```
    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"
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 1 of 118

```
    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>
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 2 of 118

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

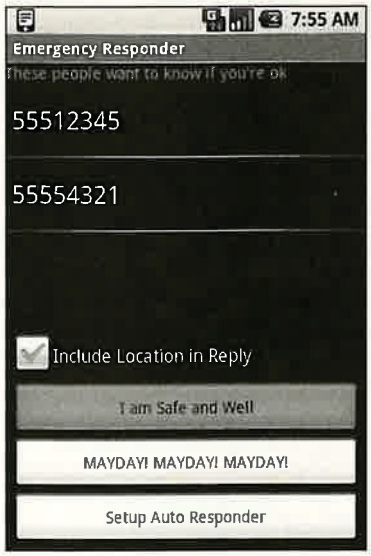


Figure 9-3

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);
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 3 of 118

```
    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) {
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 4 of 118

```
            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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 5 of 118

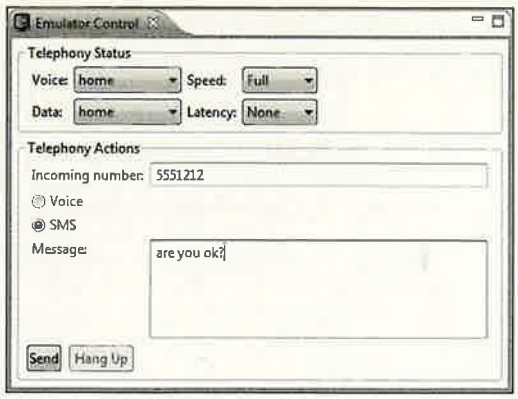


Figure 9-4

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();
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 6 of 118

```
        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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 7 of 118

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);
      }
    }
  }
};
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 8 of 118

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"
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 9 of 118

```
    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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 10 of 118

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);
  }

}
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 11 of 118

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();
  }
});
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 12 of 118

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();
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 13 of 118

```
    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>
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 15 of 118

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.

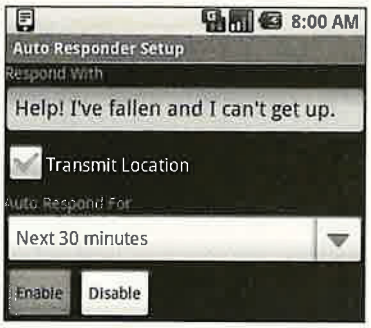


Figure 9-5

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 17 of 118

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);
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 19 of 118

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");
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 21 of 118

```
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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 22 of 118

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() {
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 23 of 118

```
    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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 25 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 26 of 118

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):

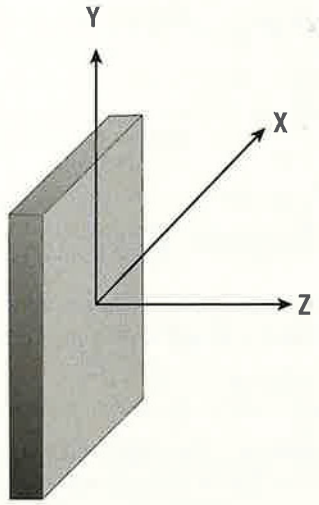


Figure 10-1

FIFTH THIRD EX1005
PART 3 OF 3
Page 27 of 118

- **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) { }
};
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 28 of 118

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;
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 29 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 31 of 118

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.

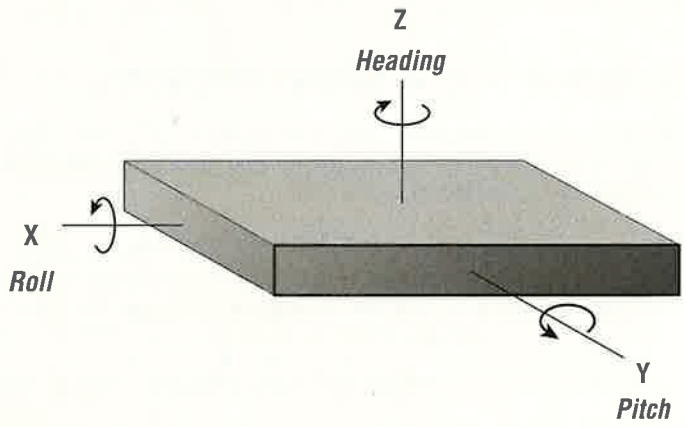


Figure 10-2

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.

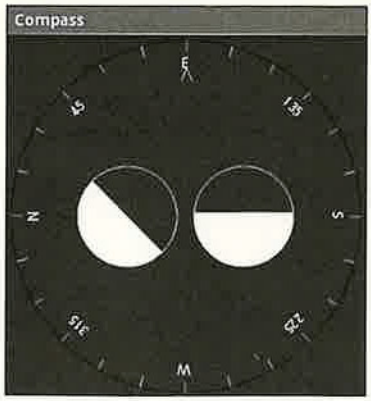


Figure 10-3

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);
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 35 of 118

```
    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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 36 of 118

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"/>
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 37 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 38 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 39 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 41 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 42 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 45 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 46 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 51 of 118

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);
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 53 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 54 of 118

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.*

FIFTH THIRD EX1005
PART 3 OF 3
Page 57 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 59 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 61 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 64 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 65 of 118

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) {
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 67 of 118

```
      // 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>
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 68 of 118

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>
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 69 of 118

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);
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 70 of 118

```
    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) {
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 71 of 118

```
    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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 73 of 118

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();
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 74 of 118

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" >
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 75 of 118

```
        </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+.

FIFTH THIRD EX1005
PART 3 OF 3
Page 76 of 118

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`).

FIFTH THIRD EX1005
PART 3 OF 3
Page 77 of 118

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`.

FIFTH THIRD EX1005
PART 3 OF 3
Page 78 of 118

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.

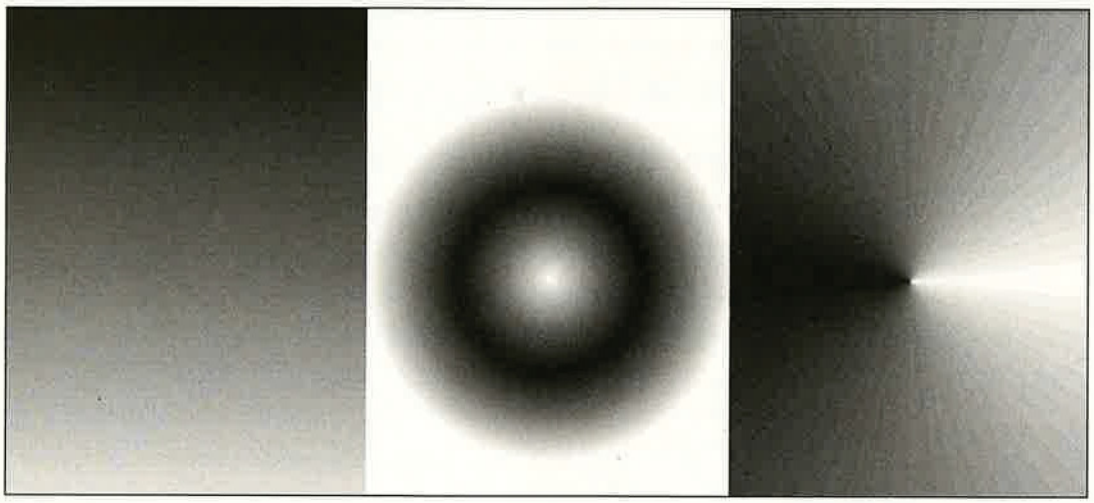


Figure 11-1

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;
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 80 of 118

```
// 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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 81 of 118

- `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).

FIFTH THIRD EX1005
PART 3 OF 3
Page 82 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 83 of 118

- **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

FIFTH THIRD EX1005
PART 3 OF 3
Page 84 of 118

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;
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 87 of 118

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;
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 88 of 118

```
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);
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 89 of 118

```
    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,
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 91 of 118

```
                                                    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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 92 of 118

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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 93 of 118

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();
```


FIFTH THIRD EX1005
PART 3 OF 3
Page 97 of 118

```
// 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.

FIFTH THIRD EX1005
PART 3 OF 3
Page 101 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 103 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 105 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 107 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 108 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 109 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 110 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 111 of 118

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

FIFTH THIRD EX1005
PART 3 OF 3
Page 112 of 118

powered by books24x7

rammer to Programmer™

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

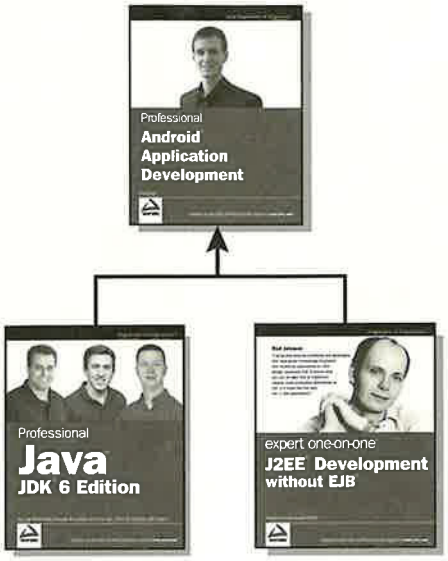
- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML

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.

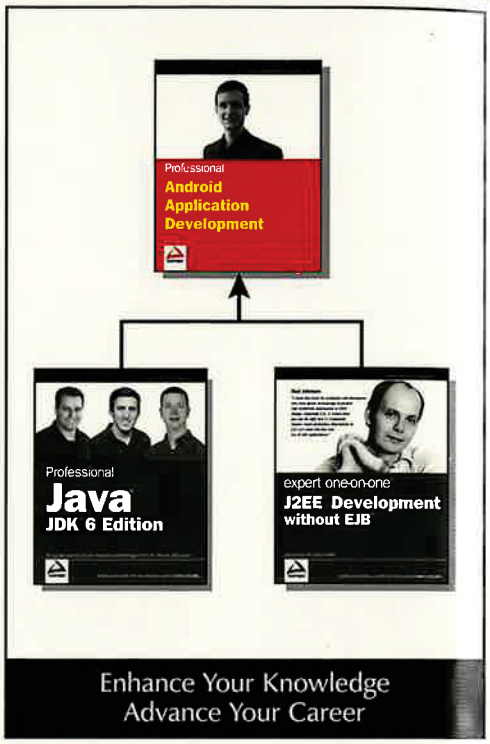
Expert One-on-One™
J2EETM Development without EJBTM
978-0-7645-5831-3
This hands-on guide shows you alternatives to EJB that can be used to create higher quality applications faster and at lower cost, and demonstrates how to leverage practical techniques and tools, including the popular open source Spring 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

ISBN: 978-0-470-34471-2
54499
9 780470 344712

38 144 AA 5171
BR1
05/09 02-013-01 Ohio
Group

Wrox
An Imprint of
WILEY

THE UNIVERSITY OF MICHIGAN

DATE DUE

OCT 21 2010

UNIVERSITY OF MICHIGAN
3 9015 07947 2265