 342–343
orientation, 65
orientation sensors
accelerometers and compasses, 323
adding to Compass View, 330–332
constants, 322
defined, 329–330
OTA (over the air), 289
outgoing phone calls, 335–336
output sources, 318
oval drawing, 374
over the air (OTA), 289
OverlayItems, 224, 239–240
Overlays
bringing to life, 389–390
defined, 231–238
ItemizedOverlays/OverlayItems, 239–240
map, 224
mapping earthquakes example, 246
My Location Overlay, 239

P

P2P (peer-to-peer) communication, 279–314
automating SMS Emergency Responder, 306–314
binding to GTalk Service, 281–282
Chat Sessions, 286–289
GTalk Connections, 282–286
GTalk Service, 280–281
instant messaging, 280
overview, 279
sending and receiving data messages, 289–291
Services, 7–8, 11
SMS, 291–297
SMS Emergency Responder example, 297–306
summary, 314
Paint, 373
painting
Canvas drawing, 374–379
improving quality with anti-aliasing, 379–380
pairing Bluetooth devices, 341–342
panning, 227–228
Parcels, 356–358

passing custom class objects, 356–358
passing responsibility, 124
patches, nine, 381
path drawing, 374
PathEffects, 378–379
paused state, 69
peer-to-peer (P2P) communication. ***See*** **P2P (peer-to-peer) communication**
Pending Intents, 292–293
permissions
adding to Emergency Responder, 298
advanced development, 354–355
CAMERA, 319
Content Providers, 189
creating earthquake viewer, 150
developing for Android, 36
listening for SMS messages, 295
Location Manager access, 213–214
maps, 225
permission tags, 48
sending SMS messages, 292
VIBRATE, 350
persistence techniques, data. ***See*** **data storage, retrieval and sharing**
phones
Android telephony, 333–339
smartphones, 10
using Bluetooth, 339–345
picture taking, 320–321
pinning Views to maps, 240–242
pipe (|), 65
Pipes, 361
pitch
adding to Compass View, 330–332
compass face example, 385–386
orientation sensors, 329
platforms, Android support, 20
playback in Android, 316–317
plug-ins
ADT, 21–24
using Intent Filters for, 130–132
politeness, 34
"pop-in" animations, 370–371
power efficiency
defined, 31
finding Location Providers based on, 212–213
preferences
automating Emergency Responder, 308–310
configuring network, 346
creating and saving, 161
creating for earthquake viewer, 165–174
defined, 159–160
retrieving Shared Preferences, 161–162
presence
defined, 283–286
notification, 8
presentation layer
decoupling with layouts, 57
using layouts in, 79–80
previews, camera, 320
primitives, drawing, 374
priority and process states, 51–52
private databases, 183–189
private Shared Preferences, 162
process management, 8
process states, 51–52
processor speed, 31
ProgressDialog, 146
Projections, 232–233
properties, phone, 338
provider tags, 48
Providers, Content. ***See*** **Content Providers**
Providers, Location. See Location Providers
proximity alerts, 219–220
proximity sensor, 322

Q

qualifiers, 64
querying
Content Providers, 190
creating earthquake Content Provider, 201
Cursors and Content Values, 176–177
databases, 181
exposing, 195–197

R

R class file
defined, 53
using resources in code, 59–60
radio buttons, 102–103
radiofrequency communications (RFCOMM), 342–344
raw files, 174–175, 270–271
Read Only file resources, 174–175
receiver tags, 48
receiving messages
data, 289–291
text, 287
recording media, 317–319
rectangle drawing, 374
referencing resources, 61
refreshing earthquake viewer, 150
registration
Broadcast Receiver, 134–135
Content Provider, 197
listener, 335
Wi-Fi configurations, 350
relational database management system (RDBMS), 176. ***See also*** **SQLite**
RelativeLayout class, 79
remote devices, 339–345
removing
contacts, 285
Overlays, 234
rows, 182–183
REPEAT, 377
repeating animations, 364
requirements, Location Providers, 212–213
resolution, Intent, 123
Resolvers, Content. ***See*** **Content Resolvers**
resources
Android project, 28
for different languages and hardware, 63–64
externalizing, 52–59
files as, 174–175
Internet, 141–143
playing media, 316–317
to-do list example, 62–63
using application, 59–62
responding to SMS messages
automating, 306–314
Emergency Responder example, 302–303
responsibility, passing, 124
responsiveness, 34–36
restarting behavior, 64–65
results
extracting from Cursors, 181–182
returning from Activities, 117–120
retrieval. ***See*** **data storage, retrieval and sharing**
returning results, 117–120
Reverse Geocoding, 220–221
RFCOMM (radiofrequency communications), 342–344
rich UIs (user interfaces). See UIs (user interfaces), building rich
ringing
audio alert Notifications, 270–271
monitoring phones, 336
roaming, 337
roll
adding to Compass View, 330–332
compass face example, 385–387
orientation sensors, 329
roster, contact, 283–286
Roster Listeners, 281, 284
rotation of tweened animations, 57–58, 362
rows
adding/updating/removing, 182–183
Cursors and Content Values, 176–177
running
Android, 9
Android applications, 26–27
to-do list example, 38
runtime
Activity life cycle, 68
Android application life cycle, 50
configuration changes, 64–65
late binding and implicit Intents, 115
launch configurations, 25–26
resolving Intent Filters, 123
software stack, 13–14

S

S60, 3
sandboxing
  data storage and retrieval with SQLite, 7
  security, 354
saving
  Activity state, 162–165
  application data, 160–161
  loading files and, 174–175
  preferences, 161
  techniques, 160
scale, 57–58
scanning for Wi-Fi hotspots, 349
screens
  creating application using Activities. *See* Activities
  creating UIs. *See* UIs (user interfaces)
  creating with layouts, 57
  developing for size, 31–32
  Dialog boxes, 144–147
  sliding user interfaces, 368
SDK (software development kit)
  contents, 12
  features, 5–8
  getting started, 19–20
  layouts, 79
seamless user experience
  Activity states, 69
  developing for Android, 36
security
  advanced development, 354–355
  of Android, 36
  Android telephony, 334
  Google Talk and, 8
  permission tags, 48
sending
  data messages, 289–291
  SMS messages, 292–294
  text messages, 286–287
Sensor Manager
  accelerometers. *See* accelerometers
  compasses. *See* compasses
  defined, 321–323
service tags, 47
Services
  binding Activities to, 258–259
  Bluetooth, 339–345
  Camera, 319–321
  creating and controlling, 250–252
  defined, 46
  earthquake example, 252–258
  GTalk. *See* GTalk Service
  Internet, 361
  IPC support with AIDL, 356–361
  location-based. *See* LBS (location-based Services)
  network and Wi-Fi, 345–350
  permissions, 355
  Sensor, 321–323
  tracking telephony changes, 336–337
Sessions, Chat, 281, 286–289
Sessions, IM, 281, 282–286
set tags, 58
Settings Content Provider, 193
Shaders
  compass face example, 382–383
  defined, 375–377
shapes, drawing, 374
shared data. *See* also data storage, retrieval and sharing
  Android features, 7
  uniqueness of Android, 11
Shared Preferences
  adding to earthquake viewer, 165–174
  creating and saving, 161
  defined, 159–160
  retrieving, 161–162
  saving application data, 160–161
short messaging service (SMS). *See* SMS (short messaging service)
shortcut keys, 103
simple values, 53–55
SimpleCursorAdapter, 136–141
simulating SMS messages, 296
sizing
  controls, 90–92
  SMS messages, 294
  Views and Overlays, 381
skinning applications with themes, 372–373
sliding user interfaces, 365–370
smartphones, 10
SMS (short messaging service)
  automating Emergency Responder, 306–314
  defined, 291–297
  Emergency Responder example, 297–306
socket connections, 342–343
software development kit (SDK). *See* SDK (software development kit)
software stack, 12–15
sound Notifications, 270–271
specialist Dialog boxes, 146
speed
  Android development, 35
  determining with accelerometers. *See* accelerometers
  finding Location Providers based on, 212–213
  mobile device limitations, 32–33
Speedometers, 326–329
Spinner controls
  automating Emergency Responder, 308–309
  preferences, 168–169
SQLite
  Android features, 7
  Cursors and Content Values, 176–177
  design considerations, 180
  extracting Cursor results, 181–182
  introducing, 176
  querying, 181
  rows, adding/updating/removing, 182–183
  saving data, 160
  saving to-do list example, 183–189
  SDK tools, 42
  SQLiteOpenHelper, 179–180
  working with, 177–179
SQLiteOpenHelper
  creating earthquake Content Provider, 199–200
  defined, 179–180
stacks
  Activity, 68
  Android software, 12–15
started service processes, 52
starting Services, 251–252
states
  Activity, 68–71
  data persistence techniques. *See* data storage, retrieval and sharing
  monitoring phone, 334–337
  monitoring Wi-Fi connectivity, 348
  priority and process, 51–52
  saving Activity, 162–165
static files, 174–175
static objects, 381
status
  accessing phone, 338
  monitoring Wi-Fi connectivity, 348
  native broadcast actions, 135–136
status bar icons
  adding Notifications to earthquake, 267–270
  Notifications, 265–266
stopped state, 69
stopping Services, 251–252
storage. *See also* data storage, retrieval and sharing
  developing for mobile devices, 31
  with SQLite, 7
street address geocoding, 220–223
strings
  action, 132–133
  custom link, 116–117
  externalizing, 54
  extras, 290
STROKE, 374
structure
  application manifest, 46–49
  submenu, 105
styles
  externalizing, 55–56
  referencing resources, 61
  referring to in current theme, 62
sub-Activities
  contact example, 124–129
  returning results from Activities, 117–120
submenus, 101, 105–107
subscription requests, 285
Surfaces, 320
SurfaceView
  defined, 88
  developing with, 390–393
Symbian, 2

synchronizing threads, 260–261
system events
broadcasting with Intents, 114
native broadcast actions, 135–136
system resources, 59–62

T

TableLayout class, 79
tables, 180
tags
animation, 58
application manifest, 47–49
Intent Filter, 121–122
uses-permission. See uses-permission tags
taps, map, 234
target emulators, 294
telephony, 333–339
Telephony Manager, 334–339
temperature sensor, 322
termination behavior, 64–65
Test Providers, 208–211
testing
with Android Emulator, 42–43
Emergency Responder example, 306
text drawing, 374
text editors, 21
text messaging. *See also* SMS (short messaging service)
Chat Sessions, 286–287
sending, 292
text resources
clearing functionality, 86–87
creating, 62–63
TextViews
creating hyperlinks in, 116–117
extending, 82–85
themes
creating new, 62–63
Dialog-themed Activities, 144
externalizing, 55–56
referring to styles in, 62
skinning applications with, 372–373
third-party applications, 11
threads, background, 259–262
three-dimensional graphics
of Android, 8
with SurfaceView , 392–393
TileModes, 377
TimePickerDialog, 146
T-Mobile G1, 9
toasts
creating, 262–264
defined, 144
To-Do list example
applications, 37–41
customizing with ArrayAdapter, 137–141
extending TextView, 82–85
menus, 107–112
resources, 62–63
saving, 163–165, 183–189
tools
Android widgets, 78–79
development, 42–43
SDK contents, 12
touchscreen, 65
touchscreens
interaction with Overlays, 389–390
interactive controls, 393–396
support, 32
Traceview, 42
trackball interaction, 397–398
tracking movement, 216–217
with accelerometers. See accelerometers
cell location, 336
with compasses. See compasses
touch screens, 395–396
tracking service changes, 336–337
tracking SMS message delivery, 292–294, 305
transactions
creating earthquake Content Provider, 201
exposing, 195–197
Transform Filter, 117
transforms, 381
transient Dialog boxes, 262–264
translate, 57–58
translucency
defined, 375
themes, 373
transmitting data
with Bluetooth, 343–344
messages, 290
transparency
animations, 362
themes, 373
triggers
Alarms, 273–274
Notifications, 267
proximity alerts, 219–220
Sensor Manager, 321–322
tweened animations
defined, 362–364
externalizing, 57–59
two-dimensional graphics
of Android, 8
hardware acceleration for, 380

U

UIQ, 3
UIs (user interfaces)
Activity menu, 101–104
Android menu system, 99–101
Compass View example, 93–98
compound controls, 85–87
creating resources for different languages and hardware, 63–64
custom widgets and controls, 88
Dialog boxes, 144–147
functionality with Activities. See Activities
fundamental design, 76
handling user interaction events, 92–93
layouts, 79–80
menus, dynamic updating and handling selections, 104
new visual interface, 88–92
overview, 75–76
state persistence. See data storage, retrieval and sharing
submenus and Context Menus, 105–107
summary, 112
to-do list example, 107–112
using custom controls, 98–99
Views, 76–79
Views, existing, 81–85
Views, new, 80–81
UIs (user interfaces), building rich
animating layouts and View Groups, 370–372
animating sliding example, 365–370
animations, 361–365
Canvas drawing, 373–374
Canvas drawing, best practices, 380–381
compass face example, 381–389
hardware acceleration for 2D graphics, 380
improving paint quality with anti-aliasing, 379–380
interactive controls, 393–398
map Overlays, 389–390
painting, 374–379
skinning applications with themes, 372–373
SurfaceView, 390–393
UIs (user interfaces) layout
defined, 28–29
environment considerations, 34
externalizing resources with, 57
to-do list example, 39–40
updating
ADT plug-in, 24
Content Providers, 191
creating earthquake Content Provider, 201–203
current location Overlay, 237–238
dynamic menu, 104
earthquake example with Alarms, 274–276
earthquake example with Services, 255–258
location, 217–219
preferences, 166–167
rows, 182–183
Sensor Manager, 322–323
Test Providers, 211
upgrading databases, 179–180

URIs (Uniform Resource Identifiers)
authority, 190
creating earthquake Content Provider, 199–200
matching, 195
user interfaces (UIs). *See* UIs (user interfaces)
users
availability, 283
environment considerations, 33–34
handing interaction, 92–93
ID, 354
techniques for seamless experience, 36
tracking movement, 216–217
uses-permission tags
accessing Internet resources, 142
advanced security, 354
defined, 48
GTalk Service, 280
Location Manager, 213–214

V

values, simple, 53–55
vector graphics, 90
velocity
creating Speedometer, 326–327
defined, 324
vertical sensor changes, 325
vibration
device control, 350–351
Notifications, 271
video
live with camera, 320
recording, 317–319
View class, 88
View Groups
animating, 370–372
compound controls, 85–87
defined, 66–67, 76
Layout Managers, 79
Views
adapters, 136–141
applying tweened animations, 364
Compass View example, 93–98
controls, 37–41
creating Activity, 66–67
creating Compass and artificial horizon, 330–333
creating new, 80–81
creating new visual interface, 88–92
creating UIs, 76–79
defined, 27–29, 76
Map, 224, 226–227
modifying existing, 81–85
SurfaceView class, 390–393
tweened animations, 362
visible Activity lifetime, 72
visible processes
Activity states, 69
defined, 52
visual components, 27–29
visual interface
creating new, 88–92
creating UIs. See UIs (user interfaces)
Manifest Editor, 49–50
VM (Dalvik Virtual Machine), 14

W

waking up Alarms, 273–274
Web Services, 361
"Where Am I?" example
Activities, 214–216
annotating, 234–238
geocoding, 222–223
mapping, 228–231
updating location in, 217–219
widgets
Android toolbox, 78–79
Android-supplied adapters, 136
compound controls, 85–87
custom, 88
leveraging Internet resources, 143
modifying existing Views, 81–85
Views, 76–77
Wi-Fi connections, 345–350
Wi-Fi Manager, 347–350
Window Manager, 35–36
Windows Mobile, 3
worker threads, background
defined, 259–262
using Toasts in, 264

X

Xfermode, 379
XML (eXtensible Markup Language) layouts, 79–80
XML (eXtensible Markup Language) resources
Android project, 28
using, 59–62
XMPP (Extensible Messaging and Presence Protocol), 7

Y

Yahoo! Pipes, 361

Z

zooming
in maps, 227–228
tweened animations, 362



Take your library wherever you go.

Now you can access more than 70 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

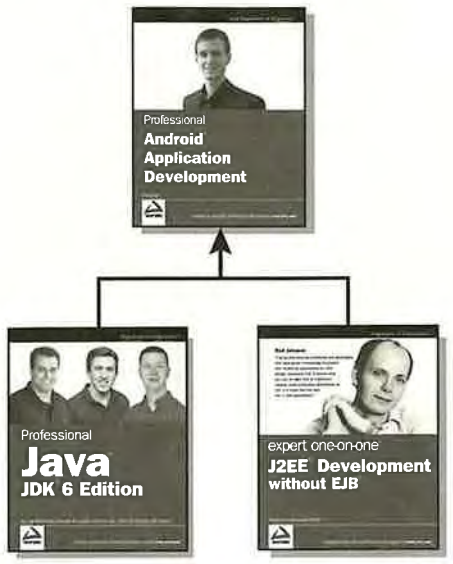
- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML

wrox

www.wrox.com

Professional

Android™ Application Development



Enhance Your Knowledge
Advance Your Career

Professional Android Application Development
978-0-470-34471-2
A hands-on guide to building mobile applications, this book features concise and compelling examples that show you how to quickly construct real-world mobile applications for Android phones. Fully up-to-date for version 1.0 of the Android SDK, it covers all the essential features, and explores the advanced capabilities of Android.

Professional Java JDK 6 Edition
978-0-471-77710-6
Building upon Ivor Horton's Beginning Java 2, this resource shows you how to use the core features of the latest JDK as well as powerful open source tools such as Ant, JUnit, and Hibernate. It will arm you with a well-rounded understanding of the professional Java development landscape.

Expert One-on-One™ J2EETM Development without EJBTM
978-0-7645-5831-3
This hands-on guide shows you alternatives to EJB that can be used to create higher quality applications faster and at lower cost, and demonstrates how to leverage practical techniques and tools, including the popular open source Spring Framework and Hibernate.

Wrox Programmer to Programmer™

Professional Android™ Application Development

Offering an open development environment, Android represents an exciting new opportunity to write innovative applications for mobile devices. This book provides you with a hands-on guide to building these applications using the Android software development kit. It takes you through a series of sample projects, each introducing new features and techniques to get the most out of Android. You'll learn all about the basic functionality as well as discover how to utilize the advanced features with the help of concise and useful examples.

Beginning with an introduction to the Android software stack, the author examines the philosophy behind creating robust, consistent, and appealing applications for mobile phones. You'll get the grounding and knowledge that is needed to write customized mobile applications using the current Android 1.0 SDK. Plus, you'll also gain the flexibility to quickly adapt to future enhancements in order to build the most cutting-edge solutions.



What you will learn from this book

- Best practices for Android mobile development
- An introduction to Activities, Intents, the manifest, and resources
- How to create user interfaces with layouts and custom views
- Techniques to store and share your application data
- Instructions for creating map-based applications, using location-based services including GPS, and geocoding locations
- How to create and use background Services and Notifications
- Working with the accelerometers, compass, and camera hardware
- All about phone and networking hardware such as telephony APIs, SMS, and network management
- Advanced development topics, including security, IPC, and some advanced graphics and user interface techniques

Who this book is for

This book is for anyone interested in creating applications for the Android mobile phone platform. It includes information that will be valuable whether you're an experienced mobile developer or just starting out writing mobile applications.

Wrox Professional guides are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies, all designed to help programmers do a better job.

p2p.wrox.com
The programmer's resource center

www.wrox.com

Recommended Computer Book Categories	Programming Languages
	Java

38 144 AA BR1 5171 05/09 02-013-01 Ohio

Wrox
An Imprint of
WILEY

ISBN: 978-0-470-34471-2
54499
9 780470 344712



THE UNIVERSITY OF MICHIGAN

DATE DUE

OCT 21 2010

UNIVERSITY OF MICHIGAN
3 9015 07947 2265

APPENDIX MEIER03

```
leader  03165cam a2200505 a 4500
001     991005543919706121
005     20250604213024.0
008     080625s2009    inua          000 0 eng
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
019     ##$a1201947138
020     ##$a9780470344712 $q(pbk.)
020     ##$a0470344717 $q(pbk.)
035     ##$aocn226356018-01cuny_network
035     ##$a(OCoLC)226356018 $z(OCoLC)1201947138
035     ##$a(OCoLC)ocn226356018
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM $dWAU $dDEBSZ $dOCLCQ $dLND $dOCLCF $dOCLCQ $dS3O $dOCLCQ $dNTG $dUKMGB $dZHC $dOCLCO $dLDP $dOCLCO $dOCLCL $dOCLCQ $dOCLCL
049     ##$aMAIN
050     #4$aQA76.76.A65 $bM43 2009
082     04$a005.1 $222 $1https://id.oclc.org/worldcat/ddc/E3jXFdk4KYFKwJQWKPH9rDFJPB
100     1#$aMeier, Reto, $d1978- $1https://id.oclc.org/worldcat/entity/E39PCjtMKtWB3fgDjJCFfbg7pP
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $c©2009.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides
500     ##$a"Wrox programmer to programmer"--Cover.
505     0#$aCh. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet --
520     1#$a"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
650     #7$aApplication software $xDevelopment. $2fast $0(OCoLC)fst00811707
650     #7$aMobile computing. $2fast $0(OCoLC)fst01024221
650     #7$aAndroid $gSystemplattform $2gnd
650     #7$aGratisprogram. $2sao
650     #7$aOperativsystem. $2sao
650     #7$aTillämpningsprogram. $2sao
630     00$aAndroid (Electronic resource)
630     07$aAndroid (Electronic resource) $2fast $0(OCoLC)fst01776064
758     ##$ihas work: $aProfessional Android application development (Text) $1https://id.oclc.org/worldcat/entity/E39PCGgYppd9Pt3ybMHbFyYgKd $4https://id.oclc.org/worldcat/ontology/hasWork
830     #0$aWrox professional guides.
901     ##$a006082470 $a006391479
999     ##$aOCLC Daily new or updates
```



BOOK

Professional Android application development

Meier, Reto, 1978-

©2009

May be available; see options

Send to

EMAIL CITATION PRINT PERMALINK REFWORKS ENDNOTE EASYBIB EXPORT RIS EXPORT BIBTEX

How to get it

Sign in to place requests Sign in

OTHER CUNY LIBRARIES

BACK

BACK TO LOCATIONS

LOCATION ITEMS

Ursula C. Schwerin Library
Available , 4th or 5th Floor, Stacks, 16 Week Loan ; QA76.76 .A65 M43 2009
(1 copy, 1 available, 0 requests)

Item in place
Loanable

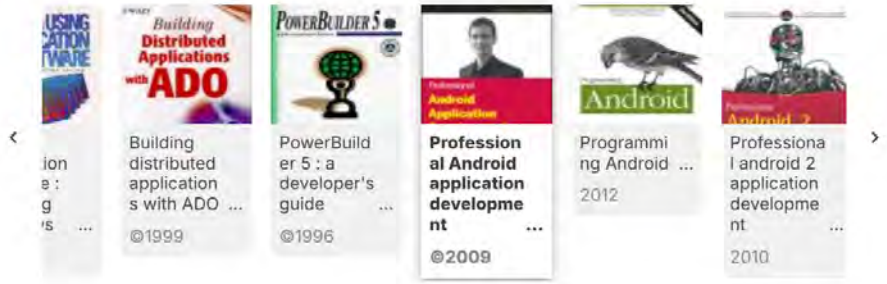
Details

Title	Professional Android application development
Creator	Meier, Reto, 1978-
Subject	Application software -- Development Mobile computing Android (Electronic resource)
Description	"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the concise and compelling examples, which will show you how to quickly construct real-world mobile applications for Android phones. With a series of exercises designed to introduce new concepts, this book will guide you through an introduction to the Android software stack and the philosophy behind creating robust, consistent, and appealing applications for mobile phones so that you can construct increasingly complex mobile phone projects."--BOOK JACKET.
Contents	• Ch. 1. Hello, Android • Ch. 2. Getting Started • Ch. 3. Creating Applications and Activities • Ch. 4. Creating User Interfaces

- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet
- Ch. 6. Data Storage, Retrieval, and Sharing
- Ch. 7. Maps, Geocoding, and Location-Based Services
- Ch. 8. Working in the Background
- Ch. 9. Peer-to-Peer Communication
- Ch. 10. Accessing Android Hardware
- Ch. 11. Advanced Android Development.

Series	Wrox professional guides › Wrox professional guides. ›
Publisher	Indianapolis, IN : Wiley
Creation Date	©2009
Format	xxi, 409 pages : illustrations ; 24 cm.
Identifier	ISBN : 9780470344712 ISBN : 0470344717 OCLC : (OCoLC)226356018 OCLC : (OCoLC)ocn226356018
Source	Library Catalog

Virtual Browse



ion e : g 's ...

Building distributed applications with ADO ... ©1999

PowerBuilder 5 : a developer's guide ... ©1996

Professional Android application development ... ©2009

Programming Android ... 2012

Professional android 2 application development ... 2010

Links

Librarian? View the source record! ›

MARC

Professional Android application development / Reto Meier.

Tag	1st	2nd	Data
LDR			01102cam 22003138a 4500
001			ocn226356018
003			OCoLC
005			20090806115937.0
008			080625s2009 inua 000 0 eng
020			$a9780470344712 (pbk.)
020			$a0470344717 (pbk.)
035			$a(OCoLC)226356018
040			$aUKM$cUKM$dBTCTA$dBAKER$dYDXCP$dBWX$dOCLCQ$dOKU$dOCLCQ$dSZ9XM$dSCT
049			$aSCTA
050		4	$aQA76.76.A65$bM43 2009
100	1		$aMeier, Reto,$d1978-
245	1	0	$aProfessional Android application development /$cReto Meier.
260			$aIndianapolis, IN :$bWiley,$cc2009.
300			$axxi, 409 p. :$bill. ;$c24 cm.
490	1		$aWrox professional guides
500			$a"Wrox programmer to programmer"--Cover.
630	0	0	$aAndroid.
650		0	$aApplication software$xDevelopment.
650		0	$aMobile computing.
830		0	$aWrox professional guides.
907			$a.b11087420$b08-10-09$c08-06-09
998			$abbk$b - - cmda$e-$feng$ginu$h0$i0
994			$aC0$bSCT



EOS.Web® Academic 2025.01 OPAC Discovery, Copyright 2025 Sirsi Corporation. All Rights Reserved.

```
leader  03315cam a2200529 a 4500
001     99147839330001451
005     20250604213024.0
008     080625s2009    inua          000 0 eng
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
019     ##$a1201947138
020     ##$a9780470344712 $q(pbk.)
020     ##$a0470344717 $q(pbk.)
035     ##$a(OCoLC)226356018 $z(OCoLC)1201947138
035     ##$a(OCoLC)ocn226356018
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM $dWAU $dDEBSZ $dOCLCQ $dLND $dOCLCF $dOCLCQ $dS3O $dOCLCQ $dNTG $dUKMGB $dZHC $dOCLCO $dLDP $dOCLCO $dOCLCL $dOCLCQ $dOCLCL
049     ##$aMAIN
050     #4$aQA76.76.A65 $bM43 2009
082     04$a005.1 $222 $1https://id.oclc.org/worldcat/ddc/E3jXFdk4KYFKwJQWKPH9rDFJPB
100     1#$aMeier, Reto, $d1978- $1https://id.oclc.org/worldcat/entity/E39PCjtMKtWB3fgDjJCFfbg7pP
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $c©2009.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides
520     1#$a"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the
500     ##$a"Wrox programmer to programmer"--Cover.
505     0#$aCh. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet -
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
650     #6$aLogiciels d'application $xDéveloppement.
650     #6$aInformatique mobile.
650     #7$aApplication software $xDevelopment. $2fast $0(OCoLC)fst00811707
650     #7$aMobile computing. $2fast $0(OCoLC)fst01024221
650     #7$aAndroid $gSystemplattform $2gnd
650     #7$aGratisprogram. $2sao
650     #7$aOperativsystem. $2sao
650     #7$aTillämpningsprogram. $2sao
630     00$aAndroid (Electronic resource)
630     04$aAndroid.
630     07$aAndroid (Electronic resource) $2fast $0(OCoLC)fst01776064
758     ##$ihas work: $aProfessional Android application development (Text) $1https://id.oclc.org/worldcat/entity/E39PCGgYppd9Pt3ybMHbFyYgKd $4https://id.oclc.org/worldcat/ontology/hasWork
830     #0$aWrox professional guides.
856     4#$qimage/jpeg $uhttp://swbplus.bsz-bw.de/bsz289149010cov.htm $v20090320122929 $3Cover
907     ##$a.b33821938 $b04-03-10 $c01-13-10 $9local
946     ##$aMARS $9local
973     ##$apbooks
994     ##$a92 $bOCACL
998     ##$a(2)s $a(2)ej $i1 $cm $da $e- $feng $ginu $h0 $9local
```



Print Book

Professional Android application development

Meier, Reto, 1978-

©2009

Available at EWU-Cheney Upper Level Books (QA76.76.A65 M454 2009)

Top
Send to
Get It
Details
Links
Virtual Browse

Send to

Export to Excel | Export RIS (Zotero) | Mendeley | EasyBib | EndNote
Export BibTeX | Permalink | Citation | Print | E-mail

Get It

Ask us for help

EWU users sign in to request item Sign in

Back to locations

EWU-Cheney
Available , Upper Level Books ; QA76.76.A65 M454 2009
(1 copy, 1 available, 0 requests)

Item in place
Loanable ; Book

GET IT FROM OTHER SUMMIT LIBRARIES - CLICK TO VIEW INSTITUTIONS

Details

Title	Professional Android application development
Attribution	Reto Meier
Author / Creator	Meier, Reto, 1978-
Summary	**Review:** "Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the concise and compelling examples, which will show you how to quickly construct real-world mobile applications for Android phones. With a series of exercises designed to introduce new concepts, this book will guide you through an introduction to the Android software stack and the philosophy behind creating robust, consistent, and appealing applications for mobile phones so that you can construct increasingly complex mobile phone projects."--BOOK JACKET.
Contents	**Table of contents**: Ch. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet -- Ch. 6. Data Storage, Retrieval, and Sharing -- Ch. 7. Maps, Geocoding, and Location-Based Services -- Ch. 8. Working in the Background -- Ch. 9. Peer-to-Peer Communication -- Ch. 10. Accessing Android Hardware -- Ch. 11. Advanced Android Development.
Subjects	Application software -- Development Mobile computing Android (Electronic resource)
Creation Date	©2009

Related Persons



Meier, Reto, 1978-

Born	1978
Occup...	Computer scientist
Source	Library of Congress Name Authority and Wikidata

Show less

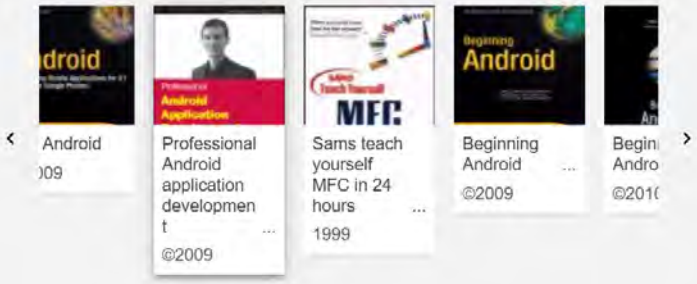
Publisher	Indianapolis, IN : Wiley
Format	xxi, 409 pages : illustrations ; 24 cm.
Language	English
Notes	**Series statement:** Wrox professional guides "Wrox programmer to programmer"--Cover.
Source	Library Catalog
Identifier	ISBN: 9780470344712 (pbk.) ISBN: 0470344717 (pbk.)
OCLC Number	226356018
MMS ID	99338219380101843

Links

Cover

Display source record

Virtual Browse



```
leader  03429cam a2200577 a 4500
001     991083946359705816
005     20250411052603.0
008     080625s2009    inua          000 0 eng
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
020     ##$a9780470344712 $q(pbk.)
020     ##$a0470344717 $q(pbk.)
035     ##$a(OCoLC)226356018 $z(OCoLC)1201947138
035     ##$z(OCoLC)1201947138
035     ##$a(OCoLC)ocn226356018
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM $dWAU $dDEBSZ $dOCLCQ $dLND $dOCLCF $dOCLCQ $dS3O $dOCLCQ $dNTG $dUKMGB $dZHC $dOCLCO $dLDP $dOCLCO $dOCLCL $dOCLCQ
049     ##$aUIUU
050     #4$aQA76.76.A65 $bM43 2009
082     04$a005.1 $222
090     ##$aQA76.76.A65 $bM46 2009 $9LOCAL
100     1#$aMeier, Reto, $d1978- $1https://id.oclc.org/worldcat/entity/E39PCjtMKtWB3fgDjJCFfbg7pP
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $c©2009.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides
500     ##$a"Wrox programmer to programmer"--Cover.
505     0#$aCh. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Int
520     1#$a"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Foll
630     00$aAndroid (Electronic resource)
630     04$aAndroid.
630     07$aAndroid (Electronic resource) $2fast $0(OCoLC)fst01776064
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
650     #6$aLogiciels d'application $xDéveloppement.
650     #6$aInformatique mobile.
650     #7$aApplication software $xDevelopment. $2fast $0(OCoLC)fst00811707
650     #7$aMobile computing. $2fast $0(OCoLC)fst01024221
650     #7$aAndroid $gSystemplattform $2gnd
650     #7$aGratisprogram. $2sao
650     #7$aOperativsystem. $2sao
650     #7$aTillämpningsprogram. $2sao
758     ##$ihas work: $aProfessional Android application development (Text) $1https://id.oclc.org/worldcat/entity/E39PCGgYppd9Pt3ybMHbFyYgKd $4https://id.oclc.org/worldcat/ontology/hasWork
830     #0$aWrox professional guides.
936     ##$aBATCHLOAD
938     ##$aBaker & Taylor $bBKTY $c44.99 $d33.74 $i0470344717 $n0007767820 $sactive
938     ##$aBaker and Taylor $bBTCP $nBK0007767820
938     ##$aYBP Library Services $bYANK $n2827953
945     ##$aWorldCat record encoding level change $b04/11/2025
959     ##$a(ISUdb)1301852 $9LOCAL
994     ##$a92 $bUIU
```



BOOK

Professional Android application development

Meier, Reto, 1978-

©2009

Available at Milner Library Floor 5 Shelves (QA76.76.A65 M46 2009)

TOP
TOOLS
GET IT
DETAILS
VIRTUAL BROWSE

Tools

QR | EXPORT RIS | EXPORT BIBTEX | ENDNOTE | CITATION | PERMALINK | PRINT | EMAIL | EXPORT TO EXCEL

Report a Problem

If you do not have a ULID, please contact us for assistance

Get It

Please sign in to check if there are any request options. Sign in

BACK TO LOCATIONS

LOCATION ITEMS

Milner Library
Available , Floor 5 Shelves ; QA76.76.A65 M46 2009
(1 copy, 1 available, 0 requests)

Item in place
Loanable

GET IT FROM OTHER INSTITUTIONS

University of Illinois at Urbana-Champaign (UIU) —Champaign and Urbana, IL
Owned by institution

Details

Title	Professional Android application development
Creator	Meier, Reto, 1978-
Creation Date	©2009
Publisher	Indianapolis, IN : Wiley
Series	Wrox professional guides Wrox professional guides.
Subject	Android Android (Electronic resource) Application software -- Development Mobile computing Android Systemplattform Gratisprogram Operativsystem Tillämpningsprogram
Contents	Ch. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast Receivers, Adapters, and the Internet -- Ch. 6. Data Storage, Retrieval, and Sharing -- Ch. 7. Maps, Geocoding, and

	Location-Based Services -- Ch. 8. Working in the Background -- Ch. 9. Peer-to-Peer Communication -- Ch. 10. Accessing Android Hardware -- Ch. 11. Advanced Android Development.
Description	"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the concise and compelling examples, which will show you how to quickly construct real-world mobile applications for Android phones. With a series of exercises designed to introduce new concepts, this book will guide you through an introduction to the Android software stack and the philosophy behind creating robust, consistent, and appealing applications for mobile phones so that you can construct increasingly complex mobile phone projects."--BOOK JACKET.
Format	xxi, 409 pages : illustrations ; 24 cm.
Note	"Wrox programmer to programmer"--Cover.
Identifier	ISBN : 9780470344712 ISBN : 0470344717 OCLC : (OCoLC)226356018 OCLC : (OCoLC)ocn226356018
Source	Library Catalog

Milner Library acknowledges that resource and metadata description are not neutral activities. We use standard vocabularies and follow academic library practices of assigning description based on national and international standards for consistent and interoperable searching. However, the catalog does contain terminology that may be biased, outdated, offensive, or harmful. Milner Library is actively involved in updating and changing standards nationally and working locally to be accurate, current, and inclusive in our resource description.

Virtual Browse



Google Workspace ... 2024

Going GAS : from VBA to Google Apps Script ... 2016

Professional Android application development ... ©2009

Murach's Android programming : training & reference ... 2015

Links

No access options above? Submit an interlibrary loan request.

Display Source Record

```
001     991083946359705816
leader  03429cam a2200577 a 4500
005     20250411052603.0
008     080625s2009    inua          000 0 eng
938     ##$aBaker & Taylor $bBKTY $c44.99 $d33.74 $i0470344717 $n0007767820 $sactive
938     ##$aBaker and Taylor $bBTCP $nBK0007767820
938     ##$aYBP Library Services $bYANK $n2827953
945     ##$aLBD data change $b09/02/2022
959     ##$a(UIUdb)5987358 $9LOCAL
994     ##$a92 $bUIU
994     ##$aC0 $bUIU $9LOCAL
050     #4$aQA76.76.A65 $bM43 2009
490     1#$aWrox professional guides
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
082     04$a005.1 $222
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM $dWAU $dDEBSZ $dOCLCQ $dLND $dOCLCF $dOCLCQ $dS3O $dOCLCQ $dNTG $dUKMGB $dZHC $dOCLCO $dLDP $dOCLCO $dOCLCL $dOCLCQ
260     ##$aIndianapolis, IN : $bWiley, $c©2009.
020     ##$a9780470344712 $q(pbk.)
020     ##$a0470344717 $q(pbk.)
100     1#$aMeier, Reto, $d1978- $1https://id.oclc.org/worldcat/entity/E39PCjtMKtWB3fgDjJCFfbg7pP
245     10$aProfessional Android application development / $cReto Meier.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
049     ##$aUIUU
500     ##$a"Wrox programmer to programmer"--Cover.
505     0#$aCh. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Int
520     1#$a"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Foll
630     00$aAndroid (Electronic resource)
630     04$aAndroid.
630     07$aAndroid (Electronic resource) $2fast $0(OCoLC)fst01776064
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
650     #6$aLogiciels d'application $xDéveloppement.
650     #6$aInformatique mobile.
650     #7$aApplication software $xDevelopment. $2fast $0(OCoLC)fst00811707
650     #7$aMobile computing. $2fast $0(OCoLC)fst01024221
650     #7$aAndroid $gSystemplattform $2gnd
650     #7$aGratisprogram. $2sao
650     #7$aOperativsystem. $2sao
650     #7$aTillämpningsprogram. $2sao
758     ##$ihas work: $aProfessional Android application development (Text) $1https://id.oclc.org/worldcat/entity/E39PCGgYppd9Pt3ybMHbFyYgKd $4https://id.oclc.org/worldcat/ontology/hasWork
830     #0$aWrox professional guides.
035     ##$a(OCoLC)226356018 $z(OCoLC)1201947138
035     ##$z(OCoLC)1201947138
035     ##$a(OCoLC)ocn226356018
```



BOOK

Professional Android application development

Meier, Reto, 1978-

Indianapolis, IN : Wiley; ©2009

ISBN 9780470344712; ISBN 0470344717; OCLC (OCoLC)226356018; OCLC (OCoLC)ocn226356018

Located at Grainger Engineering Library Stacks (005.1 M475p) >

TOP
SEND TO
GET IT
DETAILS
LINKS

Send to

EMAIL | PERMALINK | PRINT | QR | EXPORT TO EXCEL | EXPORT BIBTEX | EXPORT RIS (ZOTERO, MENDELEY, ETC.) | EASYBIB | ENDNOTE | CITATION

Get It

Please sign in to request catalog items Sign in
For help signing in click here.

< BACK TO LOCATIONS

LOCATION ITEMS

Grainger Engineering Library
On Shelf , Stacks ; 005.1 M475p
(1 copy, 1 available, 0 requests)

Item in place Copy: 1
Loanable

Report a problem with this item

Click here for availability via I-Share >

Details

Title	Professional Android application development
Attribution	Reto Meier
Creator	Meier, Reto, 1978- >
Language	English
Subject	Android >
	Android (Electronic resource) >
	Logiciels d'application -- Développement >
	Informatique mobile >
	Application software -- Development >
	Mobile computing >
	Android Systemplattform >
	Gratisprogram >
	Operativsystem >
	Tillämpningsprogram >
Description	"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the concise and compelling examples, which will show you how to quickly construct real-world mobile applications for Android phones. With a series of exercises designed to introduce new concepts, this book will guide you through an introduction to the Android software stack and the philosophy behind creating robust, consistent, and appealing applications for mobile phones so that you can

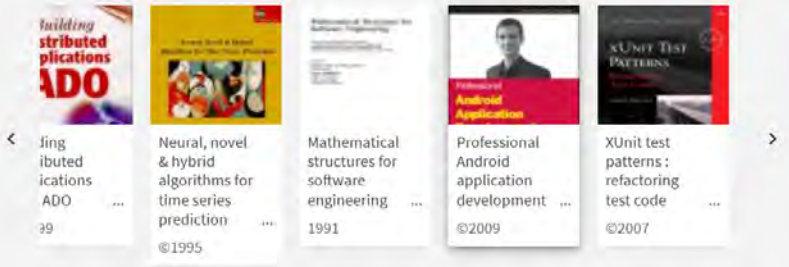
Ask a Librarian

	construct increasingly complex mobile phone projects."--BOOK JACKET.
Contents	Ch. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet -- Ch. 6. Data Storage, Retrieval, and Sharing -- Ch. 7. Maps, Geocoding, and Location-Based Services -- Ch. 8. Working in the Background -- Ch. 9. Peer-to-Peer Communication -- Ch. 10. Accessing Android Hardware -- Ch. 11. Advanced Android Development.
Series	Wrox professional guides › Wrox professional guides. ›
Publisher	Indianapolis, IN : Wiley
Place of Publication	Indianapolis, IN :
Creation Date	©2009
Identifier	ISBN : 9780470344712 ISBN : 0470344717 OCLC : (OCoLC)226356018 OCLC : (OCoLC)ocn226356018
Format	xxi, 409 pages : illustrations ; 24 cm.
Source	Library Catalog
General Notes	"Wrox programmer to programmer"--Cover.

Links

Display Source Record ›

Virtual Browse



ding ibuted ications ADO ... 99

Neural, novel & hybrid algorithms for time series prediction ... ©1995

Mathematical structures for software engineering ... 1991

Professional Android application development ... ©2009

XUnit test patterns : refactoring test code ... ©2007

```
leader  03165cam a2200505 a 4500
001     991005543919706121
005     20250604213024.0
008     080625s2009    inua          000 0 eng
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
019     ##$a1201947138
020     ##$a9780470344712 $q(pbk.)
020     ##$a0470344717 $q(pbk.)
035     ##$aocn226356018-01cuny_network
035     ##$a(OCoLC)226356018 $z(OCoLC)1201947138
035     ##$a(OCoLC)ocn226356018
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM $dWAU $dDEBSZ $dOCLCQ $dLND $dOCLCF $dOCLCQ $dS3O $dOCLCQ $dNTG $dUKMGB $dZHC $dOCLCO $dLDP $dOCLCO $dOCLCL $dOCLCQ $dOCLCL
049     ##$aMAIN
050     #4$aQA76.76.A65 $bM43 2009
082     04$a005.1 $222 $1https://id.oclc.org/worldcat/ddc/E3jXFdk4KYFKwJQWKPH9rDFJPB
100     1#$aMeier, Reto, $d1978- $1https://id.oclc.org/worldcat/entity/E39PCjtMKtWB3fgDjJCFfbg7pP
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $c02009.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides
500     ##$a"Wrox programmer to programmer"--Cover.
505     0#$aCh. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet --
520     1#$a"Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the c
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
650     #7$aApplication software $xDevelopment. $2fast $0(OCoLC)fst00811707
650     #7$aMobile computing. $2fast $0(OCoLC)fst01024221
650     #7$aAndroid $gSystemplattform $2gnd
650     #7$aGratisprogram. $2sao
650     #7$aOperativsystem. $2sao
650     #7$aTillämpningsprogram. $2sao
630     00$aAndroid (Electronic resource)
630     07$aAndroid (Electronic resource) $2fast $0(OCoLC)fst01776064
758     ##$ihas work: $aProfessional Android application development (Text) $1https://id.oclc.org/worldcat/entity/E39PCGgYppd9Pt3ybMHbFyVgkd $4https://id.oclc.org/worldcat/ontology/hasWork
830     #0$aWrox professional guides.
901     ##$a006082470 $a006391479
999     ##$aOCLC Daily new or updates
```



BOOK

Professional Android application development

Meier, Reto, 1978-

©2009

May be available; see options

Send to

EMAIL CITATION PRINT PERMALINK REFWORKS ENDNOTE EASYBIB EXPORT RIS EXPORT BIBTEX

How to get it

Sign in to place requests Sign in

OTHER CUNY LIBRARIES

New York City College of Technology

Owned by this institution

Details

Title Professional Android application development

Creator Meier, Reto, 1978-

Subject Application software -- Development
Mobile computing
Android (Electronic resource)

Description "Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the concise and compelling examples, which will show you how to quickly construct real-world mobile applications for Android phones. With a series of exercises designed to introduce new concepts, this book will guide you through an introduction to the Android software stack and the philosophy behind creating robust, consistent, and appealing applications for mobile phones so that you can construct increasingly complex mobile phone projects."--BOOK JACKET.

Contents
- Ch. 1. Hello, Android
- Ch. 2. Getting Started
- Ch. 3. Creating Applications and Activities
- Ch. 4. Creating User Interfaces
- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet
- Ch. 6. Data Storage, Retrieval, and Sharing
- Ch. 7. Maps, Geocoding, and Location-Based Services
- Ch. 8. Working in the Background
- Ch. 9. Peer-to-Peer Communication
- Ch. 10. Accessing Android Hardware
- Ch. 11. Advanced Android Development.

Series Wrox professional guides

	Wrox professional guides. ›
Publisher	Indianapolis, IN : Wiley
Creation Date	©2009
Format	xxi, 409 pages : illustrations ; 24 cm.
Identifier	ISBN : 9780470344712 ISBN : 0470344717 OCLC : (OCoLC)226356018 OCLC : (OCoLC)ocn226356018
Source	Library Catalog

Links

Check full text availability at Open Library ›

Note: You will need to create a free Internet Archive account to access available full text at Open Library

Something broken? Help us fix it! ›

If you retrieve the permalink for this record (above, in "Send to" section), it will help us troubleshoot this problem for you!

Librarian? View the source record! ›

MARC for NCSU2194927

```
000   01647cam a22004578a 4500
001   ocn226356018
003   OCoLC
005   20081230000000.0
008   080625t20092009inu 000 0 eng
015   |aGBA885526|2bnb
0167  |a014662137|2Uk
020   |a9780470344712|qpaperback
020   |a0470344717|qpaperback
035   |a(OCoLC)226356018
040   |aUKM|beng|cUKM|dBTCTA|dBAKER|dYDXCP|dBWX|dOCLCQ
049   |aNRCC|xamb
050  4|aQA76.76.A65|bM43 2009
08204|a005.1|222
1001  |aMeier, Reto,|d1978-|0http://id.loc.gov/authorities/names/no2009029358
24510|aProfessional Android application development /|cReto Meier.
264  1|aIndianapolis, IN :|bWiley,|c[2009]
264  4|c©2009
300   |axxi, 409 pages :|billustrations ;|c24 cm.
336   |atext|btxt|2rdacontent
337   |aunmediated|bn|2rdamedia
338   |avolume|bnc|2rdacarrier
```

```
4901 |aWrox professional guides
500  |a"Wrox programmer to programmer"--Cover.
596  |a14
63000|aAndroid (Electronic resource)|0http://id.loc.gov/auth
     orities/names/n2009043077
650 0|aApplication software|xDevelopment.|0http://id.loc.go
     v/authorities/subjects/sh95009362
650 0|aMobile computing.|0http://id.loc.gov/authorities/subj
     ects/sh95004596
830 0|aWrox professional guides.|0http://id.loc.gov/authorit
     ies/names/no99020592
909  |a20091113
918  |a2194927
949  |hHIL-STACKS|i2827953|kON-ORDER|zPROMPTCAT
980  |a2008-12-29|b$44.99|e$40.94|f593388|g1
983  |a40016193788
994  |a90|bNRC
917  |au2194927
985  |aHSFIR-APRV|eYANKEE
```

Close

Professional Android application development

Reto Meier.

Meier, Reto, 1978-

Indianapolis, IN : Wiley, [2009] / Book / Print



Where to find it

Hunt Library — bookBot

Call Number QA76.76 .A65 M43 2009
(Browse Shelf)

Format Book
Status Available upon request
Request item

Authors, etc.

- Meier, Reto, 1978-

Summary

A hands-on guide to building mobile applications, Professional Android Application Development features concise and compelling examples that show you how to quickly construct real-world mobile applications for Android phones. Fully up-to-date for version 1.0 of the Android software development kit, it covers all the essential features, and explores the advanced capabilities of Android (including GPS, accelerometers, and background Services) to help you

show more

Contents

Introduction

Chapter 1 Hello, Android

show more

Find similar items

Browse shelf

By Topic

- Android (Electronic resource)
- Application software > Development
- Mobile computing

Other details

Series:
Wrox professional guides

Published:
Indianapolis, IN : Wiley, [2009]
©2009

Language:
English

Notes:

Description:
xxi, 409 pages : illustrations ; 24 cm.

ISBN:
9780470344712
0470344717

OCLC Number:
226356018

Other Identifiers:

"Wrox programmer to programmer"--Cover.

British national bibliography: GBA885526

Related items

Series:

Wrox professional guides.

View MARC

Contact

D. H. Hill Jr. Library
2 Broughton Drive
Campus Box 7111
Raleigh, NC 27695-7111
(919) 515-3364

James B. Hunt Jr. Library
1070 Partners Way
Campus Box 7132
Raleigh, NC 27606-7132
(919) 515-7110

Libraries Administration
(919) 515-7188

NC State University Libraries

D. H. Hill Jr. Library
James B. Hunt Jr. Library
Design Library
Natural Resources Library
Veterinary Medicine Library

Accessibility at the Libraries
Accessibility at NC State University
Terms of Use
Jobs
Privacy Statement
Staff Confluence Login
Staff Drupal Login

Follow the Libraries

Facebook
Instagram
Threads
Bluesky
LinkedIn
Vimeo
YouTube
YouTube Archive
Flickr
Libraries' news

CHAT NOW

Libraries & Media Center

Start Over | Regular Display | Another Search | (Search History)

```
LEADER 00000cam a2200000Ia 4500
001    226356018
003    OCoLC
005    20090819105637.0
008    080625s2009    inua           000 0 eng
015    GBA885526|2bnb
016 7  014662137|2Uk
020    9780470344712 (pbk.)
020    0470344717 (pbk.)
035    (OCoLC)226356018
040    UKM|cUKM|dBTCTA|dBAKER|dYDXCP|dBWX|dOCLCQ|dOKU|dOCLCQ
       |dSZ9XM
049    TSUU
050  4 QA76.76.A65|bM43 2009
050  4 QA76.76.A65|bM43 2009
082 04 005.1|222
100 1  Meier, Reto,|d1978-
245 10 Professional Android application development /|cReto
       Meier.
260    Indianapolis, IN :|bWiley,|cc2009.
300    xxi, 409 p. :|bill. ;|c24 cm.
490 1  Wrox professional guides
500    "Wrox programmer to programmer"--Cover.
630 00 Android.
650  0 Application software|xDevelopment.
650  0 Mobile computing.
830  0 Wrox professional guides.
994    C0|bTSU
```

Location	Call No.	Status	OPAC Message
Shelf 3rd Flr - Main	QA76.76.A65 M43 2009	AVAILABLE	---

Libraries & Media Center

Start Over | Request | Save Records | MARC Display | More Information | Another Search

OCLC NO 226356018 View Entire Collection System Sorted Search

Limit to netLibrary eBooks

Limit search to available items

(Search History) Result page: Previous Next

Author Meier, Reto, 1978-

Title **Professional Android application development / Reto Meier.**

Publication Info. Indianapolis, IN : Wiley, c2009.



Bookmark link for this record

Location	Call No.	Status	OPAC Message
Shelf 3rd Flr - Main	QA76.76.A65 M43 2009	AVAILABLE	---

Details

Description xxi, 409 p. : ill. ; 24 cm.

Series Wrox professional guides
Wrox professional guides.

Note "Wrox programmer to programmer"--Cover.

Subject Android.
Application software -- Development.
Mobile computing.

ISBN 9780470344712 (pbk.)
0470344717 (pbk.)

Google Preview

Start Over | Request | Save Records | MARC Display | More Information | Another Search

Classic Catalog
Basic Search
Advanced Search
Course Reserves
Course Search
Instructor Search
My Library
Login
Renew Books
Suggest A Purchase
Make A Suggestion
Help



Professional Android application development /

Main Author:	Meier, Reto, 1978-
Format:	Book
Language:	English
Published:	Indianapolis, IN : Wiley, [2009]
Series:	Wrox professional guides.
Subjects:	Android. Application software > Development. Mobile computing.

MARC

LEADER	00000cam a22000008a 4500		
001	in00002490456		
005	20151110043033.0		
008	080625s2009 inu 000 0 eng		
015			**|a** GBA885526 **|2** bnb
016	7		**|a** 014662137 **|2** Uk
020			**|a** 9780470344712 (pbk.)
020			**|a** 0470344717 (pbk.)
035			**|a** (OCoLC)ocn226356018
035			**|a** (OCoLC)226356018
040			**|a** UKM **|c** UKM **|d** BTCTA **|d** BAKER **|d** YDXCP **|d** BWX **|d** OCLCQ **|d** UtOrBLW
049			**|a** TXAM
050		4	**|a** QA76.76.A65 **|b** M43 2009
082	0	4	**|a** 005.1 **|2** 22
100	1		**|a** Meier, Reto, **|d** 1978-
245	1	0	**|a** Professional Android application development / **|c** Reto Meier.
264		1	**|a** Indianapolis, IN : **|b** Wiley, **|c** [2009]
264		4	**|c** ©2009
300			**|a** xxi, 409 pages : **|b** illustrations ; **|c** 24 cm.

336			|a text |b txt |2 rdacontent
337			|a unmediated |b n |2 rdamedia
338			|a volume |b nc |2 rdacarrier
490	1		|a Wrox professional guides
500			|a "Wrox programmer to programmer"--Cover.
630	0	0	|a Android.
650		0	|a Application software |x Development.
650		0	|a Mobile computing.
830		0	|a Wrox professional guides.
938			|a Baker and Taylor |b BTCP |n BK0007767820
938			|a Baker & Taylor |b BKTY |c 44.99 |d 33.74 |i 0470344717 |n 0007767820 |s active
938			|a YBP Library Services |b YANK |n 2827953
938			|a Blackwell Book Service |b BBUS |n R2748888 |c $44.99
945			|a PromptCat |b 607982
946			|a stk
947			|a A14837436271
994			|a 92 |b TXA
948			|a cataloged |b h |d 2009/1/28 |h c |e asalazar |f 1:37:17 pm
999			|a MARS
999	f	f	|s 9f4a958e-c3e5-3991-b334-38ebe98e461f |i aae266d3-103b-3c7b-a9b4-a39183611db5 |t 0
952	f	f	|p normal |a Texas A&M University |b College Station |c Sterling C. Evans Library |d Evans: Library Stacks |t 0 |e QA76.76.A65 M43 2009 |h Library of Congress classification |i unmediated -- volume |m A14837436271
998	f	f	|a QA76.76.A65 M43 2009 |t 0 |l Evans: Library Stacks

UNIVERSITY OF HOUSTON
Contact Us My Accounts Give to UH Libraries
University of Houston Libraries

```
leader  01742cam a22004098i 4500
001     991015023069705701
005     20240917121709.0
008     080625s2009    inu           001 0 eng
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
020     ##$a9780470344712 $q(paperback)
020     ##$a0470344717 $q(paperback)
035     ##$a(TxHU)b40516441-01uho_inst
035     ##$a(OCoLC)226356018
040     ##$aUKM $beng $erda $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ
049     ##$aTXHU
050     #4$aQA76.76.A65 $bM43 2009
100     1#$aMeier, Reto, $d1978- $0https://id.loc.gov/authorities/names/no2009029358
245     10$aProfessional Android application development / $cReto Meier.
264     #4$aIndianapolis, IN : $bWiley, $c©2009.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides.
500     ##$a"Wrox programmer to programmer"--Cover.
500     ##$aIncludes index.
650     #0$aAndroids. $0https://id.loc.gov/authorities/subjects/sh85004921
650     #7$aAndroids. $2fast $0https://id.worldcat.org/fast/1198480
650     #0$aApplication software $xDevelopment. $0https://id.loc.gov/authorities/subjects/sh95009362
650     #7$aApplication software $xDevelopment. $2fast $0https://id.worldcat.org/fast/811707
650     #0$aMobile computing. $0https://id.loc.gov/authorities/subjects/sh95004596
650     #7$aMobile computing. $2fast $0https://id.worldcat.org/fast/1024221
830     #0$aWrox professional guides $0https://id.loc.gov/authorities/names/no99020592
907     ##$a.b40516441
910     ##$aMARCIVE 2024-08-12
998     ##$aan $b01-30-09 $c- $da $e- $feng $ginu $h0 $i2 $j06-09-13 $k12-01-08
```



BOOK

Professional Android application development

Meier, Reto, 1978-

©2009

Available at UH/MD Anderson Library Anderson General Collection (QA76.76.A65 M43 2009) >

TOP
SEND TO
GET IT
DETAILS
VIRTUAL BROWSE
LINKS

Send to

EXPORT BIBTEX | EXPORT RIS | EASYBIB | ENDNOTE ONLINE | REFWORKS | PRINT | CITATION | PERMALINK | EMAIL

Get It

Please sign in to check if there are any request options. Sign in

< BACK TO LOCATIONS

LOCATION ITEMS

UH/MD Anderson Library
Available , Anderson General Collection ; QA76.76.A65 M43 2009
(1 copy, 1 available, 0 requests)

Item in place
Loan Period: Please sign in to view loan policy.

Details

Title	Professional Android application development
Creator	Meier, Reto, 1978- >
Subject	Androids >
	Application software -- Development >
	Mobile computing >
Related titles	Available in other form: Professional Android application development >
	Available in other form: Professional Android application development >
Series	Wrox professional guides >
	Wrox professional guides. >
Publisher	Indianapolis, IN : Wiley
Creation Date	©2009
Format	xxi, 409 pages : illustrations ; 24 cm.
Source	Library Catalog
Identifier	ISBN : 9780470344712

ISBN : 0470344717
OCLC : (OCoLC)226356018

General Notes "Wrox programmer to programmer"--Cover.
Includes index.

Language English

Virtual Browse



ition
tion :
B,
id SOA ...

Enterprise modeling with UML : designing successful ...
©2000

Interdependencies in the discovery and adoption of Facebook ...
2013

Professional Android application development ...
©2009

Links

Display Source Record >

```
leader  01525cam a22004338a 4500
001     991023640459706011
005     20081219000000.0
008     080625s2009    inu           000 0 eng
015     ##$aGBA885526 $2bnb
016     7#$a014662137 $2Uk
020     ##$a9780470344712 (pbk.)
020     ##$a0470344717 (pbk.)
035     ##$a(IXA)b71113575-01utau_inst
035     ##$a(OCoLC)226356018
040     ##$aUKM $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ
049     ##$aIXAA
050     #4$aQA76.76.A65 $bM43 2009
082     04$a005.1 $222
100     1#$aMeier, Reto, $d1978-
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $cc2009.
300     ##$axxi, 409 p. : $bill. ; $c24 cm.
490     1#$aWrox professional guides
500     ##$a"Wrox programmer to programmer"--Cover.
630     00$aAndroid.
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
830     #0$aWrox professional guides
907     ##$a.b71113575 $b01-17-12 $c11-25-08
998     ##$apcg $b02-16-09 $cm $da   $e- $feng $ginu $h0 $i1
915     ##$a226356018
938     ##$aBaker and Taylor $bBTCP $nBK0007767820
938     ##$aBaker & Taylor $bBKTY $c44.99 $d33.74 $i0470344717 $n0007767820 $sactive
938     ##$aYBP Library Services $bYANK $n2827953
938     ##$aBlackwell Book Service $bBBUS $nR2748888 $c$44.99
945     ##$aPromptCat
945     ##$aPromptCat
994     ##$a92 $bIXA
945     ##$aQA 76.76 A65285 M43 2009 $e01-20-2015 21:28 $g1 $i059173025190024 $j0 $lpcg   $o- $p$0.00 $q- $r- $s-   $t10 $u15 $v17 $w0 $x0 $y.i182554879 $z02-02-09
```



BOOK

Professional Android application development / Reto Meier.

Meier, Reto, 1978-
c2009
Wrox professional guides

Available at Perry-Castañeda Library PCL Stacks (QA 76.76 A65285 M43 2009)

TOP
SEND TO
GET IT
LINKS
DETAILS
VIRTUAL BROWSE

Send to

EXPORT BIBTEX EXPORT RIS QR EXPORT TO EXCEL ENDNOTE PRINT
CITATION LINK TO RECORD EMAIL

Get It

Please sign in to check if there are any request options. Sign in

BACK TO LOCATIONS

LOCATION ITEMS

Perry-Castañeda Library
Available , PCL Stacks ; QA 76.76 A65285 M43 2009
(1 copy, 1 available, 0 requests)

LOCATE

Item in place
Loanable

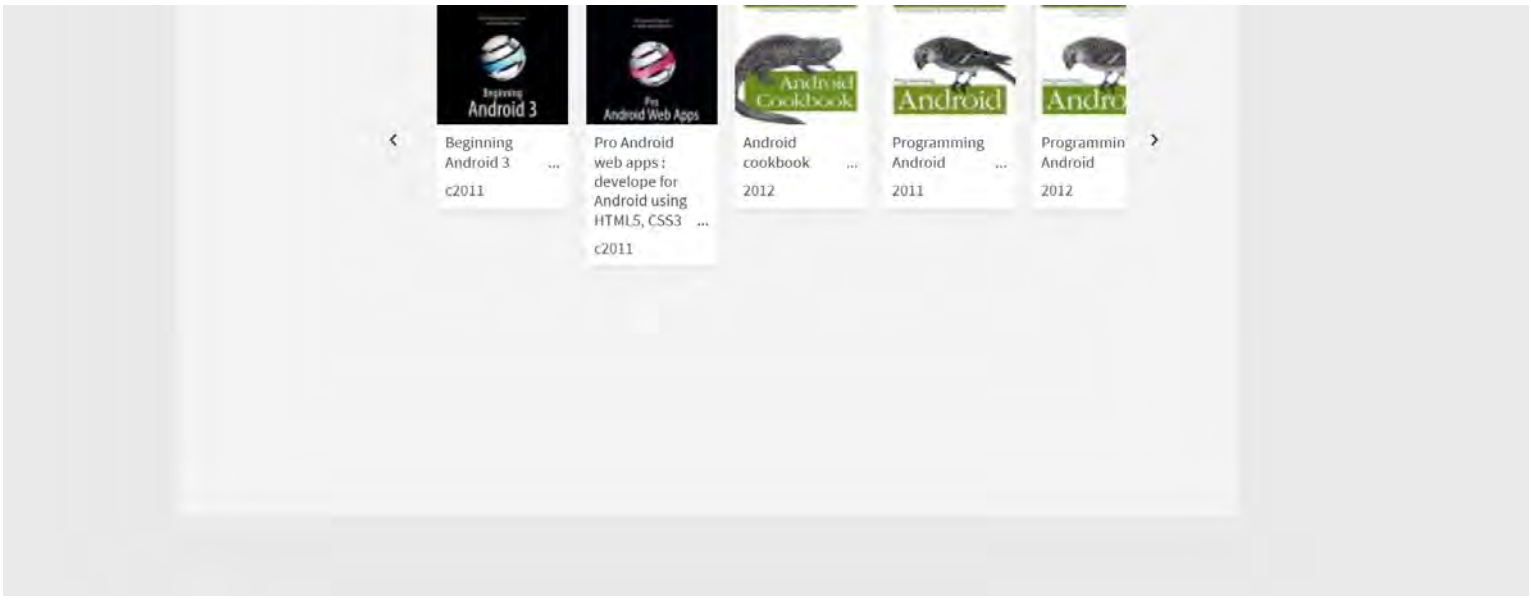
Links

Display Source Record

Details

Title	Professional Android application development / Reto Meier.
Related titles	Available in other form: Professional Android application development
Creator	Meier, Reto, 1978-
Publisher	Indianapolis, IN : Wiley, c2009.
Format	xxi, 409 p. : ill. ; 24 cm.
Series	Wrox professional guides
Notes	"Wrox programmer to programmer"--Cover.
Subject	Android Application software -- Development Mobile computing
Type	Book
Creation Date	c2009
Identifier	ISBN : 9780470344712 (pbk.) ISBN : 0470344717 (pbk.) OCLC : (OCoLC)226356018
Source	Library Catalog

Virtual Browse


Beginning Android 3
Beginning Android 3 ...
c2011
Pro Android Web Apps
Pro Android web apps : develope for Android using HTML5, CSS3 ...
c2011
Android Cookbook
Android cookbook ...
2012
Android
Programming Android ...
2011
Andro
Programmin Android
2012

```
leader       cam  22003858a 4500
001     9912391131006561
005     20100223152553.0
008     080625s2009    inua          000 0 eng
020     ##$a9780470344712 (pbk.)
020     ##$a0470344717 (pbk.)
035     ##$a(aleph)025415856UXU01
035     ##$a(OCoLC)226356018
040     ##$aUKM $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM
090     ##$aQA76.76.A65 $bM43 2009
100     1#$aMeier, Reto, $d1978-
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $cc2009.
300     ##$axxi, 409 p. : $bill. ; $c24 cm.
440     #0$aWrox professional guides
500     ##$a"Wrox programmer to programmer"--Cover. $5FPeU
630     00$aAndroid.
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
998     ##$a025415856 $9LOCAL
```

Ask Us!



BOOK

Professional Android application development

Meier, Reto, 1978-

c2009

Available at UWF John C. Pace Library (Bldg. 32) JCPL General Collection (3rd/4th Floors) (QA76.76.A65 M43 2009) >

TOP
SEND TO
FIND IT
DETAILS
VIRTUAL BROWSE

Send to

EMAIL | CITATION | PERMALINK | PRINT | EXPORT RIS | ENDNOTE

Find It

REQUEST: Request Chapter/Section Scan

Please sign in to check if there are any request options. Sign in

BACK TO LOCATIONS

LOCATION ITEMS

UWF John C. Pace Library (Bldg. 32)
Available , JCPL General Collection (3rd/4th Floors) ; QA76.76.A65 M43 2009 (1 copy, 1 available, 0 requests) LOCATE

Item in place
May be loanable

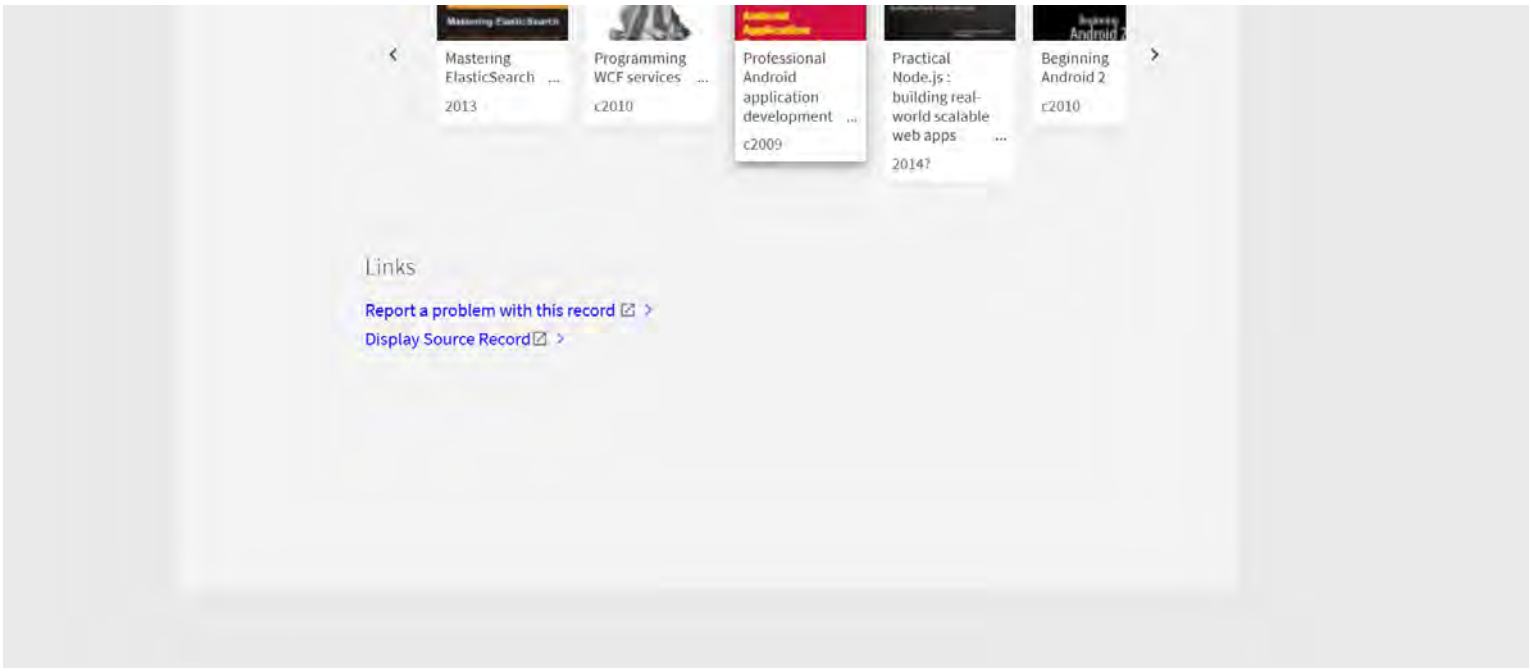
OTHER INSTITUTIONS THAT OWN THIS TITLE >

Details

Creator	Meier, Reto, 1978- >
Title	Professional Android application development
Series	Wrox professional guides >
Publisher	Indianapolis, IN : Wiley
Creation Date	c2009
Format	xxi, 409 p. : ill. ; 24 cm.
Notes	"Wrox programmer to programmer"--Cover.
Identifier	ISBN : 9780470344712 (pbk.) ISBN : 0470344717 (pbk.) OCLC : (OCoLC)226356018
MMS ID	9937958121190660
Subject	Android > Application software -- Development > Mobile computing >
Language	English
Type	Book
Source	Library Catalog

Virtual Browse




Mastering ElasticSearch ...
2013
Programming WCF services ...
c2010
Professional Android application development ...
c2009
Practical Node.js : building real-world scalable web apps ...
2014?
Beginning Android 2
c2010
Links
Report a problem with this record
Display Source Record

```
leader  03434cam a2200565 a 4500
001     99117564690001101
005     20250411052603.0
008     080625s2009    inua          000 0 eng
015     ##$aGBA885526 $2bnb
019     ##$a1201947138
020     ##$a9780470344712 $q(pbk.)
020     ##$a0470344717 $q(pbk.)
035     ##$a(OCoLC)226356018 $z(OCoLC)1201947138
035     ##$a(OCoLC)ocn226356018
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM $dWAU $dDEBSZ $dOCLCQ $dLND $dOCLCF $dOCLCQ $dS3O $dOCLCQ $dNTG $dUKMGB $dZHC $dOCLCO $dLDP $dOCLCO $dOCLCL $dOCLCQ
049     ##$aVRCC
050     #4$aQA76.76.A65 $bM43 2009
082     04$a005.1 $222
100     1#$aMeier, Reto, $d1978- $1https://id.oclc.org/worldcat/entity/E39PCjtMKtWB3fgDjJCFfbg7pP
245     10$aProfessional Android application development / $cReto Meier.
260     ##$aIndianapolis, IN : $bWiley, $c©2009.
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides
583     1#$acommitted to retain $c20230901 $d20310630 $fEAST $zCommitted to retain for the EAST shared print initiative. For more information see $uhttp://eastlibraries.org/retained-materials $5VRC
505     0#$aCh. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet
520     1#$a"Build mobile applications using Android SDK with the hands-on information in Professional Android Applic                                                                    d platform features, techniques and practices. Follow the
500     ##$a"Wrox programmer to programmer"--Cover.
630     00$aAndroid (Electronic resource)
630     07$aAndroid (Electronic resource) $2fast $0(OCoLC)fst01776064
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
650     #7$aApplication software $xDevelopment. $2fast $0(OCoLC)fst00811707
650     #7$aMobile computing. $2fast $0(OCoLC)fst01024221
758     ##$ihas work: $aProfessional Android application development (Text) $1https://id.oclc.org/worldcat/entity/E39PCGgYppd9Pt3ybMHbFyYgKd $4https://id.oclc.org/worldcat/ontology/hasWork
830     #0$aWrox professional guides
946     ##$aWorldshare MARC Update
948     ##$a200906
994     ##$a92 $bVRC
```

Chat with a librarian
Need help? Chat with us now.
Click to chat No thanks



BOOK

Professional Android application development

Reto Meier 1978-

Indianapolis, IN : Wiley, 2009.

Cabell Library 4th Floor

Available at Cabell Library General Collection (QA76.76.A65 M43 2009) >

TOP

REQUEST IT (PH...

SEND TO

DETAILS

LINKS

Chat with a librarian

Need help? Chat with us now.

Click to chat No thanks

Request It *(Physical items from VCU Libraries and remote storage)*

Requesting materials and more access options may be available. Sign in for the best experience.

BACK TO LOCATIONS

LOCATION ITEMS

Cabell Library

Available , General Collection ; QA76.76.A65 M43 2009

(1 copy, 1 available, 0 requests)

Item in place

Loanable

Send to

REPORT A PROBLEM | EXPORT TO EXCEL | EXPORT BIBTEX | EXPORT RIS | ENDNOTE | EASYBIB

CITATION | PERMALINK | PRINT | EMAIL

Details

Title	Professional Android application development
Creator	Reto Meier 1978- >
Subject	Android (Electronic resource) > Application software -- Development > Mobile computing >
Description	Summary/Abstract: "Build mobile applications using Android SDK with the hands-on information in Professional Android Application Development, a guide to Android platform features, techniques and practices. Follow the concise and compelling examples, which will show you how to quickly construct real-world mobile applications for Android phones. With a series of exercises designed to introduce new concepts, this book will guide you through an introduction to the Android software stack and the philosophy behind creating robust, consistent, and appealing applications for mobile phones so that you can construct increasingly complex mobile phone projects."--BOOK JACKET. Note: "Wrox programmer to programmer"--Cover.
Contents	Ch. 1. Hello, Android -- Ch. 2. Getting Started -- Ch. 3. Creating Applications and Activities -- Ch. 4. Creating User Interfaces -- Ch. 5. Intents, Broadcast R eceivers, Adapters, and the Internet -- Ch. 6. Data Storage, Retrieval, and Sharing -- Ch. 7. Maps, Geocoding, and Location-Based Services -- Ch. 8. Working in the Background -- Ch. 9. Peer-to-Peer Communication -- Ch. 10. Accessing Android Hardware -- Ch. 11. Advanced Android Development.
Related titles	Series : Wrox professional guides

Publisher	Indianapolis, IN : Wiley
Creation Date	2009.
Format	xxi, 409 pages : illustrations ; 24 cm.
Language	English
Identifier	ISBN : 9780470344712 ISBN : 0470344717 OCLC : (OCoLC)226356018 OCLC : (OCoLC)ocn226356018
Source	Library Catalog

Links

View Title in WorldCat >

Display Source Record >

Virtual Browse



al Studio
:book :
50 new
improved ...



Hacking Kubernetes : threat-driven analysis and defense ...
2021



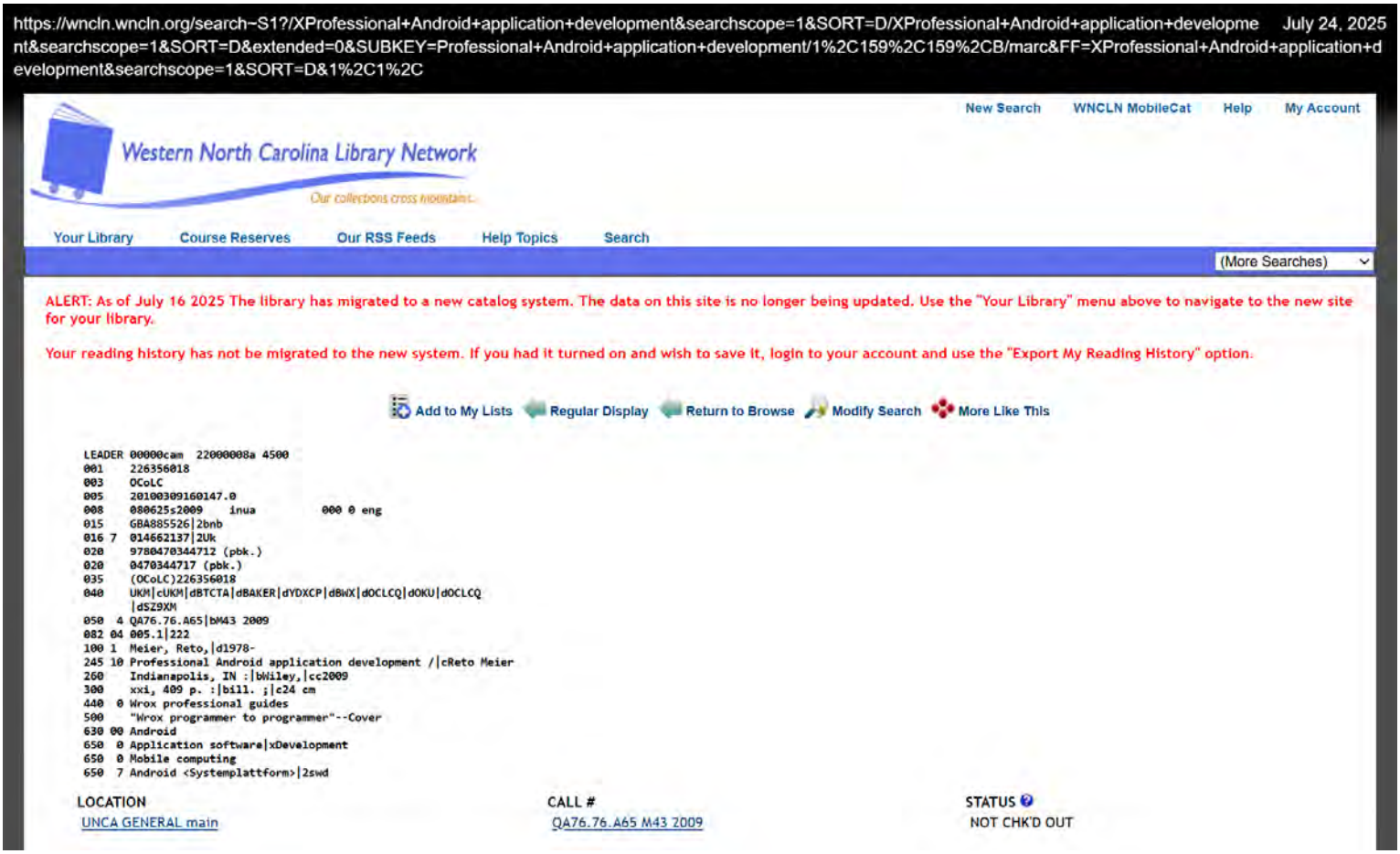
Laravel application development cookbook ...
2013



Professional Android application development ...
©2009



Programming Android ...
2012


New Search
WNCLN MobileCat
Help
My Account
Western North Carolina Library Network
Our collections cross mountains.
Your Library
Course Reserves
Our RSS Feeds
Help Topics
Search
(More Searches)
ALERT: As of July 16 2025 The library has migrated to a new catalog system. The data on this site is no longer being updated. Use the "Your Library" menu above to navigate to the new site for your library.
Your reading history has not be migrated to the new system. If you had it turned on and wish to save it, login to your account and use the "Export My Reading History" option.
Add to My Lists
Regular Display
Return to Browse
Modify Search
More Like This
LEADER 00000cam 22000008a 4500
001 226356018
003 OCoLC
005 20100309160147.0
008 080625s2009 inua 000 0 eng
015 GBA885526|2bnb
016 7 014662137|2Uk
020 9780470344712 (pbk.)
020 0470344717 (pbk.)
035 (OCoLC)226356018
040 UKM|cUKM|dBTCTA|dBAKER|dYDXCP|dBWX|dOCLCQ|dOKU|dOCLCQ |dSZ9XM
050 4 QA76.76.A65|bM43 2009
082 04 005.1|222
100 1 Meier, Reto,|d1978-
245 10 Professional Android application development /|cReto Meier
260 Indianapolis, IN :|bWiley,|cc2009
300 xxi, 409 p. :|bill. ;|c24 cm
440 0 Wrox professional guides
500 "Wrox programmer to programmer"--Cover
630 00 Android
650 0 Application software|xDevelopment
650 0 Mobile computing
650 7 Android <Systemplattform>|2swd
LOCATION
UNCA GENERAL main
CALL #
QA76.76.A65 M43 2009
STATUS
NOT CHK'D OUT

New Search WNCLN MobileCat Help My Account

Western North Carolina Library Network
Our collections cross mountains.

Your Library Course Reserves Our RSS Feeds Help Topics Search

(More Searches)

ALERT: As of July 16 2025 The library has migrated to a new catalog system. The data on this site is no longer being updated. Use the "Your Library" menu above to navigate to the new site for your library.

Your reading history has not be migrated to the new system. If you had it turned on and wish to save it, login to your account and use the "Export My Reading History" option.

Request Add to My Lists Add to Bag MARC Display Return to Browse Modify Search More Like This

KEYWORD Professional Android application develc WNCLN (ASU, UNCA, & WCU) Search

Limit search to available items

159 results found. Sorted by relevance | date | title .

Next Record



Author Meier, Reto, 1978-
Title Professional Android application development / Reto Meier
Publisher Indianapolis, IN : Wiley, c2009

Permanent Link Cite this title Report a problem

LOCATION	CALL #	STATUS
UNCA GENERAL main	QA76.76.A65 M43 2009	NOT CHK'D OUT

Book

Description xxi, 409 p. : ill. ; 24 cm
Series Wrox professional guides
NOTE "Wrox programmer to programmer"--Cover
SUBJECT Android
Topical Subject Application software -- Development
Mobile computing
Subject Android <Systemplattform>
OCLC # 226356018

More From Syndetics Unbound

Summary About The Author Look Inside You May Also Like
Reader Reviews Tags Browse Shelf

Summary

A hands-on guide to building mobile applications, *Professional Android Application Development* features concise and compelling examples that show you how to quickly construct real-world mobile applications for Android phones. Fully up-to-date for version 1.0 of the Android software development kit, it covers all the essential features, and explores the advanced capabilities of Android (read more)

About The Author



Reto Meier

Originally from Perth, Western Australia, **Reto Meier** now lives in London.
Reto is an experienced software developer with more than 10 years of experience in GUI application architecture, design, and development. He's worked in various industries, including offshore oil and gas, before moving to London and into fi nance. Always interested in emerging technologies, Reto has been involved (read more)

More by Reto Meier



Look Inside

Introduction

Chapter 1 Hello, Android

A Little Background

See Full Table of Contents Cover

You May Also Like

Top Picks



Similar Authors



David Flanagan Frank Ableson Ed Burnette Elisabeth Freeman Steve McConnell Larry Wall Tony Stubblebine Eric A. Meyer

See More Recommendations

Reader Reviews

Be the First to Write a Review

Tags

linux software android programming technical software development mobile

Browse Tags

Browse Shelf



QA76.76.A65 M375 2013 QA76.76.A65 M49 2015

See Full Shelf

Catalog enrichment powered by Syndetics Unbound

Permanent Link

```
leader  01128cam a22003378a 4500
001     993796863505211
005     20220211111855.0
008     080625t20092009inua          000 0 eng
020     ##$a9780470344712 $qpaperback
020     ##$a0470344717 $qpaperback
035     ##$a(OCoLC)226356018
035     ##$a379686
035     ##$a(NjWP)379686-wmpatrsndb-Voyager
040     ##$aUKM $beng $cUKM $dBTCTA $dBAKER $dYDXCP $dBWX $dOCLCQ $dOKU $dOCLCQ $dSZ9XM
050     #4$aQA76.76.A65 $bM43 2009
100     1#$aMeier, Reto, $d1978-
245     10$aProfessional Android application development / $cReto Meier.
264     #1$aIndianapolis, IN : $bWiley, $c[2009]
264     #4$c©2009
300     ##$axxi, 409 pages : $billustrations ; $c24 cm.
336     ##$atext $btxt $2rdacontent
337     ##$aunmediated $bn $2rdamedia
338     ##$avolume $bnc $2rdacarrier
490     1#$aWrox professional guides
500     ##$a"Wrox programmer to programmer"--Cover.
650     #0$aAndroids.
650     #0$aApplication software $xDevelopment.
650     #0$aMobile computing.
830     #0$aWrox professional guides.
929     ##$aWP Materials Only
949     ##$lWPUpbook-general $d20090929
```

Chat with a librarian



BOOK

Professional Android application development

Meier, Reto, 1978-

2009

Available at Cheng Library 2nd Floor - Circulating (QA76.76.A65 M43 2009)

TOP
GET IT
DETAILS
SEND TO
VIRTUAL BROWSE
LINKS

Get It

Please sign in to check if there are any request options. Sign in

BACK TO LOCATIONS

LOCATION ITEMS

Cheng Library
Available , 2nd Floor - Circulating ; QA76.76.A65 M43 2009
(1 copy, 1 available, 0 requests)

Item in place
Sign in to request/loan options

Details

Title	Professional Android application development
Creator	Meier, Reto, 1978-
Subject	Androids Application software -- Development Mobile computing
Series	Wrox professional guides Wrox professional guides.
Publisher	Indianapolis, IN : Wiley
Creation Date	2009
Language	English
Format	xxi, 409 pages : illustrations ; 24 cm.
Source	Library Catalog
Identifier	ISBN : 9780470344712 ISBN : 0470344717 OCLC : (OCoLC)226356018

Send to

QR PERMALINK CITATION EXPORT BIBTEX EXPORT RIS PRINT EMAIL

See something that doesn't look right? Report a Problem

Virtual Browse



Chat with a librarian


g apps for
ıing
ss the
culum : a
ıcy-based ...
Hello, Android : introducing Google's mobile development ...
2008
Camel in Action ...
2011
Professional Android application development ...
2009
FBML essentials ...
2008
Links
Display Source Record

APPENDIX
GOYAL01

```
leader  01443cam a2200349Ia 4500
001     991011891959706011
005     20200731144943.0
008     060614s2006    caua          001 0 eng d
020     ##$a1590596390
024     3#$a9781590596395
035     ##$a(IXA)b61246360-01utau_inst
035     ##$a(OCoLC)70111508 $9ExL
040     ##$aSXP $cSXP $dUAF $dIXA
049     ##$aIXAA
090     ##$aQA76.73.J38 $bG695 2006
092     ##$a005.133
100     1#$aGoyal, Vikram.
245     10$aPro JAVA ME MMAPI : $bmobile media API for Java Micro Edition / $cVikram Goyal.
260     ##$aBerkeley, CA : $bApress ; $aNew York : $bDistributed by Springer, $cc2006.
300     ##$axvii, 267 p. : $bill. ; $c25 cm.
490     1#$aExpert's voice in Java
500     ##$aIncludes index.
583     ##$adigitized $ztemporary access $fHathiTrust $uhttps://www.hathitrust.org/digital_library $ztemporary access $fHathiTrust $uhttps://www.hathitrust.org/digital_library
583     ##$adigitized $ztemporary access $fHathiTrust $uhttps://www.hathitrust.org/digital_library
650     #0$aJava (Computer program language)
650     #0$aWireless communication systems.
830     #0$aExpert's voice in Java.
907     ##$a.b61246360 $b05-20-09 $c03-12-07
915     ##$a(TxU)14353954
915     ##$a70111508
945     ##$aQA 76.73 J38 P3754 2006 $e03-04-2008 19:22 $g1 $i0591730169369S3 $j0 $lpcg   $o- $p$0.00 $q- $r- $s-   $t10 $u1 $v1 $w0 $x0 $y.i172332813 $z03-12-07
998     ##$apcg $b03-01-07 $cm $da   $e- $feng $gcau $h0 $i1
```



BOOK

Pro JAVA ME MMAPI : mobile media API for Java Micro Edition / Vikram Goyal.

Goyal, Vikram.
c2006
Expert's voice in Java.

Not available - Loan

TOP
SEND TO
GET IT
LINKS
DETAILS
VIRTUAL BROWSE

Send to

EXPORT BIBTEX EXPORT RIS QR EXPORT TO EXCEL ENDNOTE PRINT
CITATION LINK TO RECORD EMAIL

Get It

Please sign in to check if there are any request options. Sign in

BACK TO LOCATIONS

LOCATION ITEMS

Perry-Castañeda Library
Out of library , PCL Stacks ; QA 76.73 J38 P3754 2006
(1 copy, 0 available, 0 requests)

LOCATE

On loan until 09/08/2025 18:45:00 CDT
Loanable

Links

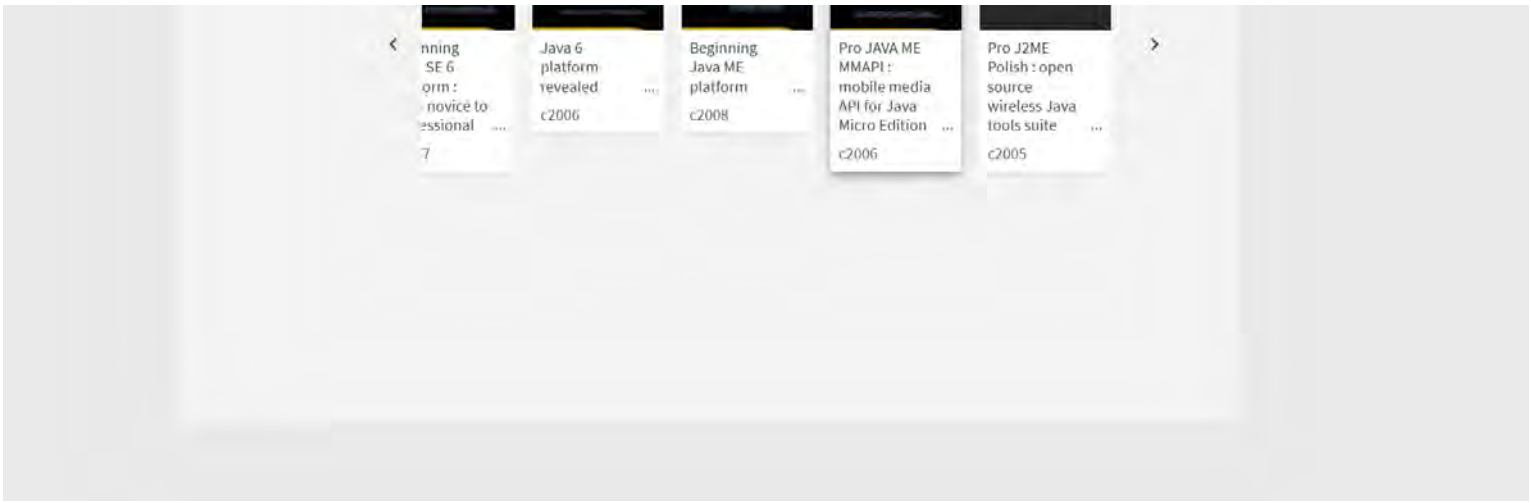
Display Source Record

Details

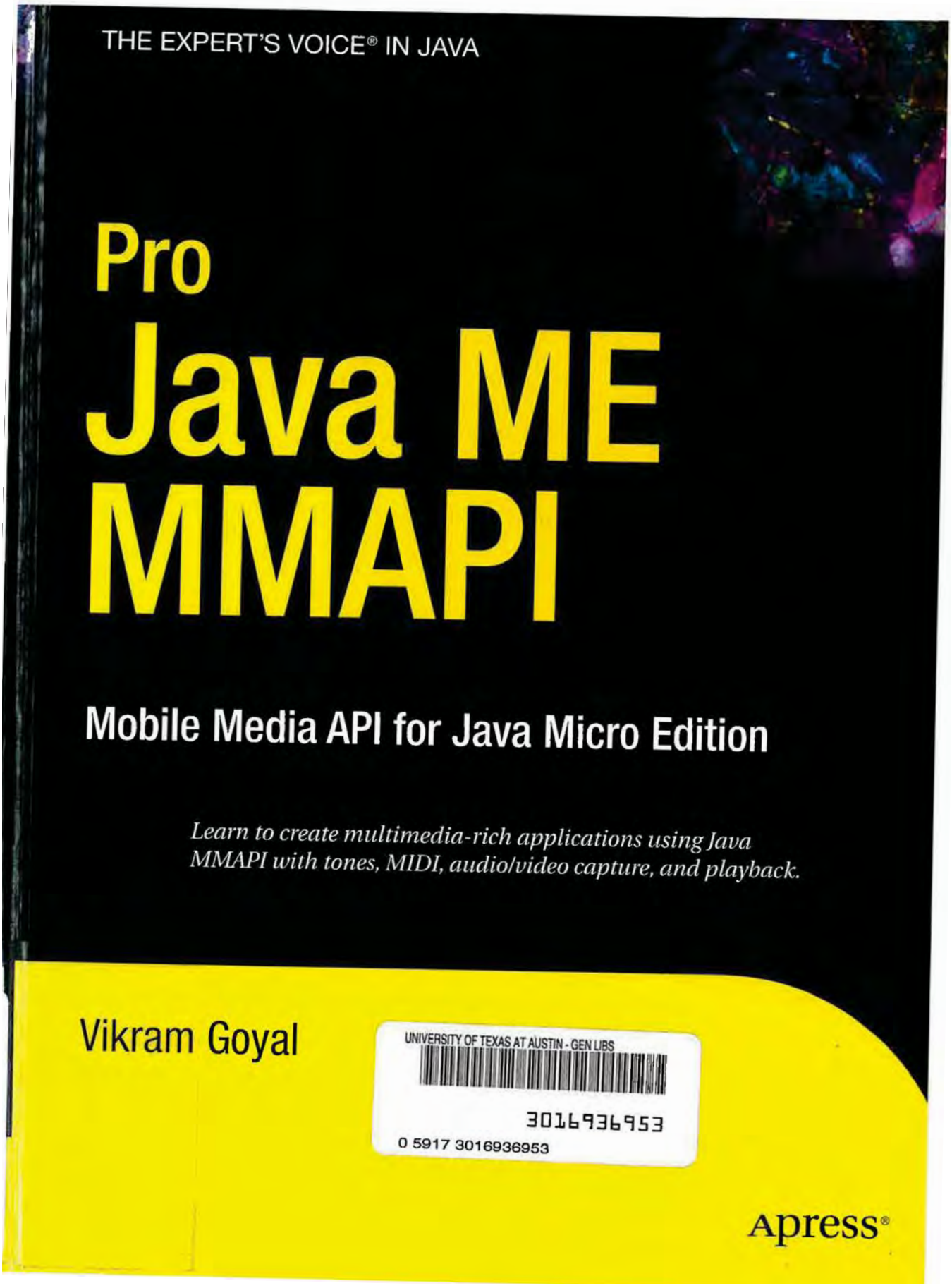
Title	Pro JAVA ME MMAPI : mobile media API for Java Micro Edition / Vikram Goyal.
Creator	Goyal, Vikram.
Publisher	Berkeley, CA : Apress ; New York : Distributed by Springer, c2006.
Format	xvii, 267 p. : ill. ; 25 cm.
Series	Expert's voice in Java.
Notes	Includes index.
Subject	Java (Computer program language) Wireless communication systems
Type	Book
Creation Date	c2006
Identifier	ISBN : 1590596390 OCLC : (OCoLC)70111508
Source	Library Catalog

Virtual Browse




nning
SE 6
orm :
novice to
essional
7
Java 6
platform
revealed
c2006
Beginning
Java ME
platform
c2008
Pro JAVA ME
MMAPI :
mobile media
API for Java
Micro Edition
c2006
Pro J2ME
Polish : open
source
wireless Java
tools suite
c2005

APPENDIX
GOYAL02


THE EXPERT'S VOICE® IN JAVA
Pro
Java ME
MMAPI
Mobile Media API for Java Micro Edition
Learn to create multimedia-rich applications using Java MMAPI with tones, MIDI, audio/video capture, and playback.
Vikram Goyal
UNIVERSITY OF TEXAS AT AUSTIN - GEN LIBS
3016936953
0 5917 3016936953
Apress®


University of
Texas Libraries
THE UNIVERSITY OF TEXAS AT AUSTIN

THE UNIVERSITY OF TEXAS AT AUSTIN
UNIVERSITY OF TEXAS LIBRARIES

DUE	RETURNED
	RET'D PCL JUL 08 2008

Pro Java ME MMAPI

Mobile Media API for Java Micro Edition

Vikram Goyal

Apress®

Pro Java ME MMAPI: Mobile Media API for Java Micro Edition

Copyright © 2006 by Vikram Goyal

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-59059-639-5

ISBN-10: 1-59059-639-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Steve Anglin
Technical Reviewer: Robert Virkus
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Julie McNamee
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Nancy Riddiough
Indexer: Carol Burbo
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

In loving memory of my Papa, who would go to the ends of the Earth to find that one particular book. This one is for you.

Pro Java ME MMAPI: Mobile Media API for Java Micro Edition

Copyright © 2006 by Vikram Goyal

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-59059-639-5

ISBN-10: 1-59059-639-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Steve Anglin
Technical Reviewer: Robert Virkus
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Julie McNamee
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Nancy Riddiough
Indexer: Carol Burbo
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

In loving memory of my Papa, who would go to the ends of the Earth to find that one particular book. This one is for you.

Contents at a Glance

About the Author xiii
About the Technical Reviewer xv
Acknowledgments xvii

CHAPTER 1 Introducing Mobile Media API (MMAPI) 1
CHAPTER 2 MMAPI Architecture 7
CHAPTER 3 Getting Started with MMAPI 19
CHAPTER 4 Media Player Lifecycle and Events 39
CHAPTER 5 Accessing Media Over the Network 59
CHAPTER 6 Creating and Playing Tones Using ToneControl 73
CHAPTER 7 Managing MIDI Using MIDIControl, TempoControl, and PitchControl 95
CHAPTER 8 Working with Audio and Video 127
CHAPTER 9 Case Study: Device Blogging 185
APPENDIX A Mobile Media API (MMAPI) Reference 239
APPENDIX B URI Syntax for Media Locators 249
APPENDIX C Advanced Multimedia Supplements—JSR 234 251

INDEX 255

Contents

About the Author xiii
About the Technical Reviewer xv
Acknowledgments xvii

CHAPTER 1 Introducing Mobile Media API (MMAPI) 1

What Is Mobile Media API (MMAPI)? 1
MMAPI Features and Requirements 2
How Do I Get MMAPI? 3
How Does MMAPI Fit with MIDP 2.0? 3
Who Supports MMAPI? 5
Summary 6

CHAPTER 2 MMAPI Architecture 7

Understanding Players and DataSources 7
DataSource Basics 8
Player Basics 9
Understanding the Manager Class 9
Creating Player Instances 10
Creating Player Instances for MIDI and Tone Sounds 11
Supported Protocols and Content Types 11
Understanding Controls 13
The MIDP 2.0 Subset 14
Feature Set Implementations 15
Security Architecture 16
Summary 17

CHAPTER 3 Getting Started with MMAPI 19

A Simple Multimedia Player 19
Creating a Functional Player 22
Writing the Code 23
Improving AudioPlayerCanvas: Caching Player Instances 29
Summary 37

CHAPTER 4 Media Player Lifecycle and Events 39

Overview 39
Exploring the Different Player States 39
UNREALIZED 40
REALIZED 41
PREFETCHED 43
STARTED 44
CLOSED 46
Responding to Player Events 47
Understanding the Event Delivery Mechanism 50
Creating an Event Handling Class 51
Handling a Custom Event 55
Summary 58

CHAPTER 5 Accessing Media Over the Network 59

Understanding Threads in Java ME 59
Understanding Permissions for Network Access 63
Putting It Together 67
Summary 71

CHAPTER 6 Creating and Playing Tones Using ToneControl 73

Understanding Synthetic Tones 73
Understanding Note, Pitch, and Frequency 74
Using a MMAPI Formula to Calculate Note Values 76
Using the playTone() Method 79
Using Tone Sequences with ToneControl 81
Defining Tone Sequences 82
Playing Sequences Using ToneControl and Player 88
Distributing Tone Sequences 89
Summary 93

CHAPTER 7 Managing MIDI Using MIDIControl, TempoControl, and PitchControl 95

Understanding MIDI 95
Understanding the MIDI Specification 96
Storing and Distributing MIDI Messages 101

Using MIDI in MMAPI 102
Playing MIDI Without MIDIControl 102
Playing MIDI with MIDIControl 113
Summary 125

CHAPTER 8 **Working with Audio and Video** 127

Querying the Capabilities of Your Device 127
Understanding Media Encodings 132
A Brief Overview of Sampled Audio 133
Storing Sampled Audio 133
Controlling Sampled Audio 134
Setting Preset Stop Times with StopTimeControl 134
Gathering Information Using MetaDataControl 139
Mixing Sampled Audio, MIDI, and Tones 142
Capturing Audio 146
Timed Capture and Playback 147
Controlled Capture and Playback 151
Saving Captured Audio 158
Capturing Audio from Existing Audio 162
Working with Video 163
Displaying Video 163
Positioning Video and Controlling Volume 169
Capturing Video and Images 173
Seeking Video Frames with FramePositioningControl 181
Streaming Media 181
Summary 184

CHAPTER 9 **Case Study: Device Blogging** 185

About the Companion Web Site 185
The Finished MIDlet in Action 185
MIDlet Startup 186
Registering with the Device Blog Web Site 186
Logging in to Device Blog Web Site 187
Creating and Posting Blog Entries 188
Creating the MIDlet Design 192
The Model 192
The View 194
The Control 197

Creating the MIDlet Code 198
Creating the Model Classes 198
Creating the Utility Classes 205
Creating the View 211
Creating the Control Classes 227
On the Server Side 232
Summary 237

APPENDIX A Mobile Media API (MMAPI) Reference 239

Package javax.microedition.media 239
Interface Control 239
Interface Controllable 239
Class Manager 240
Class MediaException 240
Interface Player 240
Interface PlayerListener 241
Interface TimeBase 242
Package javax.microedition.media.control 242
Interface FramePositioningControl 242
Interface GUIControl 242
Interface MetaDataControl 242
Interface MIDIControl 243
Interface PitchControl 243
Interface RateControl 243
Interface RecordControl 244
Interface StopTimeControl 244
Interface TempoControl 244
Interface ToneControl 245
Interface VideoControl 245
Interface VolumeControl 246
Package javax.microedition.media.protocol 246
Class ContentDescriptor 246
Class DataSource 246
Interface SourceStream 247

APPENDIX B URI Syntax for Media Locators 249

APPENDIX C Advanced Multimedia Supplements—JSR 234 251

Introduction to JSR 234 251
GlobalManager 252
Spectator 252
Module 252
MediaProcessor 252
Controls 252
JSR 234 Implementations 253

INDEX 255

About the Author



VIKRAM GOYAL is a software developer living in Brisbane, Australia. Vikram writes on Java development issues for the mobile environment at *http://today.java.net/pub/au/179*. You can contact him at *vikram@mmapibook.com*.

About the Technical Reviewer



ROBERT VIRKUS is the architect and lead programmer for the open source project J2ME Polish. He is an internationally recognized J2ME expert and is a member of Mobile Solution Group, Bremen, Germany.

After studying law and computer science in Bremen, Germany, and Sheffield, England, Robert began working in the mobile industry in 1999. He followed WAP and J2ME from their very beginnings and developed large-scale mobile betting applications.

In 2004, he founded Enough Software, the company behind J2ME Polish.

In his spare time, Robert enjoys the company of his friends and his dog, Benny. Other spare-time favorites are going to concerts of soul, ska, and punk-rock bands, and playing around with old computers such as Atari 400, Commodore 8296, and MCS Alpha 1.

Acknowledgments

Even though you see my name at the front of this book, it's the result of a collective effort on the part of several individuals. All these individuals need to be honored and acknowledged because without them, you would not be holding this book in your hands.

In order of my acquaintance with them, I would like to start with **Gary Cornell**, Apress Publisher and **Steve Anglin**, Java Editor at Apress for having the vision to build a comprehensive library of wireless Java books. **Tina Nielsen** as the Publisher's Assistant was quick with the contracts ensuring a speedy startup on the writing process. **Sofia Marchant**, the Project Manager for this book, was like a conductor who ensured that the various harmonies blended together perfectly to produce a book on time. **Robert Virkus**, the Technical Editor, was very forthcoming with his analysis of the text and code, and the final book is a better copy because of his efforts. **Julie McNamee**, who I hope reads and corrects this before it gets to you, worked tirelessly as the Copy Editor ensuring that I meant what I wrote. **Laura Esterman**, as the Production Editor, had an eagle eye that did not let even the tiniest of errors pass through via the production process.

There are, of course, other people involved in the production of this book, whom I never met, and a big Thank You to all of them for a job well done.

Finally a note about the loved ones: My mum and dad worked very hard to bring me up, and I can never thank them enough for their persistence and hard work.

My wife, Shellie, has suffered because of the hours I have spent in "the office" and deserves a reward for her patience. More than that, she supports me to be all I can and even puts up with my interference of her routine when I take away her mobile phone (to test the applications). To miss all the holidays and breaks and not complain is a feat in itself. I look forward to spending more time with her and to the upcoming addition to our family.

CHAPTER 1

Introducing Mobile Media API (MMAPI)

Embedding multimedia capabilities in a MIDlet is the next step in the evolution of MIDlets. Sun recognized early that audio and video are the future and introduced the Mobile Media API (MMAPI) via Java Specification Requests (JSR) 135 (*http://www.jcp.org/en/jsr/detail?id=135*).

The biggest requirement of the MMAPI specification was to ensure compatibility with small footprint devices while creating a specification that would be scalable for future—possibly more capable—devices. To this end, MMAPI has succeeded tremendously.

This chapter introduces you to MMAPI and explains several factors that make it a successful specification. You'll learn how this API fits in the overall scheme for MIDlet creation with the Mobile Information Device Profile (MIDP) 2.0. The chapter concludes with information on the current list of devices that support this API.

What Is Mobile Media API (MMAPI)?

MMAPI is the optional API that developers use to embed advanced multimedia capabilities in any Java-enabled device. If you've been using the audio capabilities in MIDP 2.0, you've already been using a subset of MMAPI. This subset is a forward compatible version of MMAPI created for limited-capability devices.

MMAPI allows you to create applications for Java-enabled devices that can discover and use the multimedia capabilities of the device that they are running on. You can play different formats of audio and video files from the network, a record store, or a Java Archive (JAR) file; have advanced control over the playback of these files; capture audio and video and take snapshots; play MIDI files; generate and play back tones; stream radio over the network; and do a whole lot more.

To encourage device manufactures to use this API in their Java-enabled devices, MMAPI was designed specifically to be protocol and format agnostic. In other words, this API does not come with assumptions about the supported protocol for accessing multimedia content, nor does it makes assumptions about the formats that it would be able to play. Thus, different device manufactures implement this API in their own way and make it available with the protocols and formats that their devices can support. This characteristic makes MMAPI a high-level interface and allows it to be compatible with any Java configuration.

Most devices, however, support some basic protocols and formats; for example, most allow you to access media over HTTP and play the WAV file format for audio and MP3 for video.

If a device supports a particular media format, MMAPI may mandate some control over the functionality of that format to create control uniformity across different devices. Other formats may require entirely different controls that may or may not be mandatory. Chapter 2 provides more information about mandatory controls for different formats in the "Feature Sets Implementations" section.

MMAPI Features and Requirements

MMAPI was designed for Java-enabled mobile devices, but the design is intentionally general enough that any Java-enabled device can benefit from it. This forced a set of rules on the API designers that they had to adhere to:

- **Low footprint API**: Because the main target of this API is Java-enabled mobile phones, which are severely constrained for available memory, the API must be able to support media playback in the available memory. Typically, because the Java-enabled mobile phones run on the Connected Limited Device Configuration (CLDC), the memory available ranges between 128KB and 512KB. This is the memory available for the virtual machine, the core libraries, your MIDlets, *and* MMAPI. MMAPI's place with MIDP is covered in the "How Does MMAPI Fit with MIDP 2.0?" section later in this chapter.
- **Ability to support multiple media types**: By defining the core API as a set of interfaces, MMAPI is protocol and format agnostic. Device manufactures supply their own implementation of MMAPI and implement interfaces that support the multimedia capabilities of their devices. This allows a wide range of protocols and formats to be supported based on the device, without any hard wiring of protocols and formats built-in the API itself. This makes MMAPI immensely scalable as new formats are discovered and supported by device manufacturers.
- **Support for basic controls**: Although the previous requirement states that MMAPI is protocol and format agnostic, some support for basic controls is guaranteed to be present. This creates uniform procedures for managing media, whatever format or protocol it may take. For example, all media can be played, started, or stopped.
- **Support for device capabilities discovery**: Similar to the previous requirement, all devices can be queried to discover their capabilities. This allows you to find out which protocols or formats the device supports.
- **Support for basic audio and tone generation**: Because MMAPI uses CLDC as the base minimum supported configuration, it requires some support for audio playback and tone generation. Note that the API mandates support for audio playback but doesn't restrict it to any particular format or protocol, in keeping with the ability to support multiple media types requirement. Device manufacturers are free to choose which format or playback they will support. Tone generation is important in Java-enabled mobile phones, and therefore, the API provides simple ways to play tones as well.

These requirements have led to an API interface that is truly extensible and capable of supporting a range of multimedia formats and protocols on an array of devices.

How Do I Get MMAPI?

If you have used the Java Wireless Toolkit (*http://java.sun.com/products/sjwtoolkit/download-2_3.html*) to develop your MIDlets, you already have MMAPI installed. The Toolkit comes with a reference implementation (RI) of this API installed as an optional package. Of course, MIDP 2.0 contains a subset of this API, so if you are only going to use a limited subset of MMAPI, you don't need this RI.

Most development environments (such as Netbeans and Eclipse) that support mobile application development include a version of this Toolkit, so MMAPI is included as an optional package.

Of course, the RI supplied with the Toolkit may not be your target platform (in all likelihood, it *won't* be your target platform because it contains only virtual devices). For example, you may be developing applications for the mobile devices supplied by Nokia. In which case, you need to download the implementation of MMAPI supplied by Nokia for its devices. This implementation will come bundled with the Toolkit supplied by Nokia (*http://forum.nokia.com/main/0,,034-2,00.html*). Similarly, different device manufacturers, and not just mobile device manufacturers, will supply their own implementations bundled in with their overall Java Wireless Toolkit.

In short, to start developing multimedia applications for Java-enabled devices using MMAPI, you need the Java Wireless Toolkit supplied by the device manufacturers. The last section in this chapter points you to some popular Toolkits. In this book, I will develop multimedia MIDlets using the Sun supplied Java Wireless Toolkit 2.3 and the Motorola SDK V5.2 (*http://www.motocoder.com*). In Chapters 8 and 9, I will also use the BenQ (formerly Siemens) CX 75 emulator (*http://www.benqmobile.com/developer*). The Sun Wireless Toolkit will be integrated in the Netbeans Integrated Development Environment (IDE) (*http://www.netbeans.org*), which will be the main development environment for the examples in this book. The Motorola device emulators will be used to test the MIDlets before deploying them on an actual Motorola device (the Motorola C975), and Chapters 8 and 9 will use the BenQ CX 75 emulator before testing them on the BenQ M75 device.

How Does MMAPI Fit with MIDP 2.0?

MMAPI is an optional package for the Java Micro Edition (ME) platform. MIDP 2.0 is a profile for the development of MIDlets, or applications for Java-enabled mobile devices, such as mobile phones and PDAs. MMAPI can be used with not just MIDP 2.0, but with any Java ME profile and configuration, provided an implementation is available for the device you are developing for. Refer to *Beginning J2ME: From Novice to Professional*, Third Edition, by Jonathan Knudsen and Sing Li (Apress, 2005) for a review of configurations, profiles, and development of MIDlets.

As you may already know, MMAPI is not the only optional package available for developing applications for the Java ME platform. Other prominent packages include the Web Services API (JSR 172), the Mobile 3D Graphics API (JSR 184), and the Location API (JSR 179). All of these optional packages, like the MMAPI package, are applicable to all available configurations, such as CLDC and Connected Device Configuration (CDC). On the other hand, an optional package such as the Wireless Messaging API (JSRs 120/205) is only relevant to the CLDC-based MID profile (MIDP).

MMAPI differs from the other optional packages because a scaled down version of it is present in MIDP 2.0 under the `javax.microedition.media` and `javax.microedition.media.control` packages. No other optional package makes an appearance in MIDP 2.0. So if you are developing applications that only require basic audio control, you don't need the full MMAPI installed or available. You would still be using MMAPI, just not the optional and bigger part of it. Your

application will run on all devices that support MIDP 2.0, as MIDP 1.0 does not have the scaled version.

Note Some devices support the MMAPI in MIDP 1.0 as an optional library, for example, Nokia 3650, Nokia 7650, and Sony Ericsson 610. Applications that use the features of the optional package will only run on devices that have both MIDP 2.0 and the optional MMAPI package installed. The next section includes a comprehensive list of all such devices at the time of the book going to print.

Figure 1-1 gives a bird's eye view of how MMAPI, along with the other optional packages, fits in the development of applications for Java-enabled devices. Because this book concentrates on applications developed using MIDP, which are called MIDlets, I'll use that term when talking about such applications.

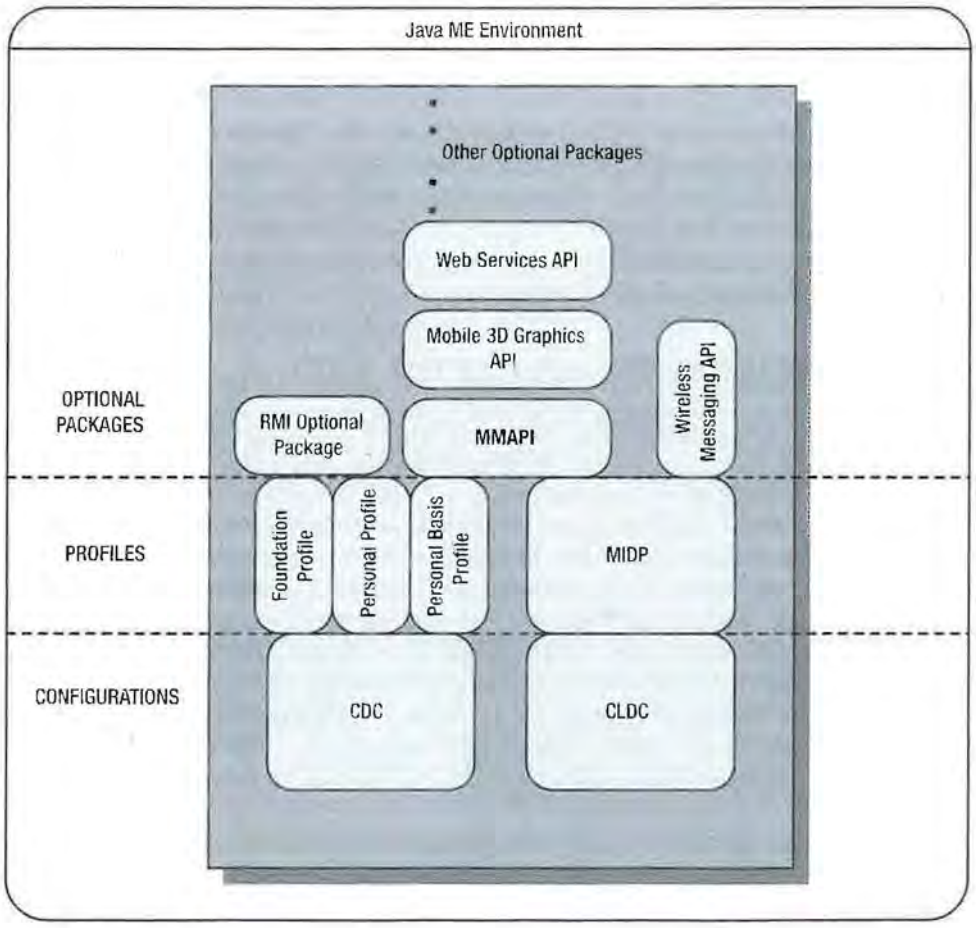


Figure 1-1. *Java ME development environment showing MMAPI as an optional package*

Who Supports MMAPI?

The specification for MMAPI was developed using the Java Community Process (JCP) (*http://www.jcp.org*) by a consortium of organizations, including device manufacturers, network operators, and multimedia companies, besides Sun Microsystems. This specification is numbered 135 and can be accessed at *http://www.jcp.org/en/jsr/detail?id=135*. The initial expert group for the development of this specification was composed of Sun, Nokia, and Beatnik.

Afterwards, several other organizations chipped in and lent their support for the development of this specification. A final release with a version number of 1.0 occurred on June 27th, 2002. A maintenance release (version 1.1) incorporating security enhancements and minor modifications was released a year later on June 26th, 2003, and is the version used for the examples in this book. Recently, JSR 234 has been released that identifies advanced supplements for this MMAPI.

As part of the development of this specification, Sun released a reference implementation of this API and a technology compatibility kit (TCK) that can be used to certify that an implementation of this API is compatible with the specification. This is par for any specification developed using the JCP. As stated earlier, this RI is bundled with the Java ME Wireless Toolkit supplied by Sun. Different device manufacturers supply their own implementations compatible with the TCK.

All devices that support MIDP 2.0 automatically support the scaled-down version of MMAPI. Many devices support the full version as an optional package. Table 1-1 gives a comprehensive list of devices that support MMAPI as an optional package.

Table 1-1. *Devices That Support MMAPI As an Optional Package*

Device Manufacturer	Supported Devices	Wireless Toolkit
Alcatel	One Touch 756	*http://www.my-onetouch.com*
BenQ	AX75 (MIDP 1.0), C70, C75, CF75/76, CL75, CX70/EMOTY, CX75, M75, S75, SL75, SXG75	*http://www.benq.com/developer*
Motorola	C975, E1000, A1000, A630, A780, A845, C380, C650, E398, E680, SLVR, T725, V180, V220, V3, V300, V303, V360, V400, V500, V525, V550, V551, V600, V620, V635, V8, V80, V980, i730	*http://www.motocoder.com*
Nokia	All Series 40, Series 60, and Series 80 based devices	*http://forum.nokia.com/main.html*
Samsung	E310, E380, E710, D400, P705, D410, 176X192 Series, E810, E310	*http://uk.samsungmobile.com/club/developers_club/cl_de_sdk_01.jsp*
Sony-Ericsson	W900, Z600, T610, T616, T618, V600, W800, K608, W550, W600, z520, D750, Z800, K600, K750, K300, K500, K700, J300, V800, Z500, S700, Z1010	*http://developer.sonyericsson.com*

Summary

This chapter introduced MMAPI, an optional package for the development of MIDlets that have embedded audio/video capabilities.

MMAPI provides these capabilities by defining a specification that is protocol and format agnostic, thereby increasing general acceptance and uptake. This has made MMAPI very popular and more and more devices support it.

With this introduction to MMAPI's features and requirements complete, the next chapter addresses the MMAPI architecture, which is key to understanding and using MMAPI.

CHAPTER 2

MMAPI Architecture

The MMAPI specification was created to cater to the widest range of multimedia options, and this is reflected in its architecture. It supports the most basic audio functions in a constrained device and at the same time provides ways to handle advanced multimedia data on more capable devices. MMAPI supports data from various sources and in various formats. MMAPI's protocol and format agnostic characteristic is achieved by a well-designed, high-level interface.

This chapter explains how the MMAPI specification achieves this aim of platform and format neutrality. You'll become familiar with the details of its architecture and the high-level objects that make it a successful specification. You'll see how the subset of MMAPI present in MIDP 2.0 differs from the overall specification. Finally, the chapter covers several small details of the MMAPI architecture, such as feature sets and security scenarios, to help you better understand and use MMAPI.

Understanding `Player`s and `DataSource`s

MIDlets can receive multimedia data from a variety of sources: located in a MIDlet's JAR file; come over the network via several different protocols (HTTP, RTP); sourced from the device's record store on which the MIDlet is running; and even come from user input via an audio or video capture device. In short, not only can this data be sourced from different locations, but also these locations can be accessed using any number of present or future protocols. At a conceptual level, MMAPI can handle all such locations and protocols without getting tied down to a specific protocol or location.

After data has been sourced, MMAPI should be able to process this data and make it available for rendering on the device. Processing multimedia data is inherently complex and requires the API to not only understand the data but decode it as well. Rendering the data and providing controls to manipulate it further complicates the process. Because MMAPI is a high-level interface, it doesn't mandate any specific data-processing functionality, besides a very low level of audio handling.

To achieve this neutrality with regard to sources of multimedia data and processing, MMAPI encapsulates these concepts into two high-level objects: `Player` and `DataSource`. Whereas `Player` is an interface that deals with processing and playing multimedia data, `DataSource` is an abstract class that encapsulates the task of data location and retrieval while maintaining protocol independence. The `Player` interface is defined in the `javax.microedition.media` package, and `DataSource` is defined in the `javax.microedition.media.protocol` package.

Player and DataSource work together to provide multimedia capability in a device. A Player instance parses data supplied to it from a DataSource instance. The Player may then render this data on the device and provide controls to manipulate it. For example, an audio Player can provide volume controls to increase or decrease the playback volume.

DataSource Basics

The DataSource class provides access to multimedia files by locating and opening a connection to them. You may not need to use this class directly, unless you are going to create a DataSource for a custom protocol or location. The MMAPI implementation provided by the device that you are targeting your MIDlet to will provide enough DataSource instances to satisfy most requirements.

Each DataSource is composed of one or more streams, known as a SourceStream (an interface in the javax.microedition.media.protocol package). A SourceStream is used to abstract a single stream of media data. Conceptually, multimedia data may be composed of several streams. For example, video files may be broken down into separate streams, one for audio and one for video. Thus, each individual stream can be acted on, independently of the other stream, thereby giving greater control than is possible otherwise. Most DataSource instances are composed of a single SourceStream.

A SourceStream also has the advantage of being randomly seekable, because it provides the seek(long where) method. This is an important consideration for multimedia data because it allows you to render this data from an arbitrary position. Of course, a particular stream may not support random seek operations, and this is reflected by the getSeekType() method. This method returns one of three integer constants, shown in Table 2-1.

Table 2-1. *Constants to Test a* SourceStream *for Seekability*

Constant	Description
NOT_SEEKABLE	The specified stream is not seekable.
SEEKABLE_TO_START	The specified stream can only be sought to its beginning.
RANDOM_ACCESSIBLE	A true seekable stream, this constant indicates that the specified stream can be randomly accessed anywhere.

A SourceStream has one more advantage. For multimedia data that supports it, a SourceStream can be used to read the logical transfer size of the data. For example, video data is normally composed of individual frames, so the logical transfer size of the data would be the size of one such frame. This allows you to create an in-memory buffer of the right size, equivalent to or greater than the size of the frame, to read data, thereby creating an efficient data read process. SourceStream instances that support this concept return a positive integer value for the method getTransferSize() and return -1 if they don't support it.

A DataSource instance is created by providing the location of the media to its constructor: DataSource(String locator). The locator parameter is specified in a Uniform Resource Identifier (URI) (*http://www.ietf.org/rfc/rfc2396.txt*) syntax—<scheme>:<scheme-specific-part>—which delineates the delivery protocol, the location of the media file, and its name. You are already familiar with HTTP for accessing files, which provides an easy example:

```
http://www.mmapibook.com/resources/media/audio/chapter2/siren.wav
```

where `http` refers to the protocol, `http://www.mmapibook.com/resources/media/audio/chapter2` to the location, and `siren.wav` to the name of the file. Appendix B gives an overview of this syntax.

Player Basics

The `Player` interface is your handle to playing and managing your multimedia data. It provides methods to render, control, and synchronize the data with other players, and listen to player events such as starting, stopping, and pausing the data.

All media types, except for simple tones, require a `Player` instance for playback and control. A media type is mapped to a `Player` instance that can handle its playback by evaluating the extension of the file that contains the multimedia data and mapping it to a MIME type. For example, if you wanted to play an audio file with the `wav` extension, a `Player` instance that can handle the MIME type `audio/x-wav` is created. Similarly, a file with an `mp4` extension results in the creation of a `Player` instance that can play `video/mp4`. In the next section, you'll learn about the `Manager` class that does this mapping between MIME types and `Player` instances.

`Player` instances are independent of the media type that they are playing and only depend on the implementation provided by the device manufacturer. This allows you to use generic methods to manipulate this data. So to start the playback of *any* multimedia data, you use the `start()` method, which starts playback as soon as possible. To pause, you use the `stop()` method, which pauses the playback. To close the data and release any resources associated with it, you use the `close()` method, and so on. Chapter 4 will introduce you to all the methods that you can use to control media playback.

If you are creating a multimedia MIDlet that is responsible for simultaneously playing multiple `Player` instances, you will run into synchronization issues, for example, making sure that video playback starts simultaneously with a separate audio track. In such cases, synchronizing different `Player` instances can be a problem without a timing reference. Each `Player` instance has a way to measure time using the `TimeBase` interface. This interface provides a constantly ticking source of time, and each `Player` instance has a default `TimeBase` instance built in. You can access this default instance by using the method `getTimeBase()`; you can override this default instance with your own implementation by using the `setTimeBase(TimeBase base)` method. To synchronize two `Player` instances so that they use the same `TimeBase`, you can use `player2.setTimeBase(player1.getTimeBase())` code. To reset a `Player` instance's `TimeBase`, you can pass a `null` value to the `setTimeBase(TimeBase base)` method.

`Player` instances generate events during different stages of their working. By implementing the `PlayerListener` interface, you can receive notifications of these asynchronous events. This interface defines several common events, but you can create your own proprietary events as well. Chapter 4 discusses the event management of `Player` instances in detail.

Understanding the `Manager` Class

The `Manager` class in the `javax.microedition.media` package provides developers access to `Player` instances. These `Player` instances, as you already know, source their data from `DataSource` instances. In a nutshell, a `Manager` class is a bridge between a `DataSource` and a corresponding `Player` instance. Figure 2-1 shows this relationship.

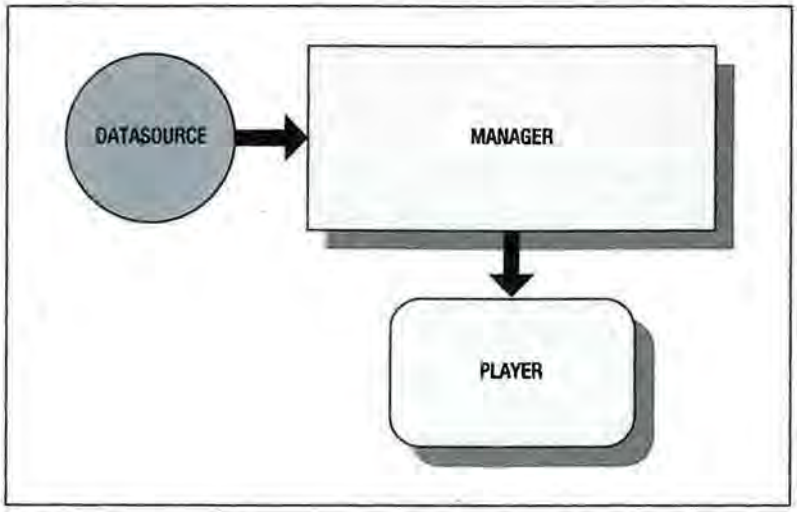


Figure 2-1. *The* `Manager` *class is the bridge between a* `DataSource` *and a* `Player`.

Creating `Player` Instances

Using methods of the `Manager` class is the only way to create `Player` instances in MMAPI. To this end, this class provides three such methods. Not surprisingly, all the methods are called `createPlayer()` and are static. The methods differ in the way the location of the multimedia data is provided to them.

As expected, `createPlayer(DataSource source)` is one of these methods. You create an instance of a `DataSource` using the `DataSource(String locator)` constructor and pass this to the `createPlayer()` method. The other two methods eventually call this method internally to create the `Player` instance.

You are more likely to use the `createPlayer(String locator)` method because it is more convenient than the previous method. Instead of having to create a `DataSource`, you only specify the location of the multimedia data using the `locator` parameter. The method internally translates it into a `DataSource` and creates a `Player` that is suitable for handling the multimedia content.

The third method, `createPlayer(InputStream is, String type)`, provides a slightly different way of creating players. It allows you to create an `InputStream` on the data and specify the content type of the data yourself, because the MIME type of the data cannot be determined with file extensions. This method has a further disadvantage because by using an `InputStream` instead of a `SourceStream`, it disallows random seek operations within the multimedia data. `Player` instances created with this method are not likely to support such operations.

Creating Player Instances for MIDI and Tone Sounds

Some Player instances do not require a physical file or location. For example, when you create a MIDlet that will play tones or Musical Instrument Digital Interface (MIDI) sounds, you need to tell the device to use its built-in MIDI synthesizer to play these sounds based on a supplied MIDI or tone sequence. The Manager class provides constants for creating Player instances based on these requirements. Table 2-2 gives details of these constants.

Table 2-2. *MIDI and Tone* Player *Constants*

Constant Name	Constant Value	Content Type	Usage
MIDI_DEVICE_LOCATOR	device://midi	audio/midi	Manager.createPlayer (Manager.MIDI_DEVICE_LOCATOR)
TONE_DEVICE_LOCATOR	device://tone	audio/x-tone-seq	Manager.createPlayer (Manager.TONE_DEVICE_LOCATOR)

You can still create MIDI and tone Player instances by specifying physical locations for them. For example, to create a MIDI Player based on a physical file, the extension of the file should be either mid or kar. Similarly, for a tone Player instance the extension of the file should be jts.

Because most simple devices support tone generation, the Manager class goes one step further by providing a static method to play single tones: playTone(int note, int duration, int volume). This provides a handy method that plays a single note for the specified duration and at the specified volume. Chapter 6 covers tone playing in detail.

Supported Protocols and Content Types

With MMAPI development for disparate devices, it becomes imperative to know beforehand what protocols are supported and what multimedia content can be rendered by the target device. The Manager class provides two methods that can help you understand the capabilities of a target MMAPI implementation.

The static getSupportedContentTypes(String protocol) method provides a list of all supported content types for a particular protocol. If you pass a null parameter to this method, it will give you a list of all content types supported by this device's MMAPI implementation. The static getSupportedProtocols(String contentType) method does the opposite; given a content type, it tells you all the protocols over which it can be accessed. If you pass a null parameter to this method, it will give you a list of all protocols supported by the device's MMAPI implementation.

Table 2-3 lists the output of running these methods on three different environments.

Table 2-3. *Supported Protocols and Content Types on Different MMAPI Implementations*

Method	Sun Java Wireless Toolkit 2.3	Motorola C975 Emulator	Motorola C975 Actual Device
getSupported ContentTypes (null)	audio/x-tone-seq, audio/x-wav, audio/midi, audio/sp-midi, image/gif, video/mpeg,← video/vnd.sun.rgb565	audio/x-wav audio/x-tone-seq audio/midi video/mng video/x-rgb	audio/amr, audio/x-amr, audio/amr-wb audio/x-amr-wb, audio/mp4, audio/x-mp4, audio/3gp, audio/mp3, audio/x-mp3, audio/mpeg3, audio/x-mpeg3, audio/mpeg, audio/x-mpeg, audio/mpg, audio/x-mpg, audio/wav, audio/x-wav, audio/au, audio/x-au, audio/basic, audio/asf, audio/x-ms-wma, audio/x-pn-realaudio, audio/midi, audio/x-midi, audio/mid, audio/x-mid, audio/sp-midi, audio/mobile-xmf, audio/imelody, audio/ x-imelody, audio/imy, audio/x-imy, text/ x-imelody, audio/ x-tone-seq, video/mp4, video/x-mp4, video/3gp, video/asf, video/x-ms-asf, video/x-ms-wmv, video/x-pn-realvideo, video/vnd.rn-realvideo, application/x-pn-realmedia, application/ vnd.rn-realmedia, image/gif, audio, video, camera
getSupported Protocols(null)	device, http, file, capture	http, https, device, capture	file, rtsp, http, https, device, capture
getSupported ContentTypes ("capture")	audio/x-wav video/vnd.sun.rgb565	audio/x-wav video/x-rgb	audio, video, camera
getSupported Protocols ("audio/x-wav")	http, file, capture	http, https, capture	file, rtsp, http, https

As you can see from Table 2-3, the supported protocols and content types vary greatly depending on the target device or emulator. Even though the emulator for Motorola C975 should support the same protocols and content types as the actual device, the difference is huge, especially for the supported content types. Actual device configurations vary greatly from their supposed emulators, and this is a typical result. Your MIDlets should always be tested on actual devices before bringing them into a production environment.

Understanding Controls

MMAPI defines `Control` objects that provide fine-grained control over the functionality of a `Player`. For example, if you have created a `Player` object to play an audio file, you will want to control its volume. Rather than provide volume control as a method in the `Player` interface, the interface `VolumeControl` is used. Volume is a type of control that is applicable to audio files, indeed to almost all multimedia files, but there may be several such individual controls that are exposed by individual media types. For example, `FramePositioningControl` is a control that is exposed by video players to control precise positioning of individual frames. The `Control` interface in `javax.microedition.media` package is used to abstract this concept.

MMAPI defines some standard controls in the `javax.microedition.media.control` package, but you are free to create your own controls by implementing the `Control` interface. The controls in the `javax.microedition.media.control` package are all defined as interfaces, and it is left for actual MMAPI implementations to provide concrete classes for them. Table 2-4 lists these controls and provides a brief explanation of each control.

Table 2-4. *Standard MMAPI Controls*

Control Interface	Description
`FramePositioningControl`	A control for video data that allows access to individual frames.
`GUIControl`	A control for data that requires a display, such as video.
`MetaDataControl`	Used to determine the metadata information stored within a media stream, such as title, copyright, author, and so on.
`MIDIControl`	A fully functional control that enables access to a device's MIDI player.
`PitchControl`	Used to control the pitch (frequency) of audio data.
`RateControl`	Used to control the playback rate of a `Player`.
`RecordControl`	Allows you to control the recording of data from a capture device, such as video from a camera or audio from a sound recorder.
`StopTimeControl`	A control that allows you to set a preset time when you want the `Player` to stop playing.
`TempoControl`	Similar to `RateControl`, this control allows you to change the tempo (speed) of playback for an audio `Player`, typically, a MIDI `Player`.
`ToneControl`	A fully functional control that allows you to play monotonic tone sequences.
`VideoControl`	Extends `GUIControl` and controls the display of video.
`VolumeControl`	The simplest control that allows you to control the volume of audio in a `Player`.

To get a list of controls that a `Player` instance exposes, use the method `getControls()` on the `Player` instance. To get to an actual control, use the method `getControl(String controlType)` and pass to it the name of the control interface. For example, to access the `VolumeControl` on a `Player`, you can use the method `getControl("VolumeControl")`. These methods are not defined in the `Player` interface, but the `Controllable` interface that the `Player` interface implements. As a point of interest, both `SourceStream` interface and the `DataSource` abstract class implement the `Controllable` interface, which gives you access to type-specific controls, if available.

As you may expect, some Player instances may expose multiple controls. For example, a Player created to handle video data will expose both a VideoControl and a VolumeControl at the very least. An audio player may expose a VolumeControl, a RateControl, a PitchControl, and a ToneControl. Some Player instances may support no controls, in which case, getControls() returns a zero length array (and not a null value).

After you have access to a control, you have access to the behavior of that control (and control of the underlying multimedia data). Each control is different in terms of its behavior and exposes methods that manipulate this behavior. For example, FramePositioningControl has a method called seek(int frameNumber) that when invoked on a video Player, renders the specified frame in the video on to the display. The method skip(int framesToSkip) skips the given number of frames from the current video position. Similarly, other controls define their own methods that best allow you to manipulate the underlying multimedia data. Detailed descriptions of the controls listed in Table 2-4 are scattered throughout this book.

The MIDP 2.0 Subset

MMAPI is an optional package, which means that device manufacturers may choose not to include an implementation for it in their devices. However, if the profile on a device is MIDP 2.0, then the device includes a subset of MMAPI. This subset was created to provide a base common MMAPI implementation for the MID 2.0 profile.

This subset was created to solve two problems: (1) resource constrained devices do not have enough processing power to display video-based multimedia data, and (2) even if these devices were powerful enough, they would be constrained in terms of the size of a MMAPI implementation. To support extensive processing and multiple multimedia data formats, the size of a MMAPI implementation can be large.

To solve these problems, the MIDP 2.0 subset of MMAPI ignores some of the requirements of the complete set. Although still a forward compatible version of MMAPI, it ignores some of the controls and provides a scaled-down version of the Manager class. Table 2-5 lists how the MIDP 2.0 subset differs from the complete MMAPI implementation

Table 2-5. *MIDP 2.0 MMAPI Subset Features*

Feature	Description
Audio only	The subset only supports audio-based data. Thus, no video-based controls, such as VideoControl, GUIControl, or FramePositioningControl, are present, and any content type that is video-based is ignored.
No MIDI support	Even in audio, only simple tones and sampled audio is supported, ignoring MIDI. Thus MIDIControl is absent and only ToneControl and VolumeControl is provided.
No support for synchronization	The subset is missing the TimeBase interface, and the Player interface is without the setTimeBase() and getTimeBase() methods. Thus, multiple Player instances cannot be synchronized to play back simultaneously.
No support for recording	Even for audio data, recording and capturing are not enabled, and thus RecordControl is absent.
No support for controlling the rate of audio playback	TempoControl and RateControl are absent.

Feature	Description
Controlling audio frequency is not supported	`PitchControl` is absent.
Other missing controls	`StopTimeControl` and `MetaDataControl` are also not supported.
No support for custom protocols	The `javax.microedition.media.protocol` package is excluded. The `Manager` class, therefore, does not include the `createPlayer(DataSource source)` method.

These features make the subset a lightweight implementation that meets the design goals of MIDP 2.0, including support for low footprint audio playback functionality in MIDlets, support for tone generation, and consistent media playback options. At the same time, the subset maintains upward compatibility with the MMAPI specification and is a true subset.

Feature Set Implementations

With intrinsic support for such a diverse range of features, protocols, and file formats, MMAPI is a truly flexible specification. It doesn't mandate support for any formats but allows device manufacturers to build in support for formats that they deem fit. However, with this flexibility, application development becomes a bit more difficult. You can query for a device implementation to give you a list of supported protocols and formats, but are you guaranteed that it will behave how you expect it to?

To alleviate this concern and to bring about some uniformity in the way multimedia data is controlled, MMAPI specification defines feature sets. A *feature set* is a grouping of multimedia data that has some common features. The API defines required and expected behavior for each feature set if the feature set is implemented by a device manufacturer's implementation. This way, if you are developing applications that target a particular feature set, you are guaranteed that it will behave by exposing controls that are similar across different manufacturers.

The simplest feature set is called *sampled audio*. As far as MMAPI is concerned, sampled audio refers to any audio data that is digital in its format, ignoring its origin. If a device supports sampled audio, it *should* implement the `VolumeControl` and `StopTimeControl` interfaces. Note the emphasis on should, which implies that this is a recommended practice, which barring any adverse circumstance, must be implemented. The MIDP 2.0 MMAPI subset is good example. Because it supports sampled audio, it should implement both `VolumeControl` and `StopTimeControl`; however, it only implements `VolumeControl` and ignores `StopTimeControl`.

MIDI is the next feature set. If a device's MMAPI implementation declares that it supports the MIDI feature set, which means that it allows you to play external audio files with extensions of `mid` or `kar`, it *should* implement `VolumeControl`, `MIDIControl`, `TempoControl`, `PitchControl`, and `StopTimeControl`. Again, notice the emphasis on should. Although there's no guarantee that the device will support all these controls, there's a good chance that it will.

The *Tone Sequence* feature set is the first set that mandates a "must" requirement. If a device allows you to play tone sequences, it *must* implement the `ToneControl` interface. If the device doesn't implement this interface, then it does not conform to the MMAPI specification. The Tone Sequence feature set is applicable to `Player` instances that are created using the `Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR)` method.

Similar to the Tone Sequence set, the *Interactive MIDI* feature set is applicable to `Player` instances that are created using the `Manager.createPlayer(Manager.MIDI_DEVICE_LOCATOR)`

method, which allows you to create MIDlets that directly manipulate the device's MIDI player. This feature set *must* implement the `MIDIControl` interface.

Finally, the *video* feature set applies to all video data. If video is supported by a device's MMAPI implementation, it *must* allow you to control the video with the `VideoControl` interface. The implementation *should* also support `FramePositioningControl`, the `StopTimeControl`, and the `VolumeControl`, but it doesn't have to. If it doesn't support the last three controls, but supports `VideoControl`, it is still an MMAPI-compliant implementation.

Security Architecture

As you now know, MMAPI does not work in isolation; it is an add-on or optional package that works on top of a profile. In this book, we are concerned with exploring MMAPI for the profile for small devices, MIDP 2.0; however, as explained in Chapter 1, MMAPI can be run on any compatible profile and configuration.

The capability to acquire and render multimedia data on any profile or configuration, even with an optional package such as MMAPI, creates a few problems with the security architecture of the device in question. The device must be able to control certain aspects of the multimedia process, such as recording using a camera and microphone for devices that have them. Similarly, the network and device's file system should be accessed only by a permission-based model.

Keeping these issues in mind, MMAPI allows control over the methods that are security sensitive. However, the specification does not dictate a particular architecture, and leaves it to specific implementations to work out the exact details, much like the interfaces that it describes for `Player` and `Control` objects.

Methods that are security conscious throw a `SecurityException` if permission to run that method has not been granted. Table 2-6 lists all these methods, the classes/interfaces they reside in, and the permissions required to make them run.

Table 2-6. *Security Conscious Methods*

Method Name	Method Class/ Interface	Permission Required
`setRecordLocation(String locator)`	`RecordControl`	`javax.microedition.media.control.RecordControl`
`setRecordStream (OutputStream stream)`	`RecordControl`	`javax.microedition.media.control.RecordControl`
`getSnapshot(String type)`	`VideoControl`	`javax.microedition.media.control.VideoControl.getSnapshot`
`createPlayer(String locator)`	`Manager`	
`createPlayer(InputStream is, String type)`	`Manager`	
`createPlayer(DataSource source)`	`Manager`	
`realize()`	`Player`	
`prefetch()`	`Player`	
`start()`	`Player`	
`start()`	`DataSource`	
`connect()`	`DataSource`	

Permissions are granted by the profile that MMAPI is running in. This is, in turn, either inherently built in the profile—for example, if recording is not supported at all then permissions will be false irrespective of any other conditions—or requested from the user running an application on the device to explicitly enable them.

As you may expect, the first three methods in Table 2-6 relate to privacy issues. An application must not start recording, even if the underlying device can record, without an explicit permission from the user of the application. The rest of the methods are related with general security issues of network access for data retrieval and depend on the protection domain that the MIDlet is running in. If data has to be retrieved over HTTP, then the user must be asked if transmission over the network is permitted, as it may result in charges billed to the user.

For HTTP/HTTPS access, the required permissions are `javax.microedition.io.Connector.http` and `javax.microedition.io.Connector.https`, respectively. Permissions are specified in the descriptor file (the Java Application Descriptor or JAD file) for the MIDlet and can be granted to a signed MIDlet. Refer to *Beginning J2ME* by Knudsen and Li (Apress, 2005) for an understanding of the process required for enabling permissions in Java ME. Chapter 5 also contains a detailed example of this process.

Summary

The MMAPI architecture was purposefully built to be extensible and not cater to a particular format or protocol. `DataSources`, `Players`, and `Controls` make the MMAPI a very flexible API that gives freedom to its implementers to support any media types, and at the same time, be true to the contract specified between the MMAPI interfaces.

This chapter explained this contract by specifying the MMAPI architecture. You saw the way the interfaces interact with each other and how device manufacturers can implement them. Using three different MMAPI implementations, you saw the different contents and protocols that they supported and how they still provided support for the basic MMAPI/MIDP 2.0 subset. Finally, you understood the security issues involved in using the libraries and classes of MMAPI.

The next chapter will leave the theory of MMAPI behind for a moment to introduce your very first media player using MMAPI.

CHAPTER 3

Getting Started with MMAPI

By now, you know that MMAPI is an optional API for building multimedia-enabled MIDlets. You also know that creating a MMAPI application requires knowledge of not just the MMAPI architecture, but also the MIDlet architecture, which you should be well conversant with. MMAPI MIDlets don't require any modifications to the way MIDlets are created, and basic architectural knowledge of the API is enough to get you started creating the simplest of multimedia MIDlets. You have already learned about the architecture in Chapter 2, and in this chapter, you'll use this knowledge to create two simple MMAPI MIDlets.

The first MIDlet is an example of the simplest MMAPI MIDlet that can be created. It plays an audio file and shows basic interaction between the `Manager` class and the `Player` interface. The second MIDlet is much more advanced and shows the interplay between the `Manager` class and the `Player` and the `Control` interfaces.

Both MIDlets will give you grounding in MMAPI MIDlet creation. Consider them as the Hello World MIDlets of the MMAPI world.

A Simple Multimedia Player

The simplest MMAPI MIDlet that can be built allows you to easily play a multimedia file from within your MIDlet without worrying about controls, feature sets, or security architecture. If all you're doing is adding some sampled audio (or any other media) in a game, MMAPI allows you to do so in two lines of code. Listing 3-1 shows this code within a complete MIDlet.

Listing 3-1. *A Simple MMAPI MIDlet*

```
package com.apress.chapter3;

import javax.microedition.midlet.MIDlet;

import javax.microedition.media.Manager;
import javax.microedition.media.Player;

public class SimplePlayer extends MIDlet {

  public void startApp() {
```

```
    try {

      Player player =
        Manager.createPlayer(
          getClass().getResourceAsStream("/media/audio/chapter3/baby.wav"),
          "audio/x-wav");

      player.start();

    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

To keep things simple at this stage, the media file is played by creating an `InputStream` on a wav file, which is embedded in the MIDlet's JAR. This media file is kept in the folder `media/audio/chapter3` and is called `baby.wav` (which is the sound of a baby crying).

Of course, you don't need to play an audio file only. You can substitute the wav file with a video file, provided the emulator supports the format of the video file. The video will not show anywhere, because this listing doesn't provide a mechanism to show the video. You can substitute the wav file for a midi, tone, or any other supported audio format. The point is that playing multimedia files using the MMAPI is as simple as creating a `Player` instance using the `Manager` class and calling method `start()` on it.

Let's analyze the two lines that make up the core of this MIDlet.

```
Player player = Manager.createPlayer(
 getClass().getResourceAsStream("/media/audio/chapter3/baby.wav"),   "audio/x-wav");
```

A `Player` instance is created here using the `Manager` class's `createPlayer(InputStream is, String contentType)` method. Simpler still would have been to use the `createPlayer(String locator)` method, which would let the `Manager` class figure out the content type of the audio file. However, using this method would have required specifying a protocol to access the file. Because the file is stored in the MIDlet's JAR file, there is no protocol to access it directly. If the multimedia file had been stored on the Internet or the emulator/device's local file system, you could have used either `http://<url-to-file>` or `file:///<file-location>` (if the optional `FileConnection` API [JSR 75] is supported by the device), respectively. However, to access files over these protocols would require explicit permission from the user or a signed MIDlet requesting these permissions, and therefore, this example would not remain simple anymore.

Note Some devices support the `resource` protocol for accessing files stored in a JAR file. In such devices, you can use `resource://media/audio/chapter3/baby.wav` or similar to create a locator for a media file. However, because this support is not universal, use it only if you are confident that the target device(s) will understand it.

The second line of code,

```
player.start();
```

is where the `Player` instance gets to start playing the media file. A lot of things happen behind the scenes when you call this method, but effectively, the media file will play back as soon as possible when this method is called. The method returns as soon as the media has started playing. The `start()` method is also used to resume playback of a media file that was previously paused (or had stopped because it had reached its end), with playback starting at the point where the media was paused.

Both the lines of code described previously throw several exceptions. Although Listing 3-1 ignores the individual exceptions, you can catch them and deal with them individually. Table 3-1 shows a list of these exceptions and information on which method throws what exception.

Table 3-1. *Exceptions Thrown Playing a Media File Using MMAPI*

Exception	Thrown By	Description
`IllegalArgumentException`	`createPlayer(String InputStream, String type)` `createPlayer(String locator)` `createPlayer(DataSource source)`	Thrown if the `InputStream`, locator, or `DataSource` is null.
`MediaException`	`createPlayer()`—all three variants	Thrown if a `Player` instance cannot be created for the given stream and type, `DataSource`, or locator, possibly because the protocol and/or the format is not supported.
`IOException`	`createPlayer()`—all three variants	Thrown if connection to the source of the media file encounters a problem, such as network congestion, corrupt file, and so on.
`SecurityException`	`createPlayer()`—all three variants	Thrown if permission to create a `Player` instance has not been granted to the calling MIDlet.
`IllegalStateException`	`start()`	Thrown if the `Player` instance cannot be started because it has been closed. More about `Player` states appears in Chapter 4.

Continued

Table 3-1. *Continued*

Exception	Thrown By	Description
`MediaException`	`start()`	A catchall exception that is thrown if the `Player` instance can't be started because of any number of reasons.
`SecurityException`	`start()`	The calling MIDlet may have permission to create the `Player` instance but not to start it, in which case, it throws this exception.

After a `Player` instance has been started, it can be paused, closed, restarted, played repeatedly, and controlled by instances of the `Control` interface. A `Player` also fires information about various events that can be listened to and appropriate activities performed based on these events. Chapter 4 will cover handling `Player` events and the lifecycle of a `Player` instance. For now, let's create a slightly advanced `Player` instance that can be paused, restarted, played in a loop, and closed.

Creating a Functional Player

Listing 3-1 created a very basic `Player` instance. In this section, you'll learn to create a slightly more advanced and functional `Player` that allows you to select a media file to play from a list and gives you control over the volume.

Before we get into the actual code, let's look at Figure 3-1, which shows the finished MIDlet, called `AudioPlayer`, in operation using the Sun Java Wireless Toolkit 2.3 DefaultColorPhone emulator, the Motorola C975 emulator, and the Motorola C975 actual device.



Figure 3-1. *AudioMIDlet shows list of audio files on Sun's DefaultColorPhone emulator, Motorola C975 emulator, and Motorola C975 device.*

The MIDlet shows a list of three audio files that the user can select and play. Figure 3-2 shows the resulting screens when the user selects one of these files.

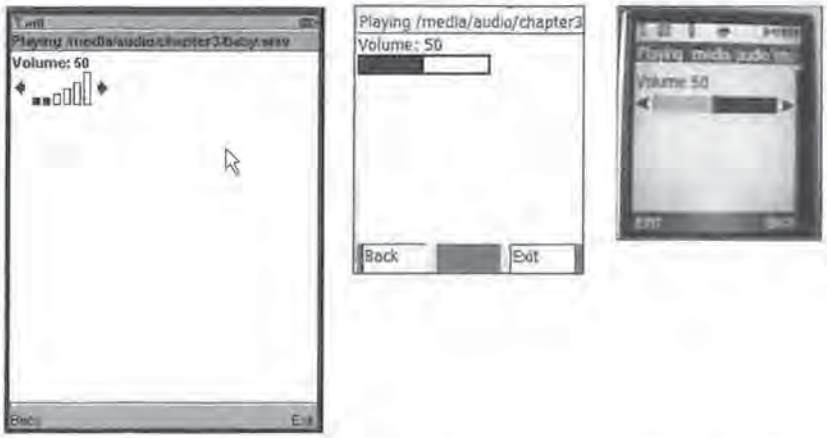


Figure 3-2. *Playing* `baby.wav` *in Sun's DefaultColorPhone emulator,* `applause.wav` *in Motorola C975 emulator, and* `laughter.wav` *in Motorola C975 device*

When a file is selected from the list, a new screen shows the actual file name and location, plays this file, and allows the user to control the volume of the playback. This volume control is rendered on the screen using a `Gauge` item. The user can use this gauge to increase or decrease the volume, and the actual volume level is reflected on the screen. The user can also choose to go back to the audio list or exit from the MIDlet. The audio file is played twice unless the user exits or goes back to the list before that.

Writing the Code

The code for this MIDlet is divided into two files: `AudioPlayer` and `AudioPlayerCanvas`. `AudioPlayerCanvas` is responsible for the canvas that the user sees when an audio file is being played and allows the user to increase or decrease the volume. More importantly, it manages the playing of the audio files and their lifecycle when they are paused or stopped. `AudioPlayerCanvas` is shown in Listing 3-2.

Listing 3-2. `AudioPlayerCanvas`*—an Interface for Playing Audio Files*

```
package com.apress.chapter3;

import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;

public class AudioPlayerCanvas implements ItemStateListener {

  // the parent MIDlet
  private AudioPlayer parent;
```

```
// form that contains canvas elements
private Form form;

// gauge to allow user to manipulate volume
private Gauge gauge;

// the volume control
private VolumeControl volume;

// the player used to play media
private Player player;

// is the player paused?
private boolean paused = false;

public AudioPlayerCanvas(AudioPlayer parent) {

  this.parent = parent;

  // create form and add elements and listeners
  form = new Form("");
  gauge = new Gauge("Volume: 50", true, 100, 50);
  form.append(gauge);
  form.addCommand(parent.exitCommand);
  form.addCommand(parent.backCommand);
  form.setCommandListener(parent);

  // a change in volume gauge will be handled by this class
  form.setItemStateListener(this);
}

public void playMedia(String locator) {

  try {

    // create the player for the specified string locator
    player = Manager.createPlayer(
      getClass().getResourceAsStream(locator), "audio/x-wav");

    // realize it
    player.realize();

    // get the volume control
    volume = (VolumeControl)player.getControl("VolumeControl");

    // initialize it to 50
    volume.setLevel(50);
```

```
        // initialize the gauge
        gauge.setValue(volume.getLevel());
        gauge.setLabel("Volume: " + volume.getLevel());

        // play it twice
        player.setLoopCount(2);

        // start the player
        player.start();

        // set the title of the form
        form.setTitle("Playing " + locator);

      } catch(Exception e) {
        e.printStackTrace();
      }

    }

    public void pauseMedia() {

      // if the player needs to be paused, either due to an incoming call,
      // or due to user actions
      if(player != null) {
        try {
          player.stop();
          paused = true;
        } catch(Exception e) {}
      }
    }

    public void restartMedia() {

      // restarting after player was paused
      if(player != null) {
        try {
          player.start();
          paused = false;
        } catch(Exception e) {}
      }
    }

    public boolean isPlayerPaused() {
      return paused;
    }

    public Form getForm() {
      return this.form;
```

```
    }

    public void itemStateChanged(Item item) {

      // there is only one item on the form, the gauge, and any change in its
      // value means the user wants to increase or decrease the playback volume
      volume.setLevel(gauge.getValue());
      gauge.setLabel("Volume: " + volume.getLevel());
    }

    public void cleanUp() {

      // clean up, either due to user action or AMS call
      if(player != null) {
        player.close();
        player = null;
      }
    }
}
```

AudioPlayerCanvas creates a Form that displays the volume control and allows the user to interact with it. It attaches a Gauge to this Form for the volume control and adds the back and exit commands to the Form. These commands are created in the parent, which is the main MIDlet class AudioPlayer, shown in Listing 3-3. However, the canvas takes care of any changes to the Gauge (and consequently to the volume of a Player instance) by implementing the itemStateChanged(Item item) method.

A Player instance is created when the parent MIDlet calls the playMedia(String locator) method on this canvas. This method attempts to create a Player instance by using the createPlayer(InputStream is, String contentType) method, guessing the content to be of type audio/x-wav. The next step after a successful player creation is the realization of the player, which is done by calling player.realize(). You'll learn more about realization in Chapter 4, but suffice to say at this point that after a Player instance has been realized, it can expose any controls that may be associated with it. Thus, you couldn't have accessed the VolumeControl associated with the player without actually realizing the player. This VolumeControl is used to initialize the Gauge besides allowing the user to control the volume in the itemStateChanged (Item item) method.

By calling player.setLoopCount(2), the player plays the audio file twice. By default, this value is set to 1, and if you wanted to play the file indefinitely, you could set this value to -1. Setting this value to 0 will result in an IllegalArgumentException. The loop count must be set before the start() method is called or rather when the Player instance is in a paused state; otherwise, an IllegalStateException will be thrown.

The Player instance is paused in the pauseMedia() method, which calls the stop() method to pause the media at the current media time. The next time the player is started, or rather, restarted, it will resume from the same media time. This occurs when the restartMedia() method is called, where the audio file's playback is restarted by a call to the start() method on the Player instance. Although this example does not allow the user to pause or restart the playback explicitly by way of commands, this behavior can be easily simulated by an incoming phone call when this MIDlet is being run.

The final noteworthy method is the `cleanUp()` method that closes each `Player` instance after it is no longer required. This method is called any time the user issues the `exit` or `back` commands. Calling `close()` on a `Player` instance releases any resources held by the player (including the audio file and the audio device, such as a speaker). After a `Player` instance has been closed, it can no longer be used.

In a nutshell, the `Player` instance is created when the `playMedia()` method is called with the location of the media file. This instance, if successfully created, is then realized, which allows it to expose the `VolumeControl`. This control is initialized and shown on a canvas to the user to interact with. Finally, the `Player` instance is started by calling the `start()` method. The `Player` instance is paused and restarted in the `pauseMedia()` and `restartMedia()` methods, respectively. The `cleanUp()` method closes the `Player` instance.

The actual MIDlet code is written in the `AudioPlayer` class. Listing 3-3 shows the code for this class.

Listing 3-3. `AudioPlayer` MIDlet *Creates* `AudioPlayerCanvas`

```
package com.apress.chapter3;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class AudioPlayer extends MIDlet implements CommandListener {

  // the list of media names
  private String[] audioDisplayList =
    {"Baby Crying", "Applause", "Laughter"};

  // the list of media locations
  private String[] audioList        =
    {"/media/audio/chapter3/baby.wav",
     "/media/audio/chapter3/applause.wav",
     "/media/audio/chapter3/laughter.wav"};

  protected Display display;
  private AudioPlayerCanvas canvas;
  private List list;

  protected Command exitCommand;
  protected Command backCommand;

  public AudioPlayer() {

    // initialize the list and add the exit command
    list = new List(
      "Pick an Audio file", List.IMPLICIT, audioDisplayList, null);
    exitCommand = new Command("Exit", Command.EXIT, 1);
    list.addCommand(exitCommand);
    list.setCommandListener(this);
```

```
    // the back command
    backCommand = new Command("Back", Command.BACK, 1);

    // create the canvas
    canvas = new AudioPlayerCanvas(this);

    // and initialize the display
    display = Display.getDisplay(this);
  }

  public void startApp() {

    // if startApp() is called after MIDlet has been paused
    if(canvas.isPlayerPaused()) {

      // restart the player
      canvas.restartMedia();
      display.setCurrent(canvas.getForm());
    } else {

      // else display the audio list
      display.setCurrent(list);
    }
  }

  public void pauseApp() {
    // pauses the playing of the player, if any
    canvas.pauseMedia();
  }

  public void destroyApp(boolean unconditional) {
    // cleans up and closes player, if any
    canvas.cleanUp();
  }

  public void commandAction(Command command, Displayable disp) {

    // exiting the MIDlet
    if(command == exitCommand) {
      canvas.cleanUp(); // clean up, if any
      notifyDestroyed(); // let AMS know clean up is done
      return;
    } else if(command == backCommand) { // back to the list
      canvas.cleanUp();
      display.setCurrent(list);
      return;
    }
```

```
        // the implicit list handling
        if(disp == list) {

          // play the current selected file
          canvas.playMedia(audioList[list.getSelectedIndex()]);

          // display the canvas's form
          display.setCurrent(canvas.getForm());
        }
      }
    }
```

The `AudioPlayer` constructor creates a `List` of audio files, adds the `exitCommand` to it, and initializes the current display for the MIDlet. It also creates the player canvas of Listing 3-2 (earlier in this chapter). When the MIDlet is started using the `startApp()` method, the method checks whether the player canvas was playing any media. If the player canvas was `startApp()` the method restarts the media. If nothing was being played, and presumably this would be the case the first time the MIDlet is run, the list of audio files is displayed for the user to select from.

After the user selects a file from the list, the implicit list handling takes over and selects the location of the actual audio file from the `String` array `audioList`. It then proceeds to call the `playMedia()` method on this location and sets the `Form` instance created by the canvas as the main display.

If the user issues either the `back` or `exit` commands, the canvas is told to clean up by calling the `cleanUp()` method before acting appropriately. The same is true when the Application Management Software (AMS) calls the `destroyApp()` method; cleanup is done before quitting by calling the `cleanUp()` method. When the MIDlet needs to be paused by the AMS by invoking the `pauseApp()` method, the `pauseMedia()` method is called.

Improving `AudioPlayerCanvas`: Caching `Player` Instances

One problem is obvious with the `AudioPlayerCanvas` class: It creates new instances of the `Player` interface each time a request to play an audio file is received, even if the audio file has been played before. This happens in the `playMedia()` method. Creating objects is an expensive task, especially multimedia objects such as `Player` instances. Performance can be improved by caching `Player` instances that do not need to change. For examples like this, where the media file and its location do not change and especially media stored in the MIDlet's JAR file, caching `Player` instances is an obvious choice.

However, caching `Player` instances is not an easy task. In Chapter 4, you'll learn about the various states of a `Player` instance. If an instance is cached, it must be cached so that the next time it is played, it starts in the right state. Also, when an instance is started, it acquires resources on the device. For example, audio files acquire access to the audio device (speakers, in-memory buffers, and so on) on a multimedia device. These resources must be released before the instance is cached so that other instances can acquire exclusive access.

Let's modify the `commandAction()` method from the `AudioPlayer` MIDlet in Listing 3-3 to find out the time taken to play one of the audio files (`baby.wav`). The following snippet shows the changes in bold.

```
long t1 = System.currentTimeMillis();

// play the current selected file
canvas.playMedia(audioList[list.getSelectedIndex()]);

long t2 = System.currentTimeMillis();

canvas.getForm().setTitle("Time taken: " + (t2-t1) + " ms");
```

The title of the canvas's form is changed to reflect the time taken to start playing a listing in milliseconds. Table 3-2 now shows the time taken to play the file `baby.wav` three times over the different devices: Sun's DefaultColorPhone emulator, Motorola C975 emulator, and the actual Motorola C975 device.

Table 3-2. *Time Taken to Start Playback of* `baby.wav` *Across Three Devices*

Device	1st Attempt	2nd Attempt	3rd Attempt
Sun's DefaultColorPhone	180ms	40ms	40ms
Motorola C975 emulator	100ms	10ms	10ms
Motorola C975 device	236ms	218ms	207ms

The values shown are reflective only and will be different on different machines and environments.

The fact that the emulators take a long time for the first attempt and much less time for the next two suggests that the emulators are performing some sort of caching themselves. The actual device doesn't seem to be doing any caching, and the times are pretty consistent for the device. However, the emulators aren't really performing any caching of `Player` instances, and the long startup delay suggests latency in acquiring audio device on the emulator's computer platform.

Let's now modify `AudioPlayerCanvas` to `CachedAudioPlayerCanvas` so that it caches `Player` instances and plays the cached instances when called upon (and creates them when called for the first time). Similarly, `AudioPlayer` will be modified to `CachingAudioPlayer` to use this new canvas class. Listing 3-4 shows `CachedAudioPlayerCanvas`.

Listing 3-4. `CachedAudioPlayerCanvas` *Caches* `Player` *Instances*

```
package com.apress.chapter3;

import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;

public class CachedAudioPlayerCanvas implements ItemStateListener {

  // the parent MIDlet
  private CachingAudioPlayer parent;
```

```
// form that contains canvas elements
private Form form;

// gauge to allow user to manipulate volume
private Gauge gauge;

// the volume control
private VolumeControl volume;

// the player used to play media
private Player player;

// is the player paused?
private boolean paused = false;

private Hashtable players;

public CachedAudioPlayerCanvas (CachingAudioPlayer parent) {

  this.parent = parent;

  // create form and add elements and listeners
  form = new Form("");
  gauge = new Gauge("Volume: 50", true, 100, 50);
  form.append(gauge);
  form.addCommand(parent.exitCommand);
  form.addCommand(parent.backCommand);
  form.setCommandListener(parent);

  // a change in volume gauge will be handled by this class
  form.setItemStateListener(this);

  players = new Hashtable();
}

public void playMedia(String locator) {

  try {

    // first look for an existing instance
    player = (Player)players.get(locator);

    if(player == null) {

      // create the player for the specified string locator
      player = Manager.createPlayer(
        getClass().getResourceAsStream(locator), "audio/x-wav");
```

```
        // fetch it
        player.prefetch();

        // put this instance in the Hashtable
        players.put(locator, player);
      }

      // get the volume control
      volume = (VolumeControl)player.getControl("VolumeControl");

      // initialize it to 50
      volume.setLevel(50);

      // initialize the gauge
      gauge.setValue(volume.getLevel());
      gauge.setLabel("Volume: " + volume.getLevel());

      // play it twice
      player.setLoopCount(2);

      // start the player
      player.start();

      // set the title of the form
      form.setTitle("Playing " + locator);

    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void pauseMedia() {

    // if the player needs to be paused, either due to an incoming call,
    // or due to user actions
    if(player != null) {
      try {
        player.stop();
        paused = true;
      } catch(Exception e) {}
    }
  }

  public void restartMedia() {

    // restarting after player was paused
```

```
    if(player != null) {
      try {
        player.start();
        paused = false;
      } catch(Exception e) {}
    }
  }

  public boolean isPlayerPaused() {
    return paused;
  }

  public Form getForm() {
    return this.form;
  }

  public void itemStateChanged(Item item) {

    // there is only one item on the form, the gauge, and any change in its
    // value means the user wants to increase or decrease the playback volume
    volume.setLevel(gauge.getValue());
    gauge.setLabel("Volume: " + volume.getLevel());
  }

  public void cleanUp() {

    // clean up, either due to user action or AMS call
    if(player != null) {

      try {
        player.setMediaTime(0);
      } catch(Exception e) {}

      player.deallocate();
      player = null;
    }
  }

  public void closeAll() {
    // iterate through the player instances and close all
    for(Enumeration e = players.elements(); e.hasMoreElements();) {
      Player p = (Player)e.nextElement();
      p.close();
    }
  }
}
```

The major changes between AudioPlayerCanvas and CachedAudioPlayerCanvas are shown in bold. A Hashtable is used to cache Player instances, and the locator String is used as the key. When a request to play a particular audio file is made, the Hashtable is searched and an instance is used if found. Note that a VolumeControl is still constructed fresh for each instance.

Instead of using the realize() method, prefetch() is used when creating a Player instance for the first time. The purpose of these methods and the distinction between them will be explained in Chapter 4. Note that using prefetch() instead of realize() allows quicker startup time for media playback.

Similarly, the cleanUp() method is modified to use deallocate() rather than close(). Further, the media time is set to 0 to allow the media to be played back from the starting point of the file each time it is started. This is done by using the method setMediaTime(), which is like a fast-forward and rewind method for moving through the media. Player instances that don't support seek operations like this will throw a MediaException. In this case, calling this method is really only necessary for the Sun emulator, which does not reset the media time on deallocation.

Finally, a new method called closeAll() is added to allow the CachingAudioPlayer MIDlet to shut down all the Player instances before it is itself shut down.

Listing 3-5 shows the modified CachingAudioPlayer. Once again, the changes from Listing 3-3 are highlighted in bold.

Listing 3-5. CachingAudioPlayer *Uses the New* CachedAudioPlayerCanvas

```
package com.apress.chapter3;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class CachingAudioPlayer extends MIDlet implements CommandListener {

  // the list of media names
  private String[] audioDisplayList =
    {"Baby Crying", "Applause", "Laughter"};

  // the list of media locations
  private String[] audioList        =
    {"/media/audio/chapter3/baby.wav",
     "/media/audio/chapter3/applause.wav",
     "/media/audio/chapter3/laughter.wav"};

  protected Display display;
  private CachedAudioPlayerCanvas canvas;
  private List list;

  protected Command exitCommand;
  protected Command backCommand;

  public CachingAudioPlayer() {
```

```
    // initialize the list and add the exit command
    list = new List(
      "Pick an Audio file", List.IMPLICIT, audioDisplayList, null);
    exitCommand = new Command("Exit", Command.EXIT, 1);
    list.addCommand(exitCommand);
    list.setCommandListener(this);

    // the back command
    backCommand = new Command("Back", Command.BACK, 1);

    // create the canvas
    canvas = new CachedAudioPlayerCanvas(this);

    // and initialize the display
    display = Display.getDisplay(this);
  }

  public void startApp() {

    // if startApp() is called after MIDlet has been paused
    if(canvas.isPlayerPaused()) {

      // restart the player
      canvas.restartMedia();
      display.setCurrent(canvas.getForm());
    } else {

      // else display the audio list
      display.setCurrent(list);
    }
  }

  public void pauseApp() {
    // pauses the playing of the player, if any
    canvas.pauseMedia();
  }

  public void destroyApp(boolean unconditional) {
    // closes all players before shutdown
    canvas.closeAll();
  }

  public void commandAction(Command command, Displayable disp) {
```

```
    // exiting the MIDlet
    if(command == exitCommand) {
      canvas.closeAll(); // close all players
      notifyDestroyed(); // let AMS know clean up is done
      return;
    } else if(command == backCommand) { // back to the list
      canvas.cleanUp();
      display.setCurrent(list);
      return;
    }

    // the implicit list handling
    if(disp == list) {

      long t1 = System.currentTimeMillis();

      // play the current selected file
      canvas.playMedia(audioList[list.getSelectedIndex()]);

      long t2 = System.currentTimeMillis();

      canvas.getForm().setTitle("Time taken: " + (t2-t1) + " ms");

      // display the canvas's form
      display.setCurrent(canvas.getForm());
    }
  }
}
```

Using the new version of these files, Table 3-3 shows the times taken to play `baby.wav` over the three different emulators/devices.

Table 3-3. *Time Taken to Play Back* `baby.wav` *with Caching of* `Player` *Instances*

Device	1st Attempt	2nd Attempt	3rd Attempt
Sun's DefaultColorPhone	180ms	30ms	30ms
Motorola C975 emulator	100ms	0ms	0ms
Motorola C975 device	202ms	14ms	14ms

Comparing Table 3-3 with Table 3-2 should show the obvious performance improvement that caching has brought about. The real gain is on the actual device, which is where it matters the most. The time to playback has been reduced to a consistent 14ms after an initial startup time of 202ms. The performance on the emulators is encouraging as well with reduction in playback time by about 10ms each over the noncached version.

Summary

Creating MMAPI-based multimedia MIDlets requires knowledge of not just MMAPI, but an understanding of the MIDlet creation process as well. The understanding required is nominal, as MMAPI MIDlets are no different from other MIDlets in terms of their architecture and process.

This chapter gave you a hands-on introduction to MMAPI by creating a simple media player. You enhanced the media player MIDlet by caching, and saw how the enhancement improved the performance of the original player.

The next chapter, on media player lifecycle and events, will explain what happens behind the scenes when the media player is used to play media files. You'll also see the various events that occur during a media player's lifecycle and how to respond and take advantage of this knowledge.

CHAPTER 4

Media Player Lifecycle and Events

As a MIDlet transitions between different states during its lifecycle, so does a `Player` instance that has a lifecycle of its own. A `Player` instance has many more states that it transitions between. These states are well defined, and transitions between them raise events that interested parties can listen to and respond accordingly.

In this chapter, you'll learn about these different states, the lifecycle of a `Player` instance, and how a `Player` instance transitions between these states. Finally, you'll learn how events generated during these transitions can be captured by interested listeners and acted upon.

Overview

A `Player` instance goes through five different states during its lifetime. The capability for an instance to go through these many states gives developers greater control over the working of an instance. These states are UNREALIZED, REALIZED, PREFETCHED, STARTED, and CLOSED.

A `Player` instance is guaranteed to go through all these states if started; that is, the instance is not just created but playback (or recording as the case may be) is initiated. Moving between states is not necessarily linear and can happen either due to programmatic control or some external or internal events. Movement from the CLOSED state to any other state is not possible.

Movement between different states results in events being fired for any listeners to act on. These events are delivered asynchronously and in the order that they are generated. The whole event delivery mechanism is extensible, which allows you to define application-specific events. Several system-level events are already defined that will satisfy most cases.

Exploring the Different `Player` States

MMAPI allows you to programmatically move between different states (except moving away from the CLOSED state). This gives you greater control over the way you manage the lifecycle of a `Player` instance, increases responsiveness, and allows manageability of these instances. This section explores these different states and the methods that allow you to gain this control.

In a nutshell, a `Player` instance starts life in the UNREALIZED state. It moves from this state to the REALIZED state when the user calls the `realize()` method. The REALIZED state

gives way to the PREFETCHED state when the `prefetch()` method is called. Calling `start()` moves the instance to the STARTED state, and calling `close()` leads to the CLOSED state. This simple transition path is displayed in Figure 4-1.

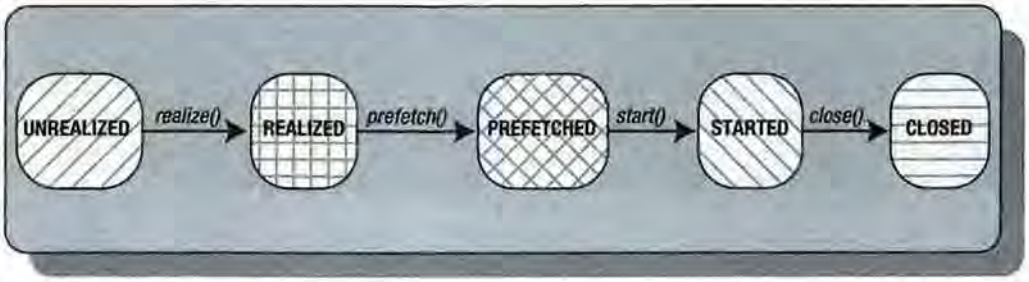


Figure 4-1. *A simple linear transition path for a* `Player` *instance*

Figure 4-1 shows the most *likely* transition path for a `Player` instance. Except for the CLOSED state, transitions can occur between the other states by calling special methods. These methods are covered shortly when the individual states are discussed.

Calling any of these methods to make a transition between different states is synchronous in nature. The methods don't return till the transition is complete. However, if any of these methods cannot make the transition, a `MediaException` is thrown to indicate so.

You can determine the current state of a `Player` instance by using the method `getState()`. This returns one of five constants defined in the `Player` interface corresponding to the five states shown in Figure 4-1. These constants are `UNREALIZED`, `REALIZED`, `PREFETCHED`, `STARTED`, and `CLOSED`.

Let's examine each of these states individually to see what they mean and how movement between them is not always so linear.

Note Most of these state transition methods are implemented in Java, as opposed to the code for parsing and decoding multimedia data, which is implemented in the native language of the device on which the MIDlet is running. Parsing and decoding are memory-intensive operations and implementing them in Java would sacrifice performance. Media transition methods, on the other hand, are not CPU-intensive and can be safely implemented in Java, as most of them are. Some parts of these methods may be implemented in native language to take advantage of device-specific performance features.

UNREALIZED

A `Player` instance starts life in an UNREALIZED state. When you use the `Manager` class to create a `Player` instance using any of the three `createPlayer()` methods, it creates a barely usable instance in the UNREALIZED state. An UNREALIZED `Player` is of no use because it doesn't have enough information to start functioning. It needs to acquire resources, such as audio and recording hardware on the device; set up in memory buffers for acquiring the media content; and communicate with the location of the media data. All these processes are performed in other states.

A Player instance moves away from the UNREALIZED state when the realize() method is called, which if successful, moves it to the REALIZED state. If unsuccessful, a MediaException is thrown, and the instance remains in the UNREALIZED state. If the realize() method blocks for a long time because it is a synchronous method, you can attempt to call the deallocate() method on the Player instance, which tries to keep the method in the UNREALIZED state. You'll learn more about this in the upcoming "REALIZED" section.

Not many actions can be performed on an UNREALIZED Player instance. For example, you cannot retrieve any controls from this instance using the getControl() or the getControls() methods, because the instance doesn't have enough information to generate the controls. You cannot change the playback time of the media, provided that it allows you to change the media time in the first place, with the setMediaTime() method. You cannot even retrieve or change the instance's TimeBase for synchronization using the getTimeBase() or setTimeBase() methods. A call to any of these methods in the UNREALIZED state results in an IllegalStateException.

The only useful operation that you can do with an UNREALIZED instance, besides realizing or closing it, is to set the number of times the instance should loop using the setLoopCount() method. You can also retrieve the likely duration of the media, which almost always returns -1, indicating an unknown time (TIME_UKNOWN constant in the Player interface).

The Player interface has a static integer constant to represent this state, UNREALIZED.

Figure 4-2 summarizes the UNREALIZED state and its transitions.

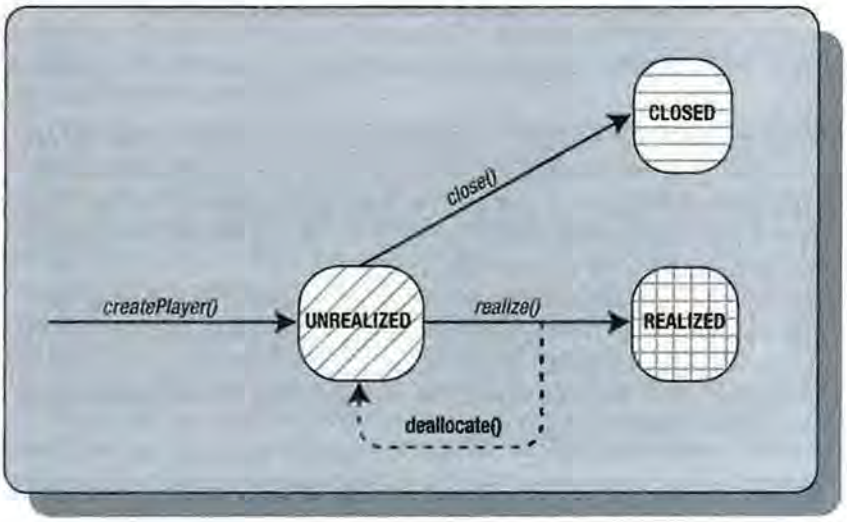


Figure 4-2. *UNREALIZED state transitions*

REALIZED

A Player instance moves from the UNREALIZED to the REALIZED state when the realize() method is called. The realize() method can be time consuming because it actually retrieves the media data. If the UNREALIZED state represents that the instance has connected to its media location, in a REALIZED state, it has in all probability retrieved this data (except for data that is streaming in nature).

After it has been REALIZED, a `Player` instance cannot go back to the UNREALIZED state. As you learned in the last section, a `realize()` method that is taking too long to return can be preempted by calling the `deallocate()` method, which keeps the instance in the UNREALIZED state. But from a REALIZED state itself, an instance can only go to the PREFETCHED or the CLOSED states. Figure 4-3 shows the REALIZED state transitions.

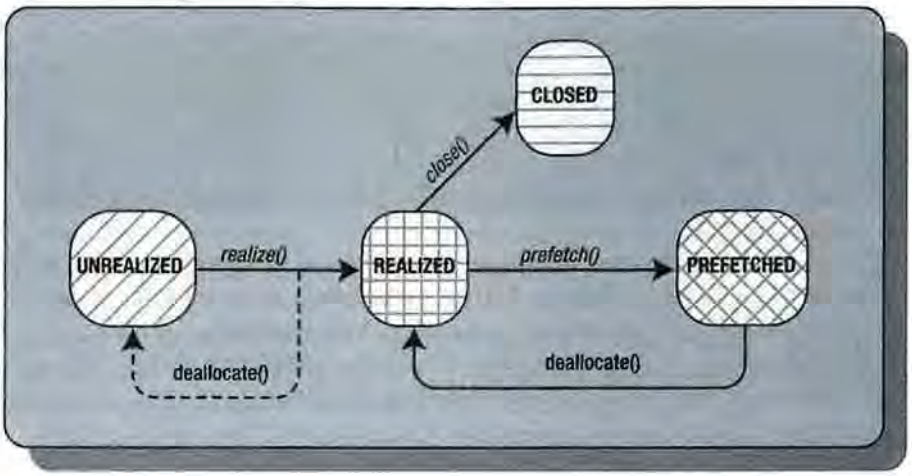


Figure 4-3. *REALIZED state transitions*

What actually happens within the `realize()` method is dependent on individual MMAPI implementations. However, after the method returns successfully, it is guaranteed that the underlying media data has been examined, and any available controls are up for use. Any resources required to play back the data that require exclusive use by your MIDlet on the device that it is running are also guaranteed *not* to have been acquired.

The `realize()` method throws, besides `IllegalStateException` and `MediaException`, a `SecurityException` as well. The `IllegalStateException` is thrown if the instance is in a CLOSED state; the `MediaException` is thrown if a media-specific error occurs; and the `SecurityException` is thrown if the MIDlet doesn't have permission to access the media file. This is closely related to Digital Rights Management (DRM) of digital data.

■ **Note** DRM is an industry term that refers to a service or technology to control access to digital data.

Because MMAPI is used to access some of the most popular digital data, such as music and video, it allows implementations to plug in a DRM technology to control this access. This allows very simple control over the media data. For example, `SecurityException` is thrown by

the realize() method if the DRM indicates that the right of the user to use the media data has expired (because it could only be used within a specific timeframe, it could only be played once, or for any other DRM-based reason). The realize() method implementation calls the DRM technology built in to the device to make this call.

Ideally, a call like this happens when the Player instance is created. DRM kicks in when createPlayer() is called on a Manager class, and you'll receive a SecurityException for trying to create an instance on an expired or inaccessible content. (You'll also receive a SecurityException for protocol-specific restrictions built in to the user's device, such as accessing the network when no permission to access the network has been given.) However, you might want to replay content without needing to create another Player instance on the same media data (as you saw in Chapter 3 when you cached Player instances). In those cases, realize() checks to make sure that DRM rules haven't been breached; if they have been, realize() throws a SecurityException.

The Player interface has a static integer constant that represents this state, REALIZED.

PREFETCHED

In the PREFETCHED state, a Player instance is in the best possible state to get started with the playback (or recording in case of a player for capturing data) of media. The instance has decoded the data and acquired access to any exclusive resources required for playback (or recording). A Player instance that hasn't been prefetched cannot be started.

Player instances move into the PREFETCHED state when the user calls the prefetch() method on a REALIZED instance. Calling deallocate() does the reverse, that is, moves a PREFETCHED instance to the REALIZED state, thereby releasing any exclusive resources acquired to move into the PREFETCHED state.

Note that calling prefetch() doesn't necessarily mean that all the media data would have been decoded and ready for playback (or recording). It just means that most of the processing has been done and the media can be played back (or recorded) with the minimum possible latency. To accentuate this point, if an instance is already in the PREFETCHED state, and you use the prefetch() method, the instance will try and minimize the latency even further. However, the reduction in startup times is not guaranteed, and different implementations of MMAPI will differ in what they exactly do if prefetch() is called twice. If you do call prefetch() on an already PREFETCHED instance, it is guaranteed not to throw any errors.

An IllegalStateException is thrown if you call this method on a CLOSED instance. A MediaException is also thrown if an error occurs when processing or decoding the media data. However, this same exception is thrown if an exclusive resource cannot be acquired. For example, if a Player instance requires exclusive access to the audio hardware on a device, and this is not available because it is being used by some other instance, a MediaException will be thrown to indicate this. Although there's no way to differentiate between the two reasons for MediaException, in the case of the latter reason, you can call prefetch() again and if exclusive access is now possible, the instance will move to the PREFETCHED state.

Similar to the realize() method, calling the prefetch() method may throw a SecurityException if the Player instance has insufficient permissions to either decode media data or acquire exclusive resources.

Figure 4-4 shows the possible state transitions in the PREFETCHED state.

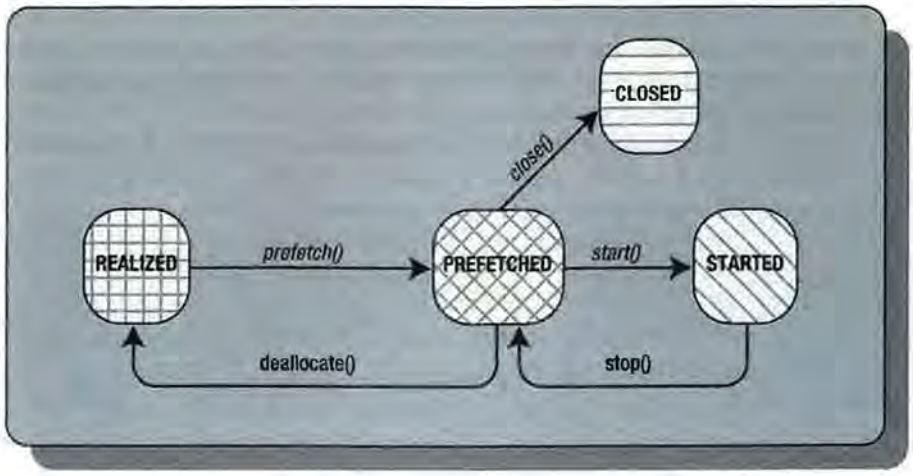


Figure 4-4. *PREFETCHED state transitions*

As you can see, there is no direct transition path for an `UNREALIZED` instance to go to the PREFETCHED state. This is why, if you call `prefetch()` on an `UNREALIZED` instance, it implies a call to the `realize()` method first. However, a transition from the PREFETCHED state to the REALIZED state can occur by using the `deallocate()` method. By doing so, you allow your instance to give up the exclusive resources acquired by your instance for other instances (or other MIDlets, applications, or AMS) to use.

An instance can arrive in the PREFETCHED state from the STARTED state as well. This transition can occur in several ways as explained in detail in the next section.

The `Player` interface has a static integer constant that represents this state, `PREFETCHED`.

STARTED

A `Player` instance in the STARTED state is playing back (or recording, streaming, and so on) actual media. This is the most useful state that an instance can be in. This state is achieved by calling the `start()` method on a `PREFETCHED` instance. However, note that calling the `start()` method doesn't guarantee that the instance will actually move immediately into the STARTED state. By calling the `start()` method, you are telling the instance to move into the STARTED state as soon as possible. Only when the `start()` method returns successfully is the instance considered to be in this state.

Of course, the instance may not be able to successfully move into the STARTED state when this method is called. Besides a `SecurityException`, thrown if there is not enough permission to start the media, a `MediaException` may be thrown if an error occurs when processing the media for playback (or recording). When any of these exceptions is thrown, the instance remains in the PREFETCHED state. As expected, an `IllegalStateException` is thrown if you try calling `start()` on an instance that is in the CLOSED state.

STARTED is the only state that has an automatic transition based on the state of the media playback (or recording). If you call `start()` on a `Player` instance, it automatically moves to the PREFETCHED state if the end of media playback is reached; that is, there is nothing left to play. STARTED also automatically moves to the PREFETCHED state if a preset stop time is reached. Preset stop times are set using the `StopTimeControl`.

Media that has a very short playback time moves to the PREFETCHED state almost immediately. For example, consider that you are trying to play an audio file and want to initiate some action when its `Player` instance is in the STARTED state. When you call the `start()` method, the instance temporarily moves to this state, but before you can react to the STARTED event, the playback would be over for a very short audio file, and the instance would have moved back to the PREFETCHED state. Thus, there are no guarantees for successfully acting on a STARTED instance because of this automatic transition.

Besides these automatic transitions, a `Player` instance also moves back to the PREFETCHED state when the `stop()` method is called in the STARTED state and the method returns successfully. The effect of the `stop()` method is to pause the instance at the current media time. (Note that there is *no* corresponding STOPPED state for the STARTED state. When stopped, an instance is in the PREFETCHED state.) Similar to the `start()` method, the `stop()` method also throws the `IllegalStateException` and `MediaException`. The first exception is thrown if this method is called on a `CLOSED` instance, whereas the second is thrown if the instance cannot be stopped.

If you call the `start()` method again, after you have stopped a previously started instance, it resumes at the media time that it was stopped at, effectively restarting paused media. This can, of course, be overridden by using the method `setMediaTime()`, which allows you to restart the playback from whenever you want it to. As expected, this will not work for media that is being recorded, and may or may not be supported for streaming data. In cases where this is not supported, calling `setMediaTime()` will throw a `MediaException`.

After a `Player` instance is in the STARTED state, calling either `setTimeBase()` or `setLoopCount()`throws an `IllegalStateException`. This makes sense. A `Player`'s `TimeBase` allows it to synchronize itself with other instances via the internal clock. After the instance has already started, this clock will be out of sync if changed midstream. Similarly, changing the number of times the instance must play back in the STARTED state will cause confusion over this count.

If you call the `start()` method on an `UNREALIZED` instance, it implies a call to the `realize()` and `prefetch()` methods, in that order. If you call it on a `REALIZED` instance, it implies a call to the `prefetch()` method first. In short, no direct transition occurs between the UNREALIZED state and the STARTED state on the one hand, and the REALIZED and STARTED state on the other, with the `start()` method taking care of these transitions for you. This is why listings in Chapter 3 were able to get away with calling the `start()` method only. However, calling the `start()` method on a `PREFETCHED` instance is always better than calling on `UNREALIZED` or `REALIZED`, because it reduces the overall startup time. You should only call the `start()` method after you've brought the instance to the PREFETCHED state, rather than letting the `start()` method bring it to that state.

Figure 4-5 shows the state transitions for the STARTED state.

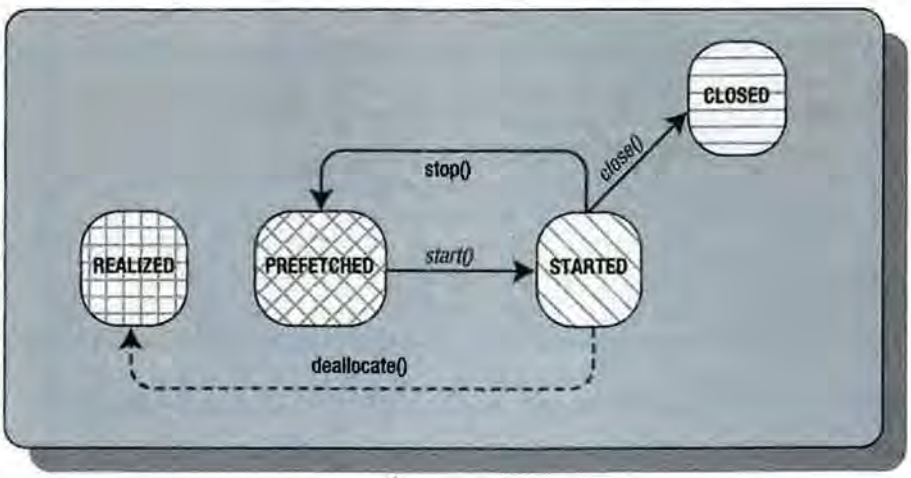


Figure 4-5. *STARTED state transitions*

You can call the `deallocate()` method on a `STARTED` instance, which internally implies a call to the `stop()` method first, and thus, the state of the instance transitions from STARTED to PREFETCHED to REALIZED, if both methods return successfully.

The `Player` interface has a static integer constant that represents this state, `STARTED`.

CLOSED

A `Player` instance in the CLOSED state is no longer usable. No methods can be called on it in this state with the exception of the `getState()` method. All other states transition to this state when the `close()` method is called, but there are no automatic transitions. If you call `close()` in the CLOSED state, no exception is thrown and the call is ignored. In fact, this method does not throw any exceptions; if any errors occur, the method returns silently and moves the instance to the CLOSED state anyway. All resources held by the instance are released, including any connections, exclusive device resources, and internal buffers.

■ **Note** Although the MMAPI specification does not say so, calling `close()` during different states causes different actions to be performed. Because the specification is silent on this issue, different implementations implement the actions in their own way. However, most implementations try to call the `deallocate()` method before performing any closeup actions. Recall that the `deallocate()` method can be called in all states (except, of course, the CLOSED state, whereas in the REALIZED state, the `deallocate()` call is ignored). Calling the `deallocate()` method in the STARTED state causes the `stop()` method to be called first. Thus, in all probability, if you call `close()` on a `STARTED Player` instance, it will go through the following cleanup methods: `close()` ➤ `stop()` ➤ `deallocate()`.

The `Player` interface has a static integer constant that represents this state, `CLOSED`.

Responding to `Player` Events

Each state transition and many other events generate a regular stream of notifications for any listener objects interested in a `Player` instance. This event delivery mechanism is implemented using an asynchronous model that is similar to most Java event delivery mechanisms.

The key class in this mechanism is the `PlayerListener` interface. Any class may implement this interface, register this implementation with the target `Player` instance, and start receiving notifications as the instance goes through its lifecycle. Several events are defined within this interface that cover a comprehensive list of `Player` events. You can create your own events as well and listen and react to them.

The `PlayerListener` interface defines only one method that implementations must implement. This method—`playerUpdate(Player player, String event, Object eventData)`—is invoked when an event takes place. To register an implementation with a `Player` instance, you use the method `addPlayerListener(PlayerListener listener)`; to remove the instance, you use `removePlayerListener(PlayerListener listener)`. Multiple listeners can be attached to a single instance, and multiple instances can send their events to the same listener. Because the `playerUpdate()` method receives the instance as an argument, it knows how to differentiate between the different instances.

As a simple example, let's modify Listing 3-1 from Chapter 3 to receive notifications from the simple `Player` instance that was created in that listing. The new code will display the events on the device screen as they are received. Listing 4-1 shows this modified code in the MIDlet called `EchoEventsMIDlet`.

Listing 4-1. `EchoEventsMIDlet` *Echoes Player Events Onscreen*

```
package com.apress.chapter4;

import javax.microedition.media.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class EchoEventsMIDlet extends MIDlet implements PlayerListener {

  private StringItem stringItem;

  public void startApp() {

    try {

      Form form = new Form("Player State");
      stringItem = new StringItem("", null)
      form.append(stringItem);
      Display.getDisplay(this).setCurrent(form);

      Player player = Manager.createPlayer(
        getClass().getResourceAsStream(
        "/media/audio/chapter4/baby.wav"), "audio/x-wav");
      player.addPlayerListener(this);
```

```
            player.start();

        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void playerUpdate(Player player, String event, Object eventData) {
        stringItem.setText(event);
        System.err.println(event);
    }
}
```

The EchoEventsMIDlet acts as the listener for the Player instance that it creates by implementing PlayerListener and adding the playerUpdate() method. When created, the Player instance registers this MIDlet by using the method addPlayerListener(this). Any event that is now generated by the corresponding event is delivered to the playerUpdate() method.

The other change in this MIDlet from Listing 3-1 is to create a Form object with a single StringItem on it to promptly display onscreen and print the current event received by the playerUpdate() method to the error output stream.

Running this MIDlet returns mostly consistent results because the Sun Java Wireless Toolkit's DefaultColorPhone emulator fires an extra volumeChanged event.

Note You may not notice the volumeChanged event on the DefaultColorPhone emulator because it may happen too fast depending on the time it takes for the media file to enter the STARTED state. Check the error output stream. The volumeChanged event is fired when the Player instance enters the PREFETCHED state (on the Sun MMAPI implementation).

The Motorola emulator and actual C975 device do not fire this volumeChanged event. The events that are fired and received by all three environments are the started and the endofmedia events. As you may guess, the started event represents when an instance has entered the STARTED state. The endofmedia event is delivered when no more media is left to play (or record/stream). This event is delivered each time an instance that is set to loop reaches the end of the media for each loop.

Of course, many more events than the two (or three if you consider the DefaultColorPhone) are fired by this simple example. Table 4-1 lists all the events defined in the MMAPI specification in the `PlayerListener` interface defined as constants.

Table 4-1. *A Complete List of* `Player` *Events Defined in the* `PlayerListener` *Interface*

Player Event Constant	Constant Value	When Fired
`BUFFERING_STARTED`	`bufferingStarted`	When an instance has started buffering media data for processing or playback.
`BUFFERING_STOPPED`	`bufferingStopped`	When an instance has exited the buffering stage.
`CLOSED`	`closed`	When an instance is closed.
`DEVICE_AVAILABLE`	`deviceAvailable`	When a system resource required by a `Player` instance becomes available for use.
`DEVICE_UNAVAILABLE`	`deviceUnavailable`	When a system resource required by a `Player` instance becomes unavailable. This event must precede the previous event.
`DURATION_UPDATED`	`durationUpdated`	When the duration of previously unknown media data becomes available.
`END_OF_MEDIA`	`endOfMedia`	When an instance has reached the end of the media during the current loop.
`ERROR`	`error`	When an error, which is usually fatal, occurs.
`RECORD_ERROR`	`recordError`	When an error occurs during recording (audio or video).
`RECORD_STARTED`	`recordStarted`	When recording of media data has started.
`RECORD_STOPPED`	`recordStopped`	When recording of media data has stopped.
`SIZE_CHANGED`	`sizeChanged`	When the size of a video display has changed for whatever reason.
`STARTED`	`started`	When the instance has entered the STARTED state.
`STOPPED`	`stopped`	When the instance has paused due to the `stop()` method being called.
`STOPPED_AT_TIME`	`stoppedAtTime`	When the instance has paused due to the `StopTimeControl`'s `setStopTime()` method.
`VOLUME_CHANGED`	`volumeChanged`	When the volume of an audio device is changed.

If you look at the signature of the `playerUpdate()` method, you'll see that it takes three parameters. The first is the `Player` instance that has thrown the event, the second is the actual event, and the third is the `eventData` as an `Object`. The `eventData` is interesting because it contains specific information about each event that can help you do something when the particular event is fired. For example, when the `stopped` event is received by this method, the `eventData`

is a Long object identifying the media time when the corresponding Player instance is stopped. Similarly, the started event's eventData contains the media time when the media is started. Almost each event carries some useful information in the corresponding eventData; Table 4-2 shows the complete list.

Table 4-2. *Events and Corresponding Event Data*

Event	Event Data
BUFFERING_STARTED	A Long object designating the time when buffering has started.
BUFFERING_STOPPED	A Long object designating the time when buffering has stopped.
CLOSED	Event data is null when this event is fired.
DEVICE_AVAILABLE	A String object that is the name of the device that is now available.
DEVICE_UNAVAILABLE	A String object that is the name of the device that is not available.
DURATION_UPDATED	A Long object designating the new duration of the media.
END_OF_MEDIA	A Long object that contains the media time when the Player instance reached the end of media and stopped.
ERROR	A String that contains the error message.
RECORD_ERROR	A String that contains the error message.
RECORD_STARTED	A Long object that designates the media time when recording has started.
RECORD_STOPPED	A Long object that designates the media time when recording has stopped.
SIZE_CHANGED	A VideoControl control object that contains information about the new size.
STARTED	A Long object designating the media time when the Player instance is started.
STOPPED	A Long object designating the media time when the Player instance is stopped (paused).
STOPPED_AT_TIME	Similar to STOPPED, the eventData contains the media time when the Player instance is stopped in the form of a Long object.
VOLUME_CHANGED	A VolumeControl control object that contains information about the new volume.

Note that in MMAPI all times are measured in microseconds, not milliseconds.

Understanding the Event Delivery Mechanism

The event delivery mechanism in MMAPI is based on an asynchronous model that allows you to create multimedia applications that do not block the main application thread. This means that events are fired using an event delivery thread separate from the main application thread. This thread may or may not be in existence till an actual event is to be delivered. For example, in the Sun's MMAPI reference implementation, this thread is only created when the first event is fired. Even then, this thread remains active only for another five seconds, after which, if no more events are delivered, the thread exits. A new thread is created the next time an event needs to be delivered. Most actual commercial implementations follow a similar model that only differs in the time that they stay alive for; however, they are guaranteed to all follow this asynchronous nature of event delivery.

The MMAPI also guarantees that events will be delivered to their respective listeners in the order they are generated. This way, events that occur very fast after one another are guaranteed to be received by the registered listeners in order, without getting overwhelmed by newer events. For example, suppose you start playing a media file, which would fire a STARTED event. However, if the media file is short, it will end very quickly and generate an END_OF_MEDIA event almost immediately after it sends the STARTED event. The listener is guaranteed to receive the STARTED event before the END_OF_MEDIA event even if it occurs nearly simultaneously.

Of course, if an error occurs at any stage during a Player instance creation or usage so that the instance cannot continue working, the event delivery mechanism sends an ERROR event. The receipt of this event implies that the instance is unusable and is in a CLOSED state.

Creating an Event Handling Class

In Chapter 3, you created a MIDlet that allowed you to select a media audio file from a list to play and control its volume. In this section, you'll create an event handling class and attach it to the functional Player instances created in that MIDlet. This event handling class is basic, but it gives you an idea of how to listen for events, handle them accordingly, and use the event and eventData parameters. Listing 4-2 shows this event handling class, called EventHandler.

Listing 4-2. EventHandler *Is the Listener for Functional* Player *Instances Created in the Previous Chapter*

```
package com.apress.chapter4;

import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.lcdui.StringItem;

public class EventHandler implements PlayerListener {

  private StringItem item;

  public EventHandler(StringItem item) {
    this.item = item;
  }

  public void playerUpdate(Player player, String event, Object eventData) {

    if(event.equals(PlayerListener.VOLUME_CHANGED)) {

      // a player's volume has been changed
      VolumeControl vc = (VolumeControl)eventData;
      updateDisplay("Volume Changed to: " + vc.getLevel());

      if(vc.getLevel() > 60)  {
        updateDisplay("Volume higher than 60 is too loud");
        vc.setLevel(60);
      }
    } else if(event.equals(PlayerListener.STOPPED)) {
```

```
            // player instance paused
            updateDisplay("Player paused at: " + (Long)eventData);
        } else if(event.equals(PlayerListener.STARTED)) {

            // player instance started (or restarted)
            updateDisplay("Player started at: " + (Long)eventData);
        } else if(event.equals(PlayerListener.END_OF_MEDIA)) {

            // player instace reached end of loop
            updateDisplay("Player reached end of loop.");
        } else if(event.equals(PlayerListener.CLOSED)) {

            // player instance closed
            updateDisplay("Player closed.");
        } else if(event.equals(PlayerListener.ERROR)) {

            // if an error occurs, eventData contains the error message
            updateDisplay("Error Message: " + (String)eventData);
        }
    }

    public void updateDisplay(String text) {

        // update the item on the screen
        item.setText(text);

        // and write to error stream as well
        System.err.println(text);
    }

}
```

The `EventHandler` constructor accepts a `StringItem` screen item to which it can write updates as it receives events. The `playerUpdate()` method is where the updates are written to the screen based on the event that has occurred.

If you change the volume of a `Player` instance, the associated `VolumeControl` is retrieved from the `eventData` after casting it appropriately. You can then query the new volume level from this control.

Note You can, of course, retrieve the same `VolumeControl` by querying the associated `Player` instance with the `getControl(" VolumeControl ")` method call that returns a reference to the same instance as referenced by the `eventData` parameter. The direct referencing `eventData` omits a method call, whereas `getControl()` method omits the use of a cast.

Here, the handler informs the user that volume over 60 is too loud and resets the volume back to 60. Note that resetting the volume in turn generates another VOLUME_CHANGED event!

The rest of the events are handled accordingly, and you can use the associated event data with appropriate casts. The updateDisplay() method updates the screen as well as writes message to the error output stream because some of the messages on the screen will happen too quickly.

The event generating Player instances now need to be told to send the instances to this handling class. This is done in the CachingAudioPlayerCanvas class where these instances are first created. This class is now modified to add a StringItem to display the messages from the event handler, create the EventHandler class, and set each Player instance up with this class as the listener. These changes are shown in bold in Listing 4-3.

Listing 4-3. *Enabling Event Handling in the* CachingAudioPlayerCanvas *Class*

```
package com.apress.chapter4;

import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;

public class CachedAudioPlayerCanvas implements ItemStateListener {

  // the parent MIDlet
  private CachingAudioPlayer parent;

  // form that contains canvas elements
  private Form form;

  // gauge to allow user to manipulate volume
  private Gauge gauge;

  // the volume control
  private VolumeControl volume;

  // the player used to play media
  private Player player;

  // is the player paused?
  private boolean paused = false;

  // to display event info
  private StringItem eventInfo;

  // the event handler
  private EventHandler handler;

  private Hashtable players;
```

```
public CachedAudioPlayerCanvas(CachingAudioPlayer parent) {

  this.parent = parent;

  // create form and add elements and listeners
  form = new Form("");
  gauge = new Gauge("Volume: 50", true, 100, 50);
  eventInfo = new StringItem("", null);
  form.append(gauge);

  // add the event info string item
  form.append(eventInfo);

  // create the EventHandler
  handler = new EventHandler(eventInfo);

  form.addCommand(parent.exitCommand);
  form.addCommand(parent.backCommand);
  form.setCommandListener(parent);

  // a change in volume gauge will be handled by this class
  form.setItemStateListener(this);

  players = new Hashtable();
}

public void playMedia(String locator) {

  try {

    // first look for an existing instance
    player = (Player)players.get(locator);

    if(player == null) {

      // create the player for the specified string locator
      player = Manager.createPlayer(
        getClass().getResourceAsStream(locator), "audio/x-wav");

      // add the EventHandler as a listener
      player.addPlayerListener(handler);

      // fetch it
      player.prefetch();

      // put this instance in the Hashtable
      players.put(locator, player);
```

```
        }

        // get the volume control
        volume = (VolumeControl)player.getControl("VolumeControl");

        // initialize it to 50
        volume.setLevel(50);

        // initialize the gauge
        gauge.setValue(volume.getLevel());
        gauge.setLabel("Volume: " + volume.getLevel());

        // play it twice
        player.setLoopCount(2);

        // start the player
        player.start();

        // set the title of the form
        form.setTitle("Playing " + locator);

      } catch(Exception e) {
        e.printStackTrace();
      }
    }

... rest of the code omitted as it doesn't change from Listing 3-4 ...
```

A single `EventHandler` instance is used for all three `Player` instances that are created. Because `playerUpdate()` receives the instance that generated the event, it's easy to distinguish between these instances, if necessary.

Handling a Custom Event

As you may realize by now, there is no special event class in the MMAPI. That is, events are distinguished as `String` constants in the `PlayerListener` interface. To create, rather to handle, a custom event, you do not need to extend any other class.

Custom event creation is primarily designed for MMAPI implementations. This means that the MMAPI specification, having designed its own mandatory events, makes it open for MMAPI implementations to create and broadcast their own events. The description of these events would be made clear in the documentation for each implementation, and the events are likely to be named in the reverse domain name convention. For example, the MMAPI reference implementation from Sun defines a custom event called `com.sun.midi.lyrics`, which is a Sun-specific event for karaoke lyrics.

Although the `PlayerListener` interface provides for events that are most common, in some special cases, you may want to define your own. For example, let's say you wanted to do something special that requires an event to be raised whenever an audio file has been played

halfway through. None of the predefined events will satisfy this requirement, so you'll need to raise and handle your own custom event. But how do you actually create and raise an event?

The short answer is that you can't. Unless you are ready to implement your own version of a `Player` instance that handles the type of media that you are after. This is not an easy task and requires you to handle all the steps required in realizing, prefetching, and decoding, not to mention interfacing with the controls that it exposes. Further, you have to use your own version over the version supplied with the MMAPI implementation that you are working with. After you have accomplished these difficult tasks, you may be able to plug in and raise your own event.

Handling custom events is, as you may expect, much easier. You only need to know the name of the event and the type of `eventData` that it exposes to be able to use it in the `playerUpdate()` method. Thus, the following code fragment will catch the `com.sun.midi.lyrics` event, and the event data exposed will be a `byte` array:

```
if(event.equals("com.sun.midi.lyrics")) {
  byte[] data = (byte[])eventData;
}
```

Note that the MMAPI specification states that to catch standard events in the `playerUpdate()` method, you should use the reference equality check, and for custom events, you should use the object equality check. Thus, `(event == PlayerListener.CLOSED)` should be preferred over `event.equals(PlayerListener.CLOSED)`, and `event.equals("com.sun.midi.lyrics")` must be used for custom events. Standard events are automatically interned because they are constants; therefore, using the reference check will be faster than the object equality check. However, the same cannot be guaranteed for custom events, so you must always use the object equality test. The `EventHandler` in Listing 4-2 used the object equality test and is now converted to use the reference check in Listing 4-4 to make it more responsive.

Listing 4-4. *Converting* `EventHandler` *to Use Reference Checking Instead of Object Equality*

```
package com.apress.chapter4;

import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.lcdui.StringItem;

public class EventHandler implements PlayerListener {

  private StringItem item;

  public EventHandler(StringItem item) {
    this.item = item;
  }

  public void playerUpdate(Player player, String event, Object eventData) {
```

```
    if(event == (PlayerListener.VOLUME_CHANGED)) {

      // a player's volume has been changed
      VolumeControl vc = (VolumeControl)eventData;
      updateDisplay("Volume Changed to: " + vc.getLevel());

      if(vc.getLevel() > 60)  {
        updateDisplay("Volume higher than 60 is too loud");
        vc.setLevel(60);
      }
    } else if(event == (PlayerListener.STOPPED)) {

      // player instance paused
      updateDisplay("Player paused at: " + (Long)eventData);
    } else if(event == (PlayerListener.STARTED)) {

      // player instance started (or restarted)
      updateDisplay("Player started at: " + (Long)eventData);
    } else if(event == (PlayerListener.END_OF_MEDIA)) {

      // player instance reached end of loop
      updateDisplay("Player reached end of loop.");
    } else if(event == (PlayerListener.CLOSED)) {

      // player instance closed
      updateDisplay("Player closed.");
    } else if(event == (PlayerListener.ERROR)) {

      // if an error occurs, eventData contains the error message
      updateDisplay("Error Message: " + (String)eventData);
    }
  }

  public void updateDisplay(String text) {

    // update the item on the screen
    item.setText(text);

    // and write to error stream as well
    System.err.println(text);
  }
```

```
}
```

Due to device fragmentation, not all MMAPI implementations support reference check for events. Instead, you have to use `equals()` for comparison from Listing 4-2, instead of the improved code from Listing 4-4. The trick is to test your target device(s) for what is supported and optimize accordingly.

Summary

The several different states that a `Player` instance goes through in processing and playing media data allows developers to gain control over these states, provide feedback, and process events at these stages. These states are UNREALIZED, REALIZED, PREFETCHED, STARTED, and CLOSED, and the transitions between them are well defined and accessible.

In this chapter, you learned the background behind these states, the how and why of the transitions that take place between them, and how to respond to the various events generated during these transitions. You learned to create an event handling class and also how to listen to custom events.

The next chapter will introduce you to accessing media data over the network using MMAPI, a task that must be handled efficiently and cleanly for responsive multimedia MIDlets.

CHAPTER 5

Accessing Media Over the Network

The greatest advantage of applications built on mobile devices is that they can be run anywhere any time. These applications are truly mobile lifestyle choices because they allow us to extend our lifestyles by bringing these applications wherever we go, whether for fun or business. Applications that keep all data on mobile phones have the added advantage of being complete without requiring network access to function or update. However, truly mobile applications take advantage of available networks to provide an enhanced user experience.

Take the case of accessing multimedia files over a network. Although games and other fun applications can work with small embedded sound and other audio files, for an even, all-encompassing experience, the user requires network connectivity for his application to stream media over the network. A good reason for streaming over storing media files locally is the size of these files, video especially. Also streaming over stored media is used to provide updated files and to maintain digital rights over the provided content.

Whatever the reasons, accessing multimedia data over a network reliably is an important aspect of learning to use MMAPI. Although the issues involved are not unique to accessing multimedia data per se, they do need to be taken care of.

In this chapter, you'll learn about these common issues and how they affect multimedia data access over the network using MMAPI. You'll learn how to overcome these issues to create MIDlets that can access the network reliably and consistently. Finally, you'll see examples of these issues and the solutions to them.

Understanding Threads in Java ME

The threading environment in Java ME is slightly different from the normal environment in Java SE. The JVM in Java ME does not create a separate application thread for your MIDlet, which is contrary to what happens in an application being run on Java SE, where an application gets its own application thread separate from the system thread. All processing in a MIDlet takes place inside the main system thread. Thus, methods that take a long time to complete hog this single thread and create a bottleneck for a functional MIDlet.

Any threads that you create yourself within your MIDlet are, of course, separate from the system thread and therefore not restrictive. These threads can do their own thing independent of the main system thread. Of course, you have to strike a fine balance between the number of

threads required versus the number of threads that the limited resources operating system can handle. Thread creation and management is a processor- and memory-hungry task and should be carefully used, especially in the Java ME environment, which runs on constrained devices.

To understand this better, take a look at Listing 5.1. When started, the MIDlet in this listing displays the name of the current system thread, which is the main MIDlet thread. When the user clicks the command for a new thread, the MIDlet creates a new thread, which displays an alert momentarily with its own name. It then simulates a long activity by sleeping for 4 seconds.

Listing 5-1. *Understanding Threads in MIDlets*

```
package com.apress.chapter5;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class ThreadTest extends MIDlet implements CommandListener {

  private Form form;
  private StringItem text;
  private Display display;

  public ThreadTest() {
    form = new Form("Thread Test");

    // displays the name of the current system thread
    text = new StringItem(
      "Current Thread:",
      Thread.currentThread().getName());

    form.append(text);

    // commands to exit and create new threads
    Command exit = new Command("Exit", Command.EXIT, 1);
    Command newThread = new Command("New Thread", Command.SCREEN, 1);

    form.addCommand(exit);
    form.addCommand(newThread);

    form.setCommandListener(this);

    display = Display.getDisplay(this);
  }

  public void startApp() {
    display.setCurrent(form);
  }
```

```
  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command cmd, Displayable disp) {
    if(cmd.getLabel().equals("Exit")) {
      notifyDestroyed();
    } else {
      System.err.print("Starting new thread ... ");

      // create a new thread
      Thread runner = new Thread(new ThreadRunner(display));

      // and start it
      runner.start();

      System.err.println("Done");
    }
  }
}

class ThreadRunner implements Runnable {

  // the parent MIDlet's display
  Display display;

  ThreadRunner(Display display) {
    this.display = display;
  }

  public void run() {

    System.err.print(
      "New thread ( " + Thread.currentThread().getName() + " ) running .. ");

    display.setCurrent(
      new Alert(Thread.currentThread().getName()));

    try {
      Thread. sleep(3000);
    } catch(Exception e) {}

    System.err.println("Done");
  }
}
```

Caution The `getName()` used in this listing for the `Thread` class is a CLDC 1.1 method. This listing will not work in CLDC 1.0 devices.

Figure 5-1 shows the MIDlet running in the Motorola emulator across a few actions, along with the output.

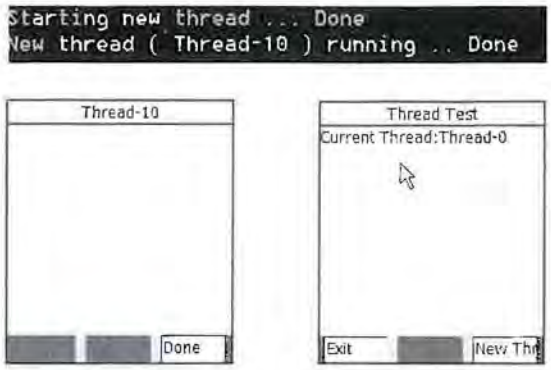


Figure 5-1. *Screen and console output for the* `ThreadTest` *MIDlet on Motorola emulator*

The MIDlet identifies the way threads work in the Java ME environment. The simulation of a long activity inside a new thread doesn't lock up the main MIDlet thread waiting for this activity to finish. The main MIDlet thread continues its processing after creating and starting the new thread. This concept is further established in Figure 5-2, which shows the main system thread and a new thread's activity and actions.

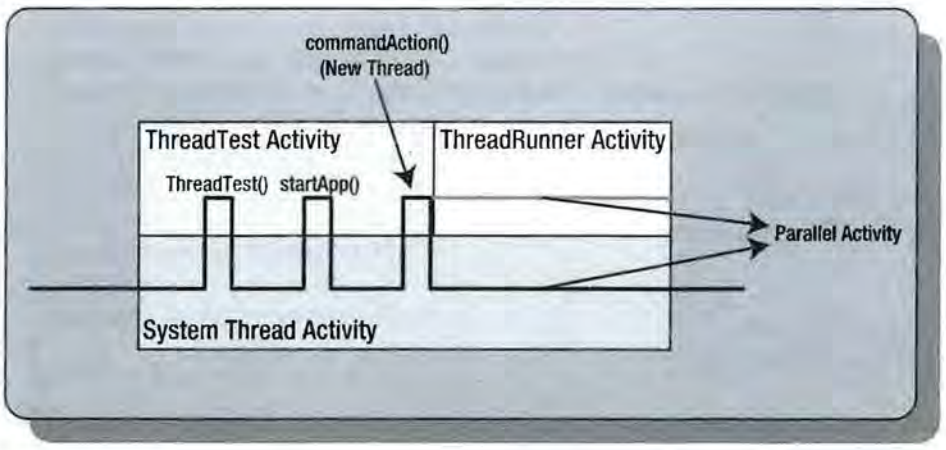


Figure 5-2. *System and new thread activity*

At this stage, you may be wondering if an understanding of thread activity in MIDlets has anything to do with using the MMAPI over a network. It does because network access of multimedia files (and all network activity) should only be done in a separate thread, distinct from the system thread. This is to make sure that network access of these files doesn't leave the main application unresponsive to the user. This is discussed further later in this chapter.

Understanding Permissions for Network Access

Whereas creating responsive MMAPI MIDlets with separate threads is one aspect of accessing network based media, another issue relates to permissions for media access over the network. MIDlets don't automatically have permissions to grab files over the network, so this must be explicitly or implicitly configured.

MIDlets run in a protection domain, which provides access to individual permissions based on whether or not the user grants (or denies) these permissions for the current MIDlet session or for the life of the MIDlet. Individual permissions must be requested explicitly, but not from within the MIDlet code. The permissions that are interesting for MMAPI development are `javax.microedition.io.Connector.http` and `javax.microedition.io.Connector.https`. This section focuses on these permissions, which allow retrieval of media files over the Web.

When any attempt is made to retrieve media files over the network, for example, by specifying an HTTP locator for the `createPlayer()` method, the MIDlet's protection domain kicks in. Because `javax.microedition.io.Connector.http` is a restricted permission, the user of the MIDlet must decide whether to grant access to the network.

As an example, consider the MIDlet in Listing 5-2, which wants to make a connection to a remote server to retrieve and play an audio file.

Listing 5-2. *Playing an Audio File Over the Network*

```
package com.apress.chapter5;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;

public class NetworkTest extends MIDlet implements CommandListener {

  private List list;
  private StringItem text;
  private Display display;

  public NetworkTest() {

    list = new List("Press Play", List.IMPLICIT);
```

```
    // commands to exit and play
    Command exit = new Command("Exit", Command.EXIT, 1);
    Command play = new Command("Play", Command.SCREEN, 1);

    list.addCommand(exit);
    list.addCommand(play);

    list.setCommandListener(this);
    display = Display.getDisplay(this);
  }

  public void startApp() {
    display.setCurrent(list);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command cmd, Displayable disp) {
    if(cmd.getLabel().equals("Exit")) {
      notifyDestroyed();
    } else {
      try {
        Player player =
          Manager.createPlayer(
        "http://www.mmapibook.com/resources/media/audio/chapter5/siren.wav");
        player.start();
      } catch(Exception e) { System.err.println(e); }
    }
  }
}
```

The bold code is where a `Player` instance is created by making a network connection for file retrieval. When you run this listing, the MIDlet needs to be granted permission by the user. Figure 5-3 shows how this permission is granted in the three different environments (Sun Java ME emulator, Motorola C975 emulator, and the Motorola C975 device, respectively).

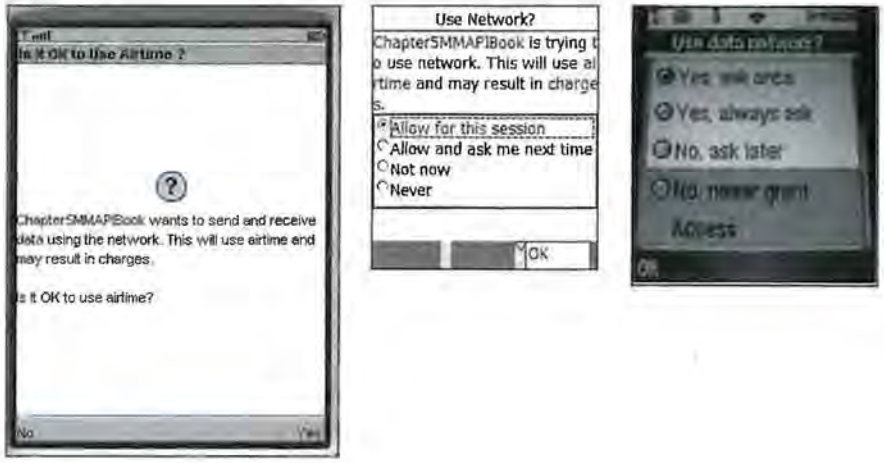


Figure 5-3. *Granting network access permission*

Most users who trust your MIDlet will want to select the Yes, Ask Once choice (or similar), so they won't be bothered the next time they run your MIDlet. By default, all MIDlets are run in an untrusted protection domain, where the user is always consulted for granting network access permission rights. You'll need to establish a level of trust for the device to run your network accessing MIDlet without requiring interaction from the user.

Trust is established by signing your MIDlet and putting it in a trusted domain. The complete details of how you do that is beyond the scope of this book, but this section outlines the steps required to get the `NetworkTest` MIDlet working without needing to request permission to access the network from the user.

To start with, you'll need to modify the JAD manifest file for this MIDlet and add an entry to request permission for HTTP access, as shown here:

```
MIDlet-Permissions: javax.microedition.io.Connector.http
```

If the connection to access the file will be over a secure connection, you need to request HTTPS access. If you're not sure, you can request both permissions.

```
MIDlet-Permissions:
  javax.microedition.io.Connector.http, javax.microedition.io.Connector.https
```

Note The preceding code lines should not be broken into two lines during coding; they are broken here due to space concerns.

Now when this MIDlet is being installed on a device, the AMS will fail the installation if it's unable to provide these permissions to the MIDlet. This means that if AMS is unable to put this MIDlet in a trusted domain, it halts the installation.

Of course, simply requesting permissions in the JAD file is only half the story. Any rogue MIDlet could request the permission this way and cause trouble. You still need to establish some sort of trust principle between the MIDlet and the AMS; the AMS will then trust your MIDlet enough to let it be installed in a trusted domain. This is done by signing the MIDlet with a certificate that can be validated up to a root certificate installed on the device.

To complete this test, at least on the Sun Java ME emulator, sign the `NetworkTest` MIDlet with the `trustedkey` key installed with the emulator. You access it by selecting Project ➤ Sign in the emulator as shown in Figure 5-4. This `trustedkey` is shown in Figure 5-5.

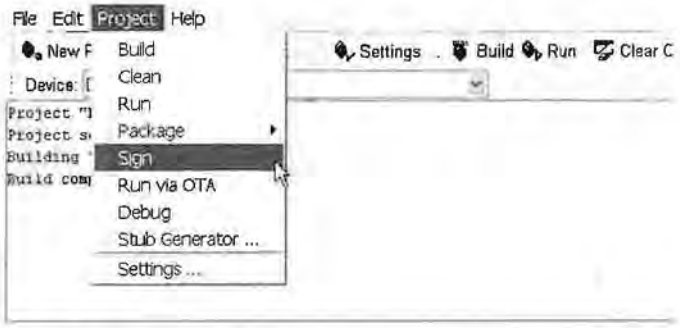


Figure 5-4. *Signing the* `NetworkTest` *MIDlet in the Sun Java ME emulator*



Figure 5-5. *The* `trustedkey` *used to sign the* `NetworkTest` *MIDlet*

After the MIDlet is signed, you can test it by using the Run via OTA menu option shown in Figure 5-6. You don't use the Run menu option like you have been doing so far because when you run the MIDlet locally (by selecting the Run menu option), it does not get installed into a protection domain. Only when installed via the Run via OTA option does your MIDlet gets a chance to be installed into a requested domain, which is the domain that you request by specifying permission for it in the JAD file.

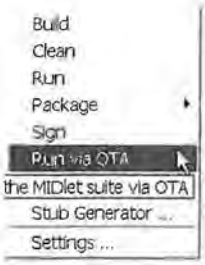


Figure 5-6. *Running* `NetworkTest` *MIDlet via Over the Air provisioning*

Of course, so far, you've only tested in the Sun Java ME emulator platform. Testing on the Motorola C975 emulator and device is only possible if you have a certificate containing a key-pair that can be trusted to a root certificate installed on the emulator and device, respectively. Even then, the actual device may only keep your MIDlet in a third-party trust domain that will never be fully trusted. To get it to a higher trust level, you need to sign it with a certificate supplied by either a manufacturer (Motorola) or an operator (for example, Vodafone). An alternate mechanism is to use the Java Verified Program, which certifies your application for deployment on devices that support this program. See more details of this program at *http://www.javaverified.com*.

To summarize, using MMAPI across a network requires looking at two issues:

- Network operations can be time consuming and should be done in a separate thread from the main application thread.
- Network access requires permission from the user. To gain this permission and then acquire a level of trust, your MIDlet must request this permission implicitly and be signed with a certificate that can be traced to a root certificate. Alternatively, explicit permission from the user is required when network access is attempted, which the user may not give.

You just learned about the second item in the preceding list; the next section covers the first issue in the list in more detail, expanding on what you've learned so far.

Putting It Together

The MIDlets created in the previous chapters have accessed their media files through the JAR file in which they are bundled. In this section, you'll create a `Player` instance that will access a media file over the network via HTTP. You'll create a separate thread from the main MIDlet thread to handle the task of creating and realizing the `Player` instance.

Listing 5-3 shows the code for the `ThreadedMIDlet` that initiates the playback. When requested by the user, it creates a new thread using the `NetworkPlayerManager` class, shown in Listing 5-4.

Listing 5-3. `ThreadedMIDlet` *Creates a New Thread to Play Back a Media File*

```
package com.apress.chapter5;

import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;
import javax.microedition.media.*;

public class ThreadedMIDlet extends MIDlet implements CommandListener {

  private List list;
  private StringItem text;
  private Display display;

  private NetworkPlayerManager mgr;
  private Command cancel = new Command("Cancel", Command.CANCEL, 1);

  public ThreadedMIDlet() {

    list = new List("Press Play", List.IMPLICIT);

    // commands to exit and play
    Command exit = new Command("Exit", Command.EXIT, 1);
    Command play = new Command("Play", Command.SCREEN, 1);

    list.addCommand(exit);
    list.addCommand(play);

    list.setCommandListener(this);
    display = Display.getDisplay(this);
  }

  public void startApp() {
    display.setCurrent(list);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void commandAction(Command cmd, Displayable disp) {
    if(cmd.getLabel().equals("Exit")) {
      notifyDestroyed();
    } else if(cmd.getLabel().equals("Cancel")) {
      mgr.cancel();
      display.setCurrent(list);
    } else {
      try {
        mgr = new NetworkPlayerManager(display, cancel, this);
        Thread runner = new Thread(mgr);
        runner.start();
```

```
      } catch(Exception e) { System.err.println(e); }
    }
  }
}
```

As you can see, Listing 5-3 is built on top of Listing 5-2. The changes are marked in bold. A new command is added to allow users to cancel a request to play the media file, in case it takes too long or the user changes his mind. The code to handle this command is added in the `commandAction()` method. More importantly, the default handling of the play command is now changed. A new class, `NetworkPlayerManager`, is created, and because it implements the `Runnable` interface, it can be started as a thread. Before going forward, let's take a look at this class in Listing 5-4.

Listing 5-4. `NetworkPlayerManager` *Class Implements the* `Runnable` *Interface and Makes Network Connections for Media Playback*

```
package com.apress.chapter5;

import javax.microedition.lcdui.*;
import javax.microedition.media.*;

public class NetworkPlayerManager implements Runnable {

  private Display display;
  private Form form;
  private StringItem msg;

  private boolean cancel = false;

  private Player player = null;

  public NetworkPlayerManager(
    Display display, Command cancelCmd, ThreadedMIDlet parent) {

    this.display = display;

    form = new Form("Network Player Manager");
    msg = new StringItem("Please Wait ... ", null);
    form.append(msg);
    form.addCommand(cancelCmd);

    form.setCommandListener(parent);
  }

  public void run() {

    display.setCurrent(form);
    boolean connected = false;
```

```
    try {
      player =
        Manager.createPlayer(
          "http://www.mmapibook.com/resources/media/audio/chapter5/siren.wav");

      player.realize();

      connected = true;
    } catch (Exception e) {
      msg.setText("Failed: " + e.getMessage());
      System.err.println(e);
      return;
    }

    if(connected && !cancel)
      msg.setText("Connected. Starting playback...");
    else {
      msg.setText("Unable to connect.");
      return;
    }

    try {
      player.start();
    } catch(Exception pe) {
      msg.setText(pe.getMessage());
      System.err.println(pe);
    }
  }

  public void cancel() {
    cancel = true;
    if(player!= null) player.deallocate();
  }
}
```

When a new instance of the `NetworkPlayerManager` class is created, it creates a `Form` instance and appends a `StringItem` for displaying messages. The cancel command is attached to this form, but the command listener for this command is still the calling MIDlet.

When the thread is started, the `run()` method takes over. It makes the `form` instance as the current display and tries to realize the `Player` instance for playback of the media file. If connection can be established, the message on the screen is updated and the playback is initiated. If, for some reason, the connection cannot be established, an error message is displayed and the display returns to the calling MIDlet.

Connection may not be established for a variety of reasons. The device may not allow your MIDlet access to the network if it isn't signed and in a correct domain, as you learned in the previous section. A path to the remote server may not be available.

While the MIDlet is trying to connect to the remote server, the user must be given a chance to cancel the request. The user can do this by issuing the cancel command. When this happens, the `cancel()` method is called, which deallocates the `Player` instance. Recall from Chapter 4 that calling `deallocate()` on an instance that is blocked at the realize call results in the instance being returned to the UNREALIZED state. Thus, if the network is intermittent or taking too long, the user can cancel the request. If the `Player` instance has already reached the REALIZED state, calling cancel will not affect anything.

If you sign this MIDlet and put it in the correct domain, you will not be asked permission to access the network when you run it. Figure 5-7 shows running a MIDlet over the Sun Java ME and the Motorola C975 emulators.

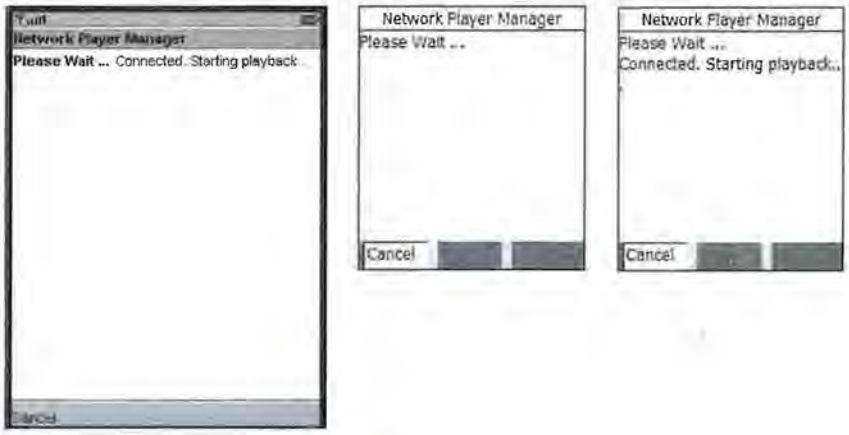


Figure 5-7. *Network access of a media file using a separate thread over Sun's DefaultColorPhone and Motorola C975 emulators*

Summary

Accessing media data over the network requires careful consideration because it involves two issues. First, network access should be done in a separate thread from the main thread, because accessing the network may block the main thread for a long time, leading to an unresponsive MIDlet. Second, network access requires permission from the user of the MIDlet as it may lead to charges from the network operator.

This chapter showed you how to overcome these two issues by explaining threads in Java ME and how they affect network access. The chapter also explained the concept of protection domains and how to overcome them for requesting permissions for network access. The chapter finished with an all-encompassing example that solved those two issues together.

The next chapter covers generating and playing tones using MMAPI with the help of the `ToneControl` control.

CHAPTER 6

Creating and Playing Tones Using ToneControl

The MMAPI is unique in providing a genuine capability to play synthetic tones on mobile devices. We all recognize the sometimes annoying, but practical, ring tones on most mobile phones. With MMAPI, you can program your own tones to build a ring tone that plays when an event occurs, to embed in a game as a musical accompaniment to the main action, or just for plain fun.

In this chapter, you'll learn how to use the built-in tone generator mechanism in MMAPI to generate and distribute synthetic tones. You'll start with the single note generator provided by the `Manager` class. You'll then learn about the `ToneControl` class, which provides mechanisms to generate complex tones based on user-defined sequences. Finally, you'll learn how to distribute these tone sequences using the JTS file structure.

First, however, you need to understand the basics of synthetic tones. Let's start with some background on this topic.

Understanding Synthetic Tones

Synthetic tones, as the name suggests, are tones generated synthetically, or more specifically, by a computer system. A ring tone is a dominant example of synthetic tones, but computer systems have been capable of generating these tones for quite some time. The ring tone industry has popularized the enormous push for this technology to enhance capabilities of mobile phones, resulting in first, monophonic, and now, polyphonic ring tones.

A synthetic tone is a musical note that has a defined duration. Monophonic ring tones only play one note of music at a time. Before the start of the next note, the previous note is cut off. Older synthesizers and almost all mobile phones and devices can play monophonic tones.

Note In music, a *note* is the unit of fixed pitch that has been given a name, or the graphic representation of that pitch in a notation system. In English, notes are given seven names: A, B, C, D, E, F, and G (in order of rising pitch).

Comparatively, polyphonic ring tones can play multiple tones at the same time, thereby providing the effect of multiple instruments playing together. Not only does it resemble a more convincing musical background, but also it's pleasing to the ears. Most modern mobile devices are capable of playing polyphonic tones.

As far as MMAPI is concerned, a single tone is specified using the note, duration, and volume. These three must be specified for the underlying engine to be able to play the tone. MMAPI provides the capability to play either a single tone using the static method `Manager.playTone(int note, int duration, int volume)` or more complex tone sequences (ring tones are nothing but sequences of tones) using the `ToneControl` interface. You'll see examples of both of these shortly.

Understanding Note, Pitch, and Frequency

To better understand how MMAPI plays tones, you must differentiate between different notes of music by understanding how notes relate to pitches and how their representation is used in MMAPI.

As stated earlier, in English seven notes are represented by the symbols A, B, C, D, E, F, and G. Each of these represents an increasing frequency of music. However, frequency is a physical term, as far as human understanding of music is concerned, and therefore, it is replaced with the more generic term of pitch. Thus, musically, pitch is the perception of the frequency of a note, and the symbols A to G represent notes in order of rising pitch. Symbol A represents a lower pitched musical note than symbol B, which is lower than symbol C, and so on.

Mathematically, each symbol is related to a frequency value, because each symbol represents a pitch. Table 6-1 shows the frequency value of each symbol.

Table 6-1. *Frequency Values for the Basic Notes*

Note Symbol	Frequency (in Hertz (Hz))
A	440
B	493.92
C	523.28
D	587.36
E	659.28
F	698.48
G	784

Of course, things are not this simple. There are a large number of variations on the basic symbols, so let's clarify some of the more important ones.

Even though notes are represented using only these seven symbols, the symbols are repetitive for frequencies that are an octave above or below them. This is better explained with an example.

Note An *octave* is the interval between one musical note and another with half or double the frequency.

Consider the symbol A. Although it represents the frequency 440, it also represents the frequencies 220 (= 440 / 2) and 880 (= 440 * 2). Further, it represents the frequencies 110 (= 220 / 2), 55 (= 110 / 2), and 27.5 (= 55 / 2) as well. All these frequencies are an octave interval apart because they are double or half of the frequency before or after them, respectively. Thus, A is a musical note that represents the frequencies 27.5, 55, 110, 220, 440, and 880. To distinguish between the different octaves, a number is placed after the symbol. A4 represents the fourth octave for the musical note A, which is 440 Hz. Table 6-2 summarizes some of these symbols.

Table 6-2. *The Different Octaves for the Musical Note A*

Symbol	Frequency (Hz)
A0	27.5
A1	55
A2	110
A3	220
A4	440
A5	880

The other seven symbols are similarly modified to arrive at different corresponding octaves. As an example, Table 6-3 shows some of the different octaves for the musical note C.

Table 6-3. *The Different Octaves for the Musical Note C*

Symbol	Frequency (Hz)
C0	16.35
C1	32.70
C2	65.41
C3	130.82
C4	261.64
C5	523.28

When no number is placed after a musical symbol, it usually means the fourth octave. Thus, A implies the frequency at 440 Hz (A4), and C implies the frequency at 261.64 (C4).

Besides octaves, the notes can differ by a semitone. If an octave difference in frequency is double or half of each frequency, a semitone is a musical interval that is one-twelfth of an octave. Alternatively, an octave contains 12 semitones.

Note A *semitone* is one-twelfth of an octave and is also known as half step. It is the smallest musical interval.

Thus, if a musical note differs from another by a semitone, it is modified into either a *sharp* or a *flat*. A sharp raises the pitch of a note by a semitone interval, whereas a flat lowers the pitch

by a semitone interval. Using musical notation, a sharp is denoted by # while b is used to denote a flat. Thus, $C^{\#}$ (pronounced C-sharp) denotes the musical note that is a semitone above C_4 and equals 277.183 Hz. This symbol may also be written as $C^{\#}_4$, but $C^{\#}$ is more common.

The figure of 277.183 for a pitch that is a semitone above C4 (261.64 Hz) comes from the fact that a semitone is one-twelfth of an octave, and an octave above a frequency means doubling over the previous frequency. This implies that a semitone above (or sharp as it is called) a frequency means multiplying it by the twelfth root of 2. The twelfth root of 2 equals 1.059463. Therefore, 261.64 * 1.059463 = 277.183.

Similarly, a flat pitch implies dividing the frequency by the twelfth root of 2. Therefore, $C^{\flat}_4$ equals 261.64 / 1.059463 = 246.95.

Given any note, you can thus arrive at the note higher or lower than that by multiplying or dividing it by the constant 1.059463.

Using a MMAPI Formula to Calculate Note Values

With this brief interlude into musical mathematics complete, let's return to the tone support in MMAPI. To play a single tone, you can use the `Manager` class static method `playTone(int note, int duration, int volume)`. The second and third parameters are self explanatory, but the first one needs an explanation. With the background in mind from the previous section, the first parameter cannot be the frequency of a note because it is only an `int` value, whereas frequencies can have a decimal part.

The Javadoc for this method defines that the `note` parameter can be a value between `0` and `127`, both inclusive. It also provides a formula to arrive at these values as well, based on the frequency of the desired note to be played as a tone by this method. This method is shown here:

```
note = ln(freq/8.176)*(17.31234049066755)
```

The value `17.31234049066755` is called the `SEMITONE_CONST` and is arrived at with the formula:

```
SEMITONE_CONST = 1/(ln(2^(1/12)))
```

Note that `ln` in both these formulas means log to the base 'e,' also called the natural logarithm of a number (as opposed to the common logarithm, which is log to the base 10).

Using this formula, let's calculate the corresponding `int note` parameter value for the note represented by $C^{\#}$. As you learned in the last section, the frequency of $C^{\#}$ is equal to 277.183.

```
note (C#) = ln(277.183/8.176) * (17.31234049066755) = 61 (after rounding)
```

To play C# using MMAPI, you use the value `61`. Listing 6-1 shows a MIDlet that will play this value when run.

Listing 6-1. *Playing C# for 5 Seconds at Max Volume*

```
package com.apress.chapter6;

import javax.microedition.midlet.MIDlet;
import javax.microedition.media.Manager;
import javax.microedition.media.MediaException;

public class CSharpMIDlet extends MIDlet {
  public CSharpMIDlet() {
  }
```

```
  public void startApp() {
    try {
      // plays C# (frequency of 277.183 for 5 seconds at max volume)
      Manager.playTone(61, 5000, 100);
    } catch(MediaException me) { System.err.println(me); }
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

}
```

The outcome on all three of our test environments will be the same sound being played for a maximum of 5 seconds at the loudest permissible volume. Not terribly exciting, but it demonstrates the basic way of using this method to play tones.

You don't have to manually calculate the integer value of a note given its frequency. Listing 6-2 shows a simple Java class that does this for you using the formula provided earlier. The class also contains a method for doing the reverse; that is, given the integer value of a note, it will tell you the corresponding frequency of the note/tone. Finally, this Java class contains a method that will print all the 128 notes along with their corresponding frequencies.

Caution Listing 6-2 cannot be run in a Java ME environment because Java ME doesn't contain the methods `ln(double val)` and `exp(double val)` in the `Math` class. Therefore, this method must be run in a Java SE environment.

Listing 6-2. *A Simple Java Class to Calculate Note Frequencies and* `int` *Values*

```
package com.apress.chapter6;

public class NoteCalculator {

  public static final double SEMITONE_CONST = 17.31234049066755;
  public static final double ARBIT_CONST = 8.176;

  public static int getNoteAsInt(double freq) {
    return (int)Math.round((Math.log(freq/ARBIT_CONST) * SEMITONE_CONST));
  }

  public static double getNoteFreq(int note) {
    return (ARBIT_CONST * Math.exp(note/SEMITONE_CONST));
  }
```

```
  public static void printAllNoteFreq() {
    for(int i = 0; i < 128; i++) {
      System.err.println(i + " : " + getNoteFreq(i));
    }
  }

  public static void main(String[] args) {

    try {
      if(args.length == 2) {
        if(args[0].equals("1")) {
          double freq = new Double(args[1]).doubleValue();
          System.err.println(args[1] + " Hz : " + getNoteAsInt(freq));
        } else {
          int note = new Integer(args[1]).intValue();
          System.err.println(args[1] + ": " + getNoteFreq(note));
        }
      } else {
        System.err.println("Usage: NoteCalculator convType convVal \r\n" +
          " where convType: 1 for freq in Hz and 2 for note value in int \r\n" +
          " and convVal is the actual value");
      }
    } catch(Exception e) {
      System.err.println(e);
    }

    // print all notes from 0 to 127
    printAllNoteFreq();
  }
}
```

Using the methods of this class, Table 6-4 lists the integer note values for the basic notes defined in Table 6-1.

Table 6-4. *Integer Note Values for the Basic Notes*

Basic Note	Integer Note Value
A	69
B	71
C	72
D	74
E	76
F	77
G	79

Using the playTone() Method

playTone() is a nonblocking method, which means that after it is called, it returns immediately. Your MIDlet continues to play the tone for the time you specify with the duration parameter. Listing 6-3 shows an example MIDlet that will play all the notes from 0 to 127 sequentially.

Listing 6-3. *Playing All Notes Sequentially*

```
package com.apress.chapter6;

import javax.microedition.midlet.MIDlet;
import javax.microedition.media.*;
import javax.microedition.lcdui.*;

public class AllTonesPlayer extends MIDlet implements CommandListener {

  Form displayForm = new Form("Playing all tones");
  StringItem info = new StringItem("", "");
  Command exit = new Command("Exit", Command.EXIT, 1);
  Thread runner;
  boolean stop = false;

  public void startApp() {

    displayForm.append(info);
    Display.getDisplay(this).setCurrent(displayForm);
    displayForm.addCommand(exit);
    displayForm.setCommandListener(this);

    // create and start a new thread to play all the notes
    runner = new Thread(new TonePlayer(info));
    runner.start();
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    if(runner != null) stop = true;

  }
```

```
  public void commandAction(Command cmd, Displayable disp) {
    // only exit command defined in this MIDlet
    destroyApp(true);
    notifyDestroyed();
  }

}

// Plays all notes sequentially
class TonePlayer implements Runnable {

  StringItem info;
  AllTonesPlayer midlet;

  public TonePlayer(StringItem info, AllTonesPlayer midlet) {
    this.info = info;
    this.midlet = midlet;
  }

  public void run() {
    try {
      for(int i = 0; i < 128; i++) {
        // wait a second before playing the next note
        Thread.sleep(1000);
        info.setText("Playing: " + i);
        Manager.playTone(i, 500, 100); // play for 500 milliseconds at max vol
        if(midlet.stop) break;
      }

    } catch(Exception me) { System.err.println(me); }
  }
}
```

The MIDlet starts playing the tones as soon as it's started. The `playTone()` method is called sequentially after every 1 second to play the note for 500 milliseconds. The note generation part is separated from the main MIDlet system thread into its own thread using the `TonePlayer` class to allow easy termination of this thread, and consequently, the tone generation sequence from the main system thread. Figure 6-1 shows the output on the Sun Java ME DefaultColorPhone emulator.



Figure 6-1. *Playing all tones in the Sun Java ME DefaultColorPhone emulator*

The `playTone()` method can throw a `MediaException` if the note can't be played due to a hardware problem with the underlying mobile device. This usually occurs if the device doesn't support the playback of tones or if the device is using the built-in tone generator (by ringing to indicate an incoming call, for example). The method also throws the runtime `IllegalArgumentException` if the `duration` parameter is set to a negative value. Setting the `volume` parameter to a negative value results in the volume being set to 0 and 100 for any value over 100.

Using Tone Sequences with `ToneControl`

Of course, playing single tones isn't very useful in itself, except in games and applications where you may want to emphasize a critical point by playing a note of high pitch to draw the user's attention. Of more interest to most developers is the ability to play tones in a sequence that resembles popular songs so they can integrate them as ring tones on mobile devices or as background tones in games.

In the last section, you saw an example of playing tones in a sequence using the `playTone()` method. The underlying hardware was told to play single tones after a predefined interval, which is hardly suitable for realistic and continuous music.

The MMAPI defines the `ToneControl` interface to play tones in a sequence. You specify the notes that you want played in any sequence that you want, including repetitive blocks, along with tempo, volume, and duration using a `byte` array. A `Player` instance for playing tones can then be used like a normal `Player` instance, including using all control and lifecycle methods that you have learned about so far.

Before you learn how to create a `Player` instance for tone playback, you need to learn how to create tone sequences using the proprietary format defined by MMAPI.

Defining Tone Sequences

Tone sequences are defined using a slightly complicated format. Basically, a tone sequences is a collection of notes and durations values as you have learned in the previous sections. Note values range from 0 to 127 representing the various frequencies, but the value of the duration for each note is defined differently than before.

For tone sequences, the duration of each note is specified using *resolution*, as opposed to measuring it in milliseconds, which you've seen so far. To understand resolution, you need to know about another standard measurement of music using the beats present in musical notes.

Understanding Time Signature

A beat, as you may know, is the pulse of music. A *time signature* is used to define how many beats are in each *bar* (segment of time) of music along with how many note values constitute one beat. Just remember that time signature is a way to quantify musical work using the beats and the notes duration that constitute one beat.

Time signature is written as a fraction. The numerator indicates the number of beats in a bar and the denominator specifies the notes/beat (that is, the fraction of note duration that makes up one beat). The most common time signature is 4/4 (called *common time*), which means that there are 4 beats in a bar and 1/4 of a note makes up a beat.

Caution Make sure it is 1/4 of a note, not 4 notes.

In other words, common time written as 4/4 implies that there will be 4 beats per measure (musical measure or bar), with each beat composed of a quarter note.

Calculating Duration Using Resolution and Tempo

Let's say you want to play a tone for 1 second. To completely specify the duration, you also need to specify the tempo of the tone of the note because tempo, which represents the beats per minute of a note, varies the way the tone is heard. Let's say the tone that you want to play for 1 second should correspond to the common time you read about in the last section. That is, there should be 4 beats (one whole note) in the tone. Then, the resolution (which is a measure of duration) is specified using the following formula:

```
duration (milliseconds) * default resolution * default tempo / 240000 = resolution
```

Now, the default resolution equals 64, and the default tempo is equal to 120 beats per minute (bpm). Thus, playing a note for 1 second using the default resolution of 64 and the default tempo of 120 bpm by using the preceding formula equals the duration of

```
1000 * 64 * 120 / 240000 = 32
```

You are, of course, allowed to change the values of the default resolution and tempo. Thus, let's say the default tempo is changed to a faster beat, with 240 bpm as the value. The same tone will have the duration of

```
1000 * 64 * 240 / 240000 = 64
```

and for a slower beat of say 60 bpm:

```
1000 * 64 * 60 / 240000 = 16
```

Similarly, changing the default resolution to say 32 will result in a duration of

```
1000 * 32 * 120 / 240000 = 16
```

If all this seems too complicated for simple tone generation, then just remember to use the duration of a note in multiples of 8, with 8 being the lowest interval for which you want the tone to play and 120 being the longest. (You can use values that are not multiples of 8 as well, but they don't produce accurate results. No duration value can be greater than 127, as the duration is represented as a byte.)

Creating Sequences

Now that you know how to specify the note and calculate the duration for the note, you can now actually create a tone sequence to play using the `ToneControl` interface. The simplest tone sequence contains the `ToneControl` version number, which is always `1`, followed by note-duration pairs. Thus, the following is a single note tone sequence:

```
ToneControl.VERSION, 1
69, 32
```

The first two values in the sequence must always be `ToneControl.VERSION` and `1`. This identifies the version of this `ToneControl` (version of the `ToneControl`, not the sequence that you are describing), and by default is equal to `1` and unlikely to change. Next, you will have a note followed by its duration as described in the previous sections. Thus, the preceding sequence consists of a single note 69 that must be played for duration of 32 resolution.

There is no specific value to end a sequence. The last note-duration pair ends the sequence. Therefore, the preceding sequence is complete and valid. In code, it looks like the following:

```
byte[] sequence = new byte[] {
  ToneControl.VERSION, 1,
  69, 32,
};
```

`VERSION` is a constant in the `ToneControl` interface and is assigned a value of `-2`.

Let's now add more notes to this sequence. Actually, let's try to create the sequence for a popular tune, "Happy Birthday to You."

The tone sequence for the first stanza for this tune is shown here in note-duration pairs:

```
72, 4, 72, 2, 74, 8, 72, 8, 77, 8, 76, 16
```

In code, it looks like this:

```
byte[] sequence = new byte[] {
  ToneControl.VERSION, 1,
  72, 4, // Hap-
  ToneControl.SILENCE, 2,
  72, 2, // -py
  ToneControl.SILENCE, 2,
  74, 8, // Birth-
```

```
  ToneControl.SILENCE, 2,
  72, 8, // -day
  ToneControl.SILENCE, 2,
  77, 8, // to
ToneControl.SILENCE, 2,
76, 16 // you
};
```

To arrive at the note values and their durations for this sequence the trick is to use tone values that are available freely on the Internet for various other formats. However, these formats are different from the MMAPI format described so far, so you need to be able to convert between these formats and the MMAPI format.

The most popular format is the Ringing Tone Text Transfer Language (RTTTL) format, published by Nokia. Various Web sites on the Internet will provide you with the RTTTL format for a song or tune (just search for the name of the song or tune along with the text "RTTTL"). For example, the RTTTL format for the complete "Happy Birthday to You" tune is

```
Happy Birthday Song:d=4,o=5,b=125:16c, 32p, 32c, 32p, 8d, 32p, 8c, 32p, 8f, 32p, e,
 16p, 16c, 32p, 32c, 32p, 8d, 32p, 8c, 32p, 8g, 32p, f, 8p, 16c, 32p, 32c, 32p,
 8c6, 32p, 8a, 32p, 8f, 32p, 8e, 32p, 8d, 32p, 16a#, 32p, 32a#, 32p, 8a, 32p, 8f,
 32p, 8g, 32p, f
```

After you get the format for your particular song/tune, you need to convert it to the MMAPI format. You could try to understand the RTTTL format before you start to convert it to the MMAPI format, or you could just use the `RingToneConverter` class provided by Sun to do this for you.

Converting RTTTL to MMAPI Format Using `RingToneConverter`

The `RingToneConverter` class is provided with the examples in the Wireless Toolkit. This class is present in the MMADemo application as a standalone Java ME class. It converts the RTTTL format to the MMAPI format and is very handy. Listing 6-4 shows a MIDlet called `ConverterMIDlet` that uses this class to load and parse the RTTTL format of the "Happy Birthday to You" tune from a file. At this stage, it simply prints the equivalent MMAPI format.

Listing 6-4. *Using the* `RingToneConverter` *Class to Parse RTTTL Formats to MMAPI Format*

```
package com.apress.chapter6;

import javax.microedition.midlet.*;

import javax.microedition.media.*;
import javax.microedition.media.control.*;


public class ConverterMIDlet extends MIDlet {
```

```
public void startApp() {
  try {

    // Load the RTTTL format of the file as an InputStream
    RingToneConverter rtc =
      new RingToneConverter(
        getClass().getResourceAsStream(
          "/media/misc/happybday.rtttl"), "Happy Birthday");

    // get the equivalent sequence
    byte[] seq = rtc.getSequence();

    // print it
    printSequence(seq);

  } catch (Exception e) {
    System.err.println(e);
  }
}

private void printSequence(byte[] seq) {

  // print the sequence in a user friendly format
  byte bite = 0;
  for(int i = 0; i < seq.length; i++) {
    // the first control structure or the note value
    bite = seq[i];
    if(i % 2 == 0) {
      if(bite == ToneControl.VERSION)
        System.err.print("ToneControl.VERSION, ");
      else if(bite == ToneControl.TEMPO)
        System.err.print("ToneControl.TEMPO, ");
      else if(bite == ToneControl.SILENCE)
        System.err.print("ToneControl.SILENCE, ");
      else if(bite == ToneControl.RESOLUTION)
        System.err.print("ToneControl.RESOLUTION, ");
      else if(bite == ToneControl.SET_VOLUME)
        System.err.print("ToneControl.SET_VOLUME, ");
      else
        System.err.print(bite + ", ");
    }
    else {
      // the value of a control structure or the note duration
      if(i != (seq.length - 1))
        System.err.println(bite + ", ");
```

```
            else
              System.err.println(bite);
          }
        }

      }

      public void pauseApp() {
      }

      public void destroyApp(boolean unconditional) {
      }
}
```

Several constructors for the `RingToneConverter` class allow you to load the RTTTL format from a variety of sources. In this case, the RTTTL format for the "Happy Birthday to You" tune is stored in a text file called `happybday.rtttl`, and the `RingToneConverter(InputStream is, String tuneName)` constructor is used to load it. You can also use the `RingToneConverter(String url, String name)` constructor to load the file from a URL, or if you have the RTTTL format for a tune in a byte[] array, you can use the `RingToneConverter(byte[] data, String tuneName)` constructor directly. All constructors ultimately use the last constructor to parse the data.

Because the constructor also does the parsing of the RTTTL format, after it returns successfully, the corresponding MMAPI sequence is easily available by calling the `getSequence()` method. This method returns the MMAPI format in a byte array sequence, perfect for use with the `ToneControl`. You can print this sequence out using the `printSequence()` method in a format that makes it easy to copy and paste the resulting MMAPI sequence format in code.

Note the other constants from the `ToneControl` class that we've used besides the `ToneControl.VERSION` constant as listed in Table 6-5.

Table 6-5. `ToneControl` *Constants*

`ToneControl` Constant	`ToneControl` Value	`ToneControl` Description
`ToneControl.VERSION`	`-2`	The first control in a sequence and denotes the `ToneControl` version. Always equal to `1`.
`ToneControl.TEMPO`	`-3`	The second control in a sequence and used to set the tempo of the following sequence. Default value is `30`.
`ToneControl.RESOLUTION`	`-4`	The third control in a sequence and used to set the resolution of a sequence, as defined earlier. Default equals `64`.
`ToneControl.SET_VOLUME`	`-8`	Can be used at any stage in a sequence to set the volume of the notes following it.
`ToneControl.SILENCE`	`-1`	Can be used at any stage in a sequence to indicate a silent note value. It must, of course, be followed by a duration value.

Other constants will be discussed in the coming sections.

Creating, Defining, and Playing Blocks of Sequences

Ring tones that have repetitive stanzas don't need to define these stanzas separately each time they are required to be played. The MMAPI format allows you to define a repetitive block once and then refer to it whenever and wherever you want it played.

Sticking with the "Happy Birthday to You" tune, let's say you wanted the first stanza played a couple of times before moving on to the rest of the tune. This first stanza is the "Happy Birthday to You" part and is reproduced here:

```
72, 4, ToneControl.SILENCE, 2, 72, 2, ToneControl.SILENCE, 2, 74, 8,
ToneControl.SILENCE, 2, 72, 8, ToneControl.SILENCE, 2, 77, 8,
ToneControl.SILENCE, 2, 76, 16
```

To mark this as a block, start by putting the `ToneControl` control constant `BLOCK_START` with a number to identify this block and end with `BLOCK_END` using the same number:

```
ToneControl.BLOCK_START, 0,
72, 4, ToneControl.SILENCE, 2, 72, 2,
ToneControl.SILENCE, 2, 74, 8, ToneControl.SILENCE, 2, 72, 8,
ToneControl.SILENCE, 2, 77, 8, ToneControl.SILENCE, 2,  76, 16,
ToneControl.BLOCK_END, 0
```

The sequence has been marked as block `0`. This block can now be played anywhere in the sequence by using the `ToneControl` control constant `PLAY_BLOCK` followed by the number identifying the block. Thus, the following placed anywhere in a sequence after the block definition will play the block just defined:

```
ToneControl.PLAY_BLOCK, 0
```

Of course, you can't play a block that hasn't been defined. Therefore, all block definitions must come before an attempt to play them is made. There is no limit to the number of blocks that you can define.

The following byte array shows the sequence of playing the first two stanzas of "Happy Birthday to You," with the first stanza repeated once.

```
byte[] sequence = new byte[] {
  ToneControl.VERSION, 1,
  ToneControl.BLOCK_START, 0,
  72, 4, ToneControl.SILENCE, 2, 72, 2, ToneControl.SILENCE, 2, 74, 8,
  ToneControl.SILENCE,  2, 72, 8, ToneControl.SILENCE, 2, 77, 8,
  ToneControl.SILENCE, 2, 76, 16,
  ToneControl.BLOCK_END, 0,
  ToneControl.PLAY_BLOCK, 0, // plays block 0
  ToneControl.SILENCE, 4,  72, 4, ToneControl.SILENCE, 2, 72, 2,
  ToneControl.SILENCE, 2, 74, 8, ToneControl.SILENCE, 2, 72, 8,
  ToneControl.SILENCE, 2, 79, 8,
  ToneControl.SILENCE, 2, 77, 16, // plays the second stanza
  ToneControl.PLAY_BLOCK, 0 // plays block 0 again
}
```

`BLOCK_START`, `BLOCK_END`, and `PLAY_BLOCK` constants have a value of -5, -6, and -7, respectively.

Playing Sequences Using ToneControl and Player

Defining sequences is great, but it's of no use if you can't play them using MMAPI. So let's see how you can play the sequences you have learned to create in the previous sections using a combination of the `ToneControl` and `Player` interfaces.

Listing 6-4 showed how to parse a tone file in RTTTL format to the MMAPI format. Let's now add a method (see Listing 6-5) to the `ConverterMIDlet` defined in this listing that will play this converted tone.

Listing 6-5. *Playing Tone Sequences*

```
private void playSequence(byte[] seq) throws Exception {

  // create a Tone Player
  Player player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);

  // must realize it before getting ToneControl
  player.realize();

  // now get the ToneControl
  ToneControl toneControl = (ToneControl)player.getControl("ToneControl");

  // set the sequence
  toneControl.setSequence(seq);

  // and start the player - this will play the sequence
  player.start();
}
```

After creating or generating a sequence, playing it using MMAPI is relatively simple. You need a `Player` instance that can play tones. The `Manager` class provides a constant for creating a tone-based `Player` instance, which is supplied by the mobile device's hardware. This constant is the `TONE_DEVICE_LOCATOR`, and its value is equal to `device://tone`. So a tone-based `Player` instance is created by supplying this constant as the locator to the `createPlayer()` method.

This tone-based `Player` instance will now behave like any other instance. You can start it, stop it, realize it, prefetch it, and so on. To play tones, you need to access the `ToneControl` control provided by this instance. However, recall from Chapter 4 that an unrealized instance cannot provide any controls. Therefore, in Listing 6-5, the `Player` instance is realized before the `ToneControl` control is fetched using the `getControl()` method.

Finally, `ToneControl` is supplied the tone sequence by using the `setSequence()` method. This sequence is the final MMAPI formatted sequence, either created manually or by parsing the RTTTL format.

The `Player` instance is now ready and can be started when required. In Listing 6-5, the instance is started immediately by using the normal `start()` method. This instance will behave like any other `Player` instance; for example, you can pause it using the `stop()` method, make it repeat using the `setLoopCount()` method, or listen to events by attaching a `PlayerListener`.