11

Developing Android Services

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to create a service that runs in the background
- How to perform long-running tasks in a separate thread
- How to perform repeated tasks in a service
- How an activity and a service communicate

A service is an application in Android that runs in the background without needing to interact with the user. For example, while using an application, you may want to play some background music at the same time. In this case, the code that is playing the background music has no need to interact with the user, and hence it can be run as a service. Services are also ideal for situations in which there is no need to present a UI to the user. A good example of this scenario is an application that continually logs the geographical coordinates of the device. In this case, you can write a service to do that in the background. In this chapter, you will learn how to create your own services and use them to perform background tasks asynchronously.

CREATING YOUR OWN SERVICES

The best way to understand how a service works is by creating one. The following Try It Out shows you the steps to create a simple service. Subsequent sections add more functionality to this service. For now, you will learn how to start and stop a service.

TRY IT OUT Creating a Simple Service

codefile Services.zip available for download at Wrox.com

1. Using Eclipse, create a new Android project and name it **Services**.
2. Add a new Java Class file to the project and name it **MyService**. Populate the `MyService.java` file with the following code:

```
package net.learn2develop.Services;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.widget.Toast;

public class MyService extends Service {

    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // We want this service to continue running until it is explicitly
        // stopped, so return sticky.
        Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
        return START_STICKY;
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
    }
}
```

3. In the `AndroidManifest.xml` file, add the following statement in bold:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.learn2develop.Services"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="14" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
```

```
            android:label="@string/app_name"
            android:name=".ServicesActivity" >
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".MyService" />
    </application>

</manifest>
```

4. In the `main.xml` file, add the following statements in bold, replacing `TextView`:

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

<Button android:id="@+id/btnStartService"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Start Service"
    android:onClick="startService"/>

<Button android:id="@+id/btnStopService"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Stop Service"
    android:onClick="stopService" />

</LinearLayout>
```

5. Add the following statements in bold to the `ServicesActivity.java` file:

```
package net.learn2develop.Services;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ServicesActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void startService(View view) {
```

```
        startService(new Intent(getBaseContext(), MyService.class));
    }

    public void stopService(View view) {
        stopService(new Intent(getBaseContext(),
MyService.class));
    }

}
```

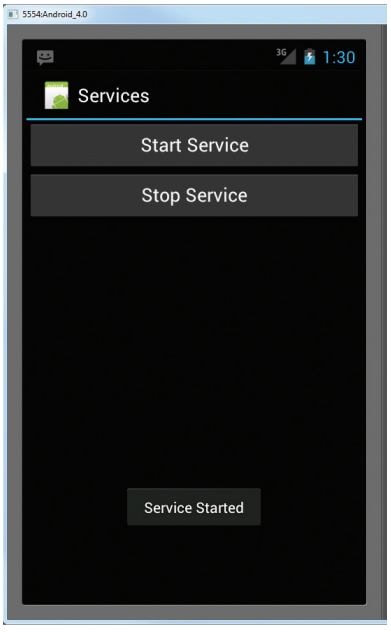

FIGURE 11-1

6. Press F11 to debug the application on the Android emulator.
7. Clicking the Start Service button will start the service (see Figure 11-1). To stop the service, click the Stop Service button.

How It Works

This example demonstrated the simplest service that you can create. The service itself is not doing anything useful, of course, but it serves to illustrate the creation process.

First, you defined a class that extends the `Service` base class. All services extend the `Service` class:

```
public class MyService extends Service {
}
```

Within the `MyService` class, you implemented three methods:

```
@Override
public IBinder onBind(Intent arg0) {  ...  }

@Override
public int onStartCommand(Intent intent, int flags, int startId) { ... }

@Override
public void onDestroy() { ... }
```

The `onBind()` method enables you to bind an activity to a service. This in turn enables an activity to directly access members and methods inside a service. For now, you simply return a `null` for this method. Later in this chapter you will learn more about binding.

The `onStartCommand()` method is called when you start the service explicitly using the `startService()` method (discussed shortly). This method signifies the start of the service, and you code it to do the things you need to do for your service. In this method, you returned the constant `START_STICKY` so that the service will continue to run until it is explicitly stopped.

The `onDestroy()` method is called when the service is stopped using the `stopService()` method. This is where you clean up the resources used by your service.

All services that you have created must be declared in the `AndroidManifest.xml` file, like this:

```
<service android:name=".MyService" />
```

If you want your service to be available to other applications, you can always add an intent filter with an action name, like this:

```
<service android:name=".MyService">
    <intent-filter>
        <action android:name="net.learn2develop.MyService" />
    </intent-filter>
</service>
```

To start a service, you use the `startService()` method, like this:

```
startService(new Intent(getBaseContext(), MyService.class));
```

If you are calling this service from an external application, then the call to the `startService()` method looks like this:

```
startService(new Intent("net.learn2develop.MyService"));
```

To stop a service, use the `stopService()` method:

```
stopService(new Intent(getBaseContext(), MyService.class));
```

Performing Long-Running Tasks in a Service

Because the service you created in the previous section does not do anything useful, in this section you will modify it so that it performs a task. In the following Try It Out, you will simulate the service of downloading a file from the Internet.

TRY IT OUT Making Your Service Useful

1. Using the Services project created in the first example, add the following statements in bold to the `ServicesActivity.java` file:

```
package net.learn2develop.Services;

import java.net.MalformedURLException;
import java.net.URL;

import android.app.Service;
```

```
import android.content.Intent;
import android.os.IBinder;
import android.widget.Toast;

public class MyService extends Service {

    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // We want this service to continue running until it is explicitly
        // stopped, so return sticky.
        //Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();

        try {
            int result = DownloadFile(new URL("http://www.amazon.com/somefile.pdf"));
            Toast.makeText(getBaseContext(),
                "Downloaded " + result + " bytes",
                Toast.LENGTH_LONG).show();
        } catch (MalformedURLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return START_STICKY;
    }

    private int DownloadFile(URL url) {
        try {
            //---simulate taking some time to download a file---
            Thread.sleep(5000);
        } catch (InterruptedException e) {
             e.printStackTrace();
        }
        //---return an arbitrary number representing
        // the size of the file downloaded---
        return 100;
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
    }
```

2. Press F11 to debug the application on the Android emulator.
3. Click the Start Service button to start the service to download the file. Note that the activity is frozen for a few seconds before the `Toast` class displays the "Downloaded 100 bytes" message (see Figure 11-2).

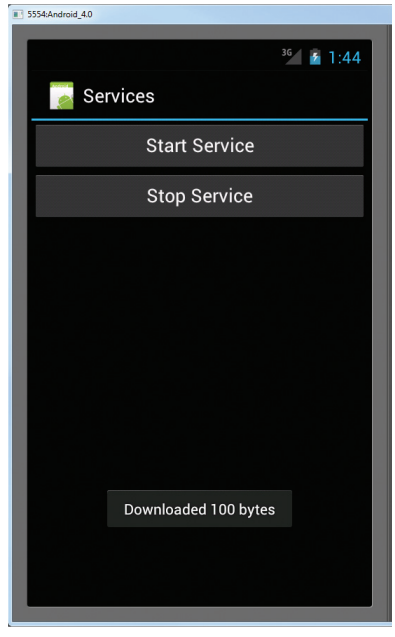

FIGURE 11-2

How It Works

In this example, your service calls the `DownloadFile()` method to simulate downloading a file from a given URL. This method returns the total number of bytes downloaded (which you have hardcoded as 100). To simulate the delays experienced by the service when downloading the file, you used the `Thread.Sleep()` method to pause the service for five seconds (5,000 milliseconds).

As you start the service, note that the activity is suspended for about five seconds, which is the time taken for the file to be downloaded from the Internet. During this time, the entire activity is not responsive, demonstrating a very important point: The service runs on the same thread as your activity. In this case, because the service is suspended for five seconds, so is the activity.

Hence, for a long-running service, it is important that you put all long-running code into a separate thread so that it does not tie up the application that calls it. The following Try It Out shows you how.

TRY IT OUT Performing Tasks in a Service Asynchronously

codefile Services.zip available for download at Wrox.com

1. Using the Services project created in the first example, add the following statements in bold to the `MyService.java` file:

```
package net.learn2develop.Services;

import java.net.MalformedURLException;
import java.net.URL;

import android.app.Service;
import android.content.Intent;
import android.os.AsyncTask;
import android.os.IBinder;
import android.util.Log;
import android.widget.Toast;

public class MyService extends Service {

    @Override
    public IBinder onBind(Intent arg0) {
```

```
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // We want this service to continue running until it is explicitly
        // stopped, so return sticky.
        //Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();

        try {
            new DoBackgroundTask().execute(
                    new URL("http://www.amazon.com/somefiles.pdf"),
                    new URL("http://www.wrox.com/somefiles.pdf"),
                    new URL("http://www.google.com/somefiles.pdf"),
                    new URL("http://www.learn2develop.net/somefiles.pdf"));

        } catch (MalformedURLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return START_STICKY;
    }

    private int DownloadFile(URL url) {
        try {
            //---simulate taking some time to download a file---
            Thread.sleep(5000);
        } catch (InterruptedException e) {
             e.printStackTrace();
        }
        //---return an arbitrary number representing
        // the size of the file downloaded---
        return 100;
    }

    private class DoBackgroundTask extends AsyncTask<URL, Integer, Long> {
        protected Long doInBackground(URL... urls) {
            int count = urls.length;
            long totalBytesDownloaded = 0;
            for (int i = 0; i < count; i++) {
                totalBytesDownloaded += DownloadFile(urls[i]);
                //---calculate percentage downloaded and
                // report its progress---
                publishProgress((int) (((i+1) / (float) count) * 100));
            }
            return totalBytesDownloaded;
        }

        protected void onProgressUpdate(Integer... progress) {
            Log.d("Downloading files",
                    String.valueOf(progress[0]) + "% downloaded");
            Toast.makeText(getBaseContext(),
                String.valueOf(progress[0]) + "% downloaded",
```

```
                Toast.LENGTH_LONG).show();
        }

        protected void onPostExecute(Long result) {
            Toast.makeText(getBaseContext(),
                    "Downloaded " + result + " bytes",
                    Toast.LENGTH_LONG).show();
            stopSelf();
        }
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
    }
}
```

2. Press F11 to debug the application on the Android emulator.
3. Click the Start Service button. The `Toast` class will display a message indicating what percentage of the download is completed. You should see four of them: 25%, 50%, 75%, and 100%.
4. You can see output similar to the following in the LogCat window:

```
12-06 01:58:24.967: D/Downloading files(6020): 25% downloaded
12-06 01:58:30.019: D/Downloading files(6020): 50% downloaded
12-06 01:58:35.078: D/Downloading files(6020): 75% downloaded
12-06 01:58:40.096: D/Downloading files(6020): 100% downloaded
```

How It Works

This example illustrates one way in which you can execute a task asynchronously within your service. You do so by creating an inner class that extends the `AsyncTask` class. The `AsyncTask` class enables you to perform background execution without needing to manually handle threads and handlers.

The `DoBackgroundTask` class extends the `AsyncTask` class by specifying three generic types:

```
private class DoBackgroundTask extends AsyncTask<URL, Integer, Long> {
```

In this case, the three types specified are `URL`, `Integer` and `Long`. These three types specify the data type used by the following three methods that you implement in an `AsyncTask` class:

- `doInBackground()` — This method accepts an array of the first generic type specified earlier. In this case, the type is `URL`. This method is executed in the background thread and is where you put your long-running code. To report the progress of your task, you call the `publishProgress()` method, which invokes the next method, `onProgressUpdate()`, which you implement in an `AsyncTask` class. The return type of this method takes the third generic type specified earlier, which is `Long` in this case.

- `onProgressUpdate()` — This method is invoked in the UI thread and is called when you call the `publishProgress()` method. It accepts an array of the second generic type specified earlier. In this case, the type is `Integer`. Use this method to report the progress of the background task to the user.
- `onPostExecute()` — This method is invoked in the UI thread and is called when the `doInBackground()` method has finished execution. This method accepts an argument of the third generic type specified earlier, which in this case is a `Long`.

Figure 11-3 summarizes the types specified and their relationship to the three methods inside a subclass of the `AsyncTask` class.

FIGURE 11-3

To download multiple files in the background, you created an instance of the `DoBackgroundTask` class and then called its `execute()` method by passing in an array of `URL`s:

```
try {
    new DoBackgroundTask().execute(
            new URL("http://www.amazon.com/somefiles.pdf"),
            new URL("http://www.wrox.com/somefiles.pdf"),
            new URL("http://www.google.com/somefiles.pdf"),
            new URL("http://www.learn2develop.net/somefiles.pdf"));

} catch (MalformedURLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

The preceding causes the service to download the files in the background, and reports the progress as a percentage of files downloaded. More important, the activity remains responsive while the files are downloaded in the background, on a separate thread.

Note that when the background thread has finished execution, you can manually call the `stopSelf()` method to stop the service:

```
protected void onPostExecute(Long result) {
    Toast.makeText(getBaseContext(),
            "Downloaded " + result + " bytes",
            Toast.LENGTH_LONG).show();
    stopSelf();
}
```

The `stopSelf()` method is the equivalent of calling the `stopService()` method to stop the service.

Performing Repeated Tasks in a Service

In addition to performing long-running tasks in a service, you might also perform some repeated tasks in a service. For example, you may write an alarm clock service that runs persistently in the background. In this case, your service may need to periodically execute some code to check whether a prescheduled time has been reached so that an alarm can be sounded. To execute a block of code to be executed at a regular time interval, you can use the `Timer` class within your service. The following Try It Out shows you how.

TRY IT OUT Running Repeated Tasks Using the Timer Class

codefile Services.zip available for download at Wrox.com

1. Using the Services project again, add the following statements in bold to the `MyService.java` file:

```
package net.learn2develop.Services;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Timer;
import java.util.TimerTask;

import android.app.Service;
import android.content.Intent;
import android.os.AsyncTask;
import android.os.IBinder;
import android.util.Log;
import android.widget.Toast;

public class MyService extends Service {
    int counter = 0;
    static final int UPDATE_INTERVAL = 1000;
    private Timer timer = new Timer();

    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // We want this service to continue running until it is explicitly
        // stopped, so return sticky.
        //Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();

        doSomethingRepeatedly();

        try {
            new DoBackgroundTask().execute(
                    new URL("http://www.amazon.com/somefiles.pdf"),
                    new URL("http://www.wrox.com/somefiles.pdf"),
                    new URL("http://www.google.com/somefiles.pdf"),
                    new URL("http://www.learn2develop.net/somefiles.pdf"));

        } catch (MalformedURLException e) {
```

```
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return START_STICKY;
}

private void doSomethingRepeatedly() {
    timer.scheduleAtFixedRate(new TimerTask() {
        public void run() {
            Log.d("MyService", String.valueOf(++counter));
        }
    }, 0, UPDATE_INTERVAL);
}

private int DownloadFile(URL url) {
    try {
        //---simulate taking some time to download a file---
        Thread.sleep(5000);
    } catch (InterruptedException e) {
         e.printStackTrace();
    }
    //---return an arbitrary number representing
    // the size of the file downloaded---
    return 100;
}

private class DoBackgroundTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalBytesDownloaded = 0;
        for (int i = 0; i < count; i++) {
            totalBytesDownloaded += DownloadFile(urls[i]);
            //---calculate percentage downloaded and
            // report its progress---
            publishProgress((int) (((i+1) / (float) count) * 100));
        }
        return totalBytesDownloaded;
    }

    protected void onProgressUpdate(Integer... progress) {
        Log.d("Downloading files",
                String.valueOf(progress[0]) + "% downloaded");
        Toast.makeText(getBaseContext(),
            String.valueOf(progress[0]) + "% downloaded",
            Toast.LENGTH_LONG).show();
    }

    protected void onPostExecute(Long result) {
        Toast.makeText(getBaseContext(),
                "Downloaded " + result + " bytes",
                Toast.LENGTH_LONG).show();
        stopSelf();
    }
}

@Override
```

```
    public void onDestroy() {
        super.onDestroy();

        if (timer != null){
            timer.cancel();
        }

        Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
    }
}
```

2. Press F11 to debug the application on the Android emulator.
3. Click the Start Service button.
4. Observe the output displayed in the LogCat window. It will be similar to the following:

```
12-06 02:37:54.118: D/MyService(7752): 1
12-06 02:37:55.109: D/MyService(7752): 2
12-06 02:37:56.120: D/MyService(7752): 3
12-06 02:37:57.111: D/MyService(7752): 4
12-06 02:37:58.125: D/MyService(7752): 5
12-06 02:37:59.137: D/MyService(7752): 6
```

How It Works

In this example, you created a `Timer` object and called its `scheduleAtFixedRate()` method inside the `doSomethingRepeatedly()` method that you have defined:

```
private void doSomethingRepeatedly() {
    timer.scheduleAtFixedRate( new TimerTask() {
        public void run() {
            Log.d("MyService", String.valueOf(++counter));
        }
    }, 0, UPDATE_INTERVAL);
}
```

You passed an instance of the `TimerTask` class to the `scheduleAtFixedRate()` method so that you can execute the block of code within the `run()` method repeatedly. The second parameter to the `scheduleAtFixedRate()` method specifies the amount of time, in milliseconds, before first execution. The third parameter specifies the amount of time, in milliseconds, between subsequent executions.

In the preceding example, you essentially print out the value of the counter every second (1,000 milliseconds). The service repeatedly prints the value of `counter` until the service is terminated:

```
@Override
public void onDestroy() {
    super.onDestroy();

    if (timer != null){
        timer.cancel();
    }

    Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
}
```

For the `scheduleAtFixedRate()` method, your code is executed at fixed time intervals, regardless of how long each task takes. For example, if the code within your `run()` method takes two seconds to complete, then your second task will start immediately after the first task has ended. Similarly, if your delay is set to three seconds and the task takes two seconds to complete, then the second task will wait for one second before starting.

Also, observe that you call the `doSomethingRepeatedly()` method directly in the `onStartCommand()` method, without needing to wrap it in a subclass of the `AsyncTask` class. This is because the `TimerTask` class itself implements the `Runnable` interface, which allows it to run on a separate thread.

Executing Asynchronous Tasks on Separate Threads Using IntentService

Earlier in this chapter, you learned how to start a service using the `startService()` method and stop a service using the `stopService()` method. You have also seen how you should execute long-running task on a separate thread — not the same thread as the calling activities. It is important to note that once your service has finished executing a task, it should be stopped as soon as possible so that it does not unnecessarily hold up valuable resources. That's why you use the `stopSelf()` method to stop the service when a task has been completed. Unfortunately, a lot of developers often forgot to terminate a service when it is done performing its task. To easily create a service that runs a task asynchronously and terminates itself when it is done, you can use the `IntentService` class.

The `IntentService` class is a base class for `Service` that handles asynchronous requests on demand. It is started just like a normal service; and it executes its task within a worker thread and terminates itself when the task is completed. The following Try It Out demonstrates how to use the `IntentService` class.

TRY IT OUT Using the IntentService Class to Auto-Stop a Service

codefile Services.zip available for download at Wrox.com

1. Using the Services project created in the first example, add a new Class file named **MyIntentService.java**.
2. Populate the `MyIntentService.java` file as follows:

```
package net.learn2develop.Services;

import java.net.MalformedURLException;
import java.net.URL;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

public class MyIntentService extends IntentService {

    public MyIntentService() {
        super("MyIntentServiceName");
    }

    @Override
```

```
    protected void onHandleIntent(Intent intent) {
        try {
            int result =
                DownloadFile(new URL("http://www.amazon.com/somefile.pdf"));
            Log.d("IntentService", "Downloaded " + result + " bytes");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    private int DownloadFile(URL url) {
        try {
            //---simulate taking some time to download a file---
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 100;
    }
}
```

3. Add the following statement in bold to the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.learn2develop.Services"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="14" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".ServicesActivity" >
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service android:name=".MyService">
            <intent-filter>
                <action android:name="net.learn2develop.MyService" />
            </intent-filter>
        </service>
        <service android:name=".MyIntentService" />

    </application>

</manifest>
```

4. Add the following statement in bold to the `ServicesActivity.java` file:

```
public void startService(View view) {
    //startService(new Intent(getBaseContext(), MyService.class));
    //OR
    //startService(new Intent("net.learn2develop.MyService"));
    startService(new Intent(getBaseContext(), MyIntentService.class));
}
```

5. Press F11 to debug the application on the Android emulator.

6. Click the Start Service button. After about five seconds, you should see something similar to the following statement in the LogCat window:

```
12-06 13:35:32.181: D/IntentService(861): Downloaded 100 bytes
```

How It Works

First, you defined the `MyIntentService` class, which extends the `IntentService` class instead of the `Service` class:

```
public class MyIntentService extends IntentService {
}
```

You needed to implement a constructor for the class and call its superclass with the name of the intent service (setting it with a string):

```
public MyIntentService() {
    super("MyIntentServiceName");
}
```

You then implemented the `onHandleIntent()` method, which is executed on a worker thread:

```
@Override
protected void onHandleIntent(Intent intent) {
    try {
        int result =
            DownloadFile(new URL("http://www.amazon.com/somefile.pdf"));
        Log.d("IntentService", "Downloaded " + result + " bytes");
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
}
```

The `onHandleIntent()` method is where you place the code that needs to be executed on a separate thread, such as downloading a file from a server. When the code has finished executing, the thread is terminated and the service is stopped automatically.

ESTABLISHING COMMUNICATION BETWEEN A SERVICE AND AN ACTIVITY

Often a service simply executes in its own thread, independently of the activity that calls it. This doesn't pose any problem if you simply want the service to perform some tasks periodically and the activity does not need to be notified about the service's status. For example, you may have a service that periodically logs the geographical location of the device to a database. In this case, there is no need for your service to interact with any activities, because its main purpose is to save the coordinates into a database. However, suppose you want to monitor for a particular location. When the service logs an address that is near the location you are monitoring, it might need to communicate that information to the activity. If so, you need to devise a way for the service to interact with the activity.

The following Try It Out demonstrates how a service can communicate with an activity using a `BroadcastReceiver`.

TRY IT OUT Invoking an Activity from a Service

codefile Services.zip available for download at Wrox.com

1. Using the Services project created earlier, add the following statements in bold to the `MyIntentService.java` file:

```
package net.learn2develop.Services;

import java.net.MalformedURLException;
import java.net.URL;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

public class MyIntentService extends IntentService {

    public MyIntentService() {
        super("MyIntentServiceName");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        try {
            int result =
                DownloadFile(new URL("http://www.amazon.com/somefile.pdf"));
            Log.d("IntentService", "Downloaded " + result + " bytes");

            //---send a broadcast to inform the activity
            // that the file has been downloaded---
            Intent broadcastIntent = new Intent();
            broadcastIntent.setAction("FILE_DOWNLOADED_ACTION");
            getBaseContext().sendBroadcast(broadcastIntent);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
```

```
    }

    private int DownloadFile(URL url) {
        try {
            //---simulate taking some time to download a file---
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return 100;
    }
}
```

2. Add the following statements in bold to the `ServicesActivity.java` file:

```
package net.learn2develop.Services;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class ServicesActivity extends Activity {
    IntentFilter intentFilter;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    public void onResume() {
        super.onResume();

        //---intent to filter for file downloaded intent---
        intentFilter = new IntentFilter();
        intentFilter.addAction("FILE_DOWNLOADED_ACTION");

        //---register the receiver---
        registerReceiver(intentReceiver, intentFilter);
    }

    @Override
    public void onPause() {
        super.onPause();

        //---unregister the receiver---
```

```
        unregisterReceiver(intentReceiver);
    }

    public void startService(View view) {
        //startService(new Intent(getBaseContext(), MyService.class));
        //OR
        //startService(new Intent("net.learn2develop.MyService"));
        startService(new Intent(getBaseContext(), MyIntentService.class));
    }

    public void stopService(View view) {
        stopService(new Intent(getBaseContext(), MyService.class));
    }

    private BroadcastReceiver intentReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            Toast.makeText(getBaseContext(), "File downloaded!",
                    Toast.LENGTH_LONG).show();
        }
    };

}
```

3. Press F11 to debug the application on the Android emulator.
4. Click the Start Service button. After about five seconds, the `Toast` class will display a message indicating that the file has been downloaded (see Figure 11-4).

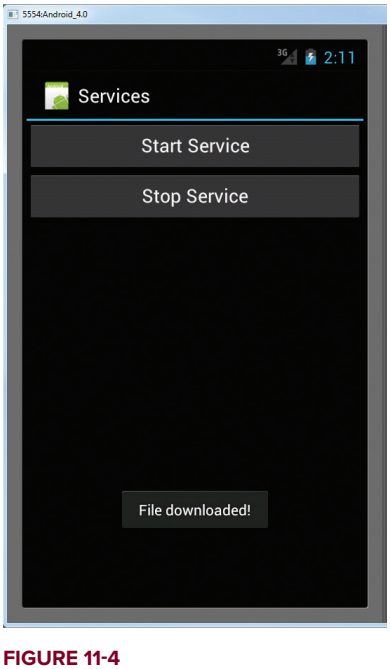

FIGURE 11-4

How It Works

To notify an activity when a service has finished its execution, you broadcast an intent using the `sendBroadcast()` method:

```
@Override
protected void onHandleIntent(Intent intent) {
    try {
        int result =
            DownloadFile(new URL("http://www.amazon.com/somefile.pdf"));
        Log.d("IntentService", "Downloaded " + result + " bytes");

        //---send a broadcast to inform the activity
        // that the file has been downloaded---
        Intent broadcastIntent = new Intent();
        broadcastIntent.setAction("FILE_DOWNLOADED_ACTION");
        getBaseContext().sendBroadcast(broadcastIntent);

    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
}
```

The action of this intent that you are broadcasting is set to `"FILE_DOWNLOADED_ACTION"`, which means any activity that is listening for this intent will be invoked. Hence, in your `ServicesActivity.java` file, you listen for this intent using the `registerReceiver()` method from the `IntentFilter` class:

```
@Override
public void onResume() {
    super.onResume();

    //---intent to filter for file downloaded intent---
    intentFilter = new IntentFilter();
    intentFilter.addAction("FILE_DOWNLOADED_ACTION");

    //---register the receiver---
    registerReceiver(intentReceiver, intentFilter);
}
```

When the intent is received, it invokes an instance of the `BroadcastReceiver` class that you have defined:

```
private BroadcastReceiver intentReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(getBaseContext(), "File downloaded!",
                Toast.LENGTH_LONG).show();
    }
  };
```

NOTE *Chapter 8 discusses the* `BroadcastReceiver` *class in more detail.*

In this case, you displayed the message "File downloaded!" Of course, if you need to pass some data from the service to the activity, you can make use of the `Intent` object. The next section discusses this.

BINDING ACTIVITIES TO SERVICES

So far, you have seen how services are created and how they are called and terminated when they are done with their task. All the services that you have seen are simple — either they start with a counter and increment at regular intervals or they download a fixed set of files from the Internet. However, real-world services are usually much more sophisticated, requiring the passing of data so that they can do the job correctly for you.

Using the service demonstrated earlier that downloads a set of files, suppose you now want to let the calling activity determine what files to download, instead of hardcoding them in the service. Here is what you need to do.

First, in the calling activity, you create an `Intent` object, specifying the service name:

```
public void startService(View view) {
    Intent intent = new Intent(getBaseContext(), MyService.class);
}
```

You then create an array of `URL` objects and assign it to the `Intent` object through its `putExtra()` method. Finally, you start the service using the `Intent` object:

```
public void startService(View view) {
    Intent intent = new Intent(getBaseContext(), MyService.class);
    try {
        URL[] urls = new URL[] {
                new URL("http://www.amazon.com/somefiles.pdf"),
                new URL("http://www.wrox.com/somefiles.pdf"),
                new URL("http://www.google.com/somefiles.pdf"),
                new URL("http://www.learn2develop.net/somefiles.pdf")};
        intent.putExtra("URLs", urls);
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
    startService(intent);
}
```

Note that the `URL` array is assigned to the `Intent` object as an `Object` array.

On the service's end, you need to extract the data passed in through the `Intent` object in the `onStartCommand()` method:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    // We want this service to continue running until it is explicitly
    // stopped, so return sticky.
    Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
    Object[] objUrls = (Object[]) intent.getExtras().get("URLs");
    URL[] urls = new URL[objUrls.length];
    for (int i=0; i<objUrls.length-1; i++) {
        urls[i] = (URL) objUrls[i];
    }
    new DoBackgroundTask().execute(urls);
    return START_STICKY;
}
```

The preceding first extracts the data using the `getExtras()` method to return a `Bundle` object. It then uses the `get()` method to extract out the `URL` array as an `Object` array. Because in Java you cannot directly cast an array from one type to another, you have to create a loop and cast each member of the array individually. Finally, you execute the background task by passing the URL array into the `execute()` method.

This is one way in which your activity can pass values to the service. As you can see, if you have relatively complex data to pass to the service, you have to do some additional work to ensure that the data is passed correctly. A better way to pass data is to bind the activity directly to the service so that the activity can call any public members and methods on the service directly. The following Try It Out shows you how to bind an activity to a service.

TRY IT OUT Accessing Members of a Property Directly through Binding

codefile Services.zip available for download at Wrox.com

1. Using the Services project created earlier, add the following statements in bold to the `MyService.java` file (note that you are modifying the existing `onStartCommand()`):

```
import android.os.Binder;

import android.os.IBinder;

public class MyService extends Service {
    int counter = 0;
    URL[] urls;
    static final int UPDATE_INTERVAL = 1000;
    private Timer timer = new Timer();
    private final IBinder binder = new MyBinder();

    public class MyBinder extends Binder {
        MyService getService() {
            return MyService.this;
        }
    }

    @Override
    public IBinder onBind(Intent arg0) {
```

```
        return binder;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // We want this service to continue running until it is explicitly
        // stopped, so return sticky.
        Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
        new DoBackgroundTask().execute(urls);
        return START_STICKY;
    }


    private void doSomethingRepeatedly() { … }

    private int DownloadFile(URL url) { ... }

    private class DoBackgroundTask extends AsyncTask<URL, Integer, Long> { ... }

    @Override
    public void onDestroy() { ... }
}
```

2. In the `ServicesActivity.java` file, add the following statements in bold (note the change to the existing `startService()` method):

```
import android.content.ComponentName;
import android.os.IBinder;
import android.content.ServiceConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class ServicesActivity extends Activity {
    IntentFilter intentFilter;

    MyService serviceBinder;
    Intent i;

    private ServiceConnection connection = new ServiceConnection() {
        public void onServiceConnected(
            ComponentName className, IBinder service) {
            //--called when the connection is made--
            serviceBinder = ((MyService.MyBinder)service).getService();
            try {
                URL[] urls = new URL[] {
                    new URL("http://www.amazon.com/somefiles.pdf"),
                    new URL("http://www.wrox.com/somefiles.pdf"),
                    new URL("http://www.google.com/somefiles.pdf"),
                    new URL("http://www.learn2develop.net/somefiles.pdf")};
                    //---assign the URLs to the service through the
                    // serviceBinder object---
                    serviceBinder.urls = urls;
                } catch (MalformedURLException e) {
                    e.printStackTrace();
                }
```

```
                startService(i);
            }
            public void onServiceDisconnected(ComponentName className) {
                //---called when the service disconnects---
                serviceBinder = null;
            }
        };

        public void startService(View view) {
            i = new Intent(ServicesActivity.this, MyService.class);
            bindService(i, connection, Context.BIND_AUTO_CREATE);
        }

        @Override
        public void onCreate(Bundle savedInstanceState) { ... }

        @Override
        public void onResume() { ... }

        @Override
        public void onPause() { ... }

        public void stopService(View view) { ... }

        private BroadcastReceiver intentReceiver = new BroadcastReceiver() {
            ...
        };

}
```

3. Press F11 to debug the application. Clicking the Start Service button will start the service as normal.

How It Works

To bind activities to a service, you must first declare an inner class in your service that extends the `Binder` class:

```
public class MyBinder extends Binder {
    MyService getService() {
        return MyService.this;
    }
}
```

Within this class you implemented the `getService()` method, which returns an instance of the service.

You then created an instance of the `MyBinder` class:

```
private final IBinder binder = new MyBinder();
```

You also modified the `onBind()` method to return the `MyBinder` instance:

```
@Override
public IBinder onBind(Intent arg0) {
    return binder;
}
```

In the `onStartCommand()` method, you then called the `execute()` method using the `urls` array, which you declared as a public member in your service:

```
public class MyService extends Service {
    int counter = 0;
    URL[] urls;
...
...
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // We want this service to continue running until it is explicitly
        // stopped, so return sticky.
        Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
        new DoBackgroundTask().execute(urls);
        return START_STICKY;
    }
}
```

This `URL` array can be set directly from your activity, which you did next.

In the `ServicesActivity.java` file, you first declared an instance of your service and an `Intent` object:

```
MyService serviceBinder;
Intent i;
```

The `serviceBinder` object will be used as a reference to the service, which you accessed directly.

You then created an instance of the `ServiceConnection` class so that you could monitor the state of the service:

```
private ServiceConnection connection = new ServiceConnection() {
    public void onServiceConnected(
        ComponentName className, IBinder service) {
        //---called when the connection is made---
        serviceBinder = ((MyService.MyBinder)service).getService();
        try {
            URL[] urls = new URL[] {
                new URL("http://www.amazon.com/somefiles.pdf"),
                new URL("http://www.wrox.com/somefiles.pdf"),
                new URL("http://www.google.com/somefiles.pdf"),
                new URL("http://www.learn2develop.net/somefiles.pdf")};
                //---assign the URLs to the service through the
                // serviceBinder object---
                serviceBinder.urls = urls;
            } catch (MalformedURLException e) {
                e.printStackTrace();
            }
            startService(i);
    }
    public void onServiceDisconnected(ComponentName className) {
        //---called when the service disconnects---
        serviceBinder = null;
    }
};
```

You need to implement two methods: `onServiceConnected()` and `onServiceDisconnected()`. The `onServiceConnected()` method is called when the activity is connected to the service; the `onServiceDisconnected()` method is called when the service is disconnected from the activity.

In the `onServiceConnected()` method, when the activity is connected to the service, you obtained an instance of the service by using the `getService()` method of the `service` argument and then assigning it to the `serviceBinder` object. The `serviceBinder` object is a reference to the service, and all the members and methods in the service can be accessed through this object. Here, you created a URL array and then directly assigned it to the public member in the service:

```
URL[] urls = new URL[] {
    new URL("http://www.amazon.com/somefiles.pdf"),
    new URL("http://www.wrox.com/somefiles.pdf"),
    new URL("http://www.google.com/somefiles.pdf"),
    new URL("http://www.learn2develop.net/somefiles.pdf")};
    //---assign the URLs to the service through the
    // serviceBinder object---
    serviceBinder.urls = urls;
```

You then started the service using an `Intent` object:

```
startService(i);
```

Before you can start the service, you have to bind the activity to the service. This you did in the `startService()` method of the Start Service button:

```
public void startService(View view) {
    i = new Intent(ServicesActivity.this, MyService.class);
    bindService(i, connection, Context.BIND_AUTO_CREATE);
}
```

The `bindService()` method enables your activity to be connected to the service. It takes three arguments: an `Intent` object, a `ServiceConnection` object, and a flag to indicate how the service should be bound.

UNDERSTANDING THREADING

So far, you have seen how services are created and why it is important to ensure that your long-running tasks are properly handled, especially when updating the UI thread. Earlier in this chapter (as well as in Chapter 10), you also saw how to use the `AsyncTask` class for executing long-running code in the background. This section briefly summarizes the various ways to handle long-running tasks correctly using a variety of methods available.

For this discussion, assume that you have an Android project named **Threading**. The `main.xml` file contains a Button and TextView:

```
<?xml version="1.0" encoding="utf-8"?>
 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
     android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

    <Button
        android:id="@+id/btnStartCounter"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start"
        android:onClick="startCounter" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="TextView" />

</LinearLayout>
```

Suppose you want to display a counter on the activity, from 0 to 1,000. In your `ThreadingActivity` class, you have the following code:

```
package net.learn2develop.Threading;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class ThreadingActivity extends Activity {
    TextView txtView1;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        txtView1 = (TextView) findViewById(R.id.textView1);
    }

    public void startCounter(View view) {
        for (int i=0; i<=1000; i++) {
            txtView1.setText(String.valueOf(i));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
```

```
                Log.d("Threading", e.getLocalizedMessage());
            }
        }
    }
}
```

When you run the application and click the Start button, the application is briefly frozen, and after a while you may see the message shown in Figure 11-5.

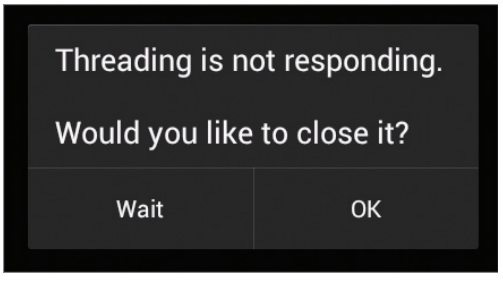

FIGURE 11-5

The UI freezes because the application is continuously trying to display the value of the counter at the same time it is pausing for one second after it has been displayed. This ties up the UI, which is waiting for the display of the numbers to be completed. The result is a nonresponsive application that will frustrate your users.

To solve this problem, one option is to wrap the part of the code that contains the loop using a `Thread` and `Runnable` class, like this:

```
public void startCounter(View view) {
    new Thread(new Runnable() {
        public void run() {
            for (int i=0; i<=1000; i++) {
                txtView1.setText(String.valueOf(i));
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Log.d("Threading", e.getLocalizedMessage());
                }
            }
        }
    }).start();
}
```

In the preceding code, you first create a class that implements the `Runnable` interface. Within this class, you put your long-running code within the `run()` method. The `Runnable` block is then started using the `Thread` class.

NOTE *A `Runnable` is a block of code that can be executed by a thread.*

However, the preceding application will not work, and it will crash if you try to run it. This code that is placed inside the `Runnable` block is on a separate thread, and in the preceding example you are trying to update the UI from another thread, which is not a safe thing to do because Android UIs are not thread-safe. To resolve this, you need to use the `post()` method of a `View` to create

another `Runnable` block to be added to the message queue. In short, the new `Runnable` block created will be executed in the UI thread, so it would now be safe to execute your application:

```
public void startCounter(View view) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i=0; i<=1000; i++) {
                final int valueOfi = i;

                //---update UI---
                txtView1.post(new Runnable() {
                    public void run() {
                        //---UI thread for updating---
                        txtView1.setText(String.valueOf(valueOfi));
                    }
                });

                //---insert a delay
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Log.d("Threading", e.getLocalizedMessage());
                }
            }
        }
    }).start();
}
```

This application will now work correctly, but it is complicated and makes your code difficult to maintain.

A second option to update the UI from another thread is to use the `Handler` class. A `Handler` enables you to send and process messages, similar to using the `post()` method of a `View`. The following code snippets shows a `Handler` class called `UIupdater` that updates the UI using the message that it receives:

NOTE *For the following code to work, you need to import the* `android.os.Handler` *package as well as add the* `static` *modifier to* `txtView1`.

```
//---used for updating the UI on the main activity---
static Handler UIupdater = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        byte[] buffer = (byte[]) msg.obj;

        //---convert the entire byte array to string---
        String strReceived = new String(buffer);

        //---display the text received on the TextView---
        txtView1.setText(strReceived);
```

```
            Log.d("Threading", "running");
        }
    };

    public void startCounter(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i=0; i<=1000; i++) {
                    //---update the main activity UI---
                    ThreadingActivity.UIupdater.obtainMessage(
                        0,  String.valueOf(i).getBytes() ).sendToTarget();
                    //---insert a delay
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        Log.d("Threading", e.getLocalizedMessage());
                    }
                }
            }
        }).start();
    }

}
```

A detailed discussion of the `Handler` class is beyond the scope of this book. For more details, check out the documentation at `http://developer.android.com/reference/android/os/Handler.html`.

So far, the two methods just described enable you to update the UI from a separate thread. In Android, you could use the simpler `AsyncTask` class to do this. Using the `AsyncTask`, you could rewrite the preceding code as follows:

```
    private class DoCountingTask extends AsyncTask<Void, Integer, Void> {
        protected Void doInBackground(Void... params) {
            for (int i = 0; i < 1000; i++) {
                //---report its progress---
                publishProgress(i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Log.d("Threading", e.getLocalizedMessage());
                }
            }
            return null;
        }

        protected void onProgressUpdate(Integer... progress) {
            txtView1.setText(progress[0].toString());
            Log.d("Threading", "updating...");
        }
    }

    public void startCounter(View view) {
        new DoCountingTask().execute();
    }
```

The preceding code will update the UI safely from another thread. What about stopping the task? If you run the preceding application and then click the Start button, the counter will start to display from zero. However, if you press the back button on the emulator/device, the task continues to run even though the activity has been destroyed. You can verify this through the LogCat window. If you want to stop the task, use the following code snippets:

```
public class ThreadingActivity extends Activity {
    static TextView txtView1;

    DoCountingTask task;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        txtView1 = (TextView) findViewById(R.id.textView1);
    }

    public void startCounter(View view) {
        task = (DoCountingTask) new DoCountingTask().execute();
    }

    public void stopCounter(View view) {
        task.cancel(true);
    }

    private class DoCountingTask extends AsyncTask<Void, Integer, Void> {
        protected Void doInBackground(Void... params) {
            for (int i = 0; i < 1000; i++) {
                //---report its progress---
                publishProgress(i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Log.d("Threading", e.getLocalizedMessage());
                }
                if (isCancelled()) break;
            }
            return null;
        }

        protected void onProgressUpdate(Integer... progress) {
            txtView1.setText(progress[0].toString());
            Log.d("Threading", "updating...");
        }
    }

    @Override
    protected void onPause() {
        super.onPause();
        stopCounter(txtView1);
    }
}
```

To stop the `AsyncTask` subclass, you need to get an instance of it first. To stop the task, call its `cancel()` method. Within the task, you call the `isCancelled()` method to check whether the task should be terminated.

SUMMARY

In this chapter, you learned how to create a service in your Android project to execute long-running tasks. You have seen the many approaches you can use to ensure that the background task is executed in an asynchronous fashion, without tying up the main calling activity. You have also learned how an activity can pass data into a service, and how you can alternatively bind to an activity so that it can access a service more directly.

EXERCISES

1. Why is it important to put long-running code in a service on a separate thread?
2. What is the purpose of the `IntentService` class?
3. Name the three methods you need to implement in an `AsyncTask` class.
4. How can a service notify an activity of an event happening?
5. For threading, what is the recommended method to ensure that your code runs without tying up the UI of your application?

Answers to the exercises can be found in Appendix C.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Creating a service	Create a class and extend the `Service` class.
Implementing the methods in a service	Implement the following methods: `onBind()`, `onStartCommand()`, and `onDestroy()`.
Starting a service	Use the `startService()` method.
Stopping a service	Use the `stopService()` method.
Performing long-running tasks	Use the `AsyncTask` class and implement three methods: `doInBackground()`, `onProgressUpdate()`, and `onPostExecute()`.
Performing repeated tasks	Use the `Timer` class and call its `scheduleAtFixedRate()` method.
Executing tasks on a separate thread and auto-stopping a service	Use the `IntentService` class.
Enabling communication between an activity and a service	Use the `Intent` object to pass data into the service. For a service, broadcast an `Intent` to notify an activity.
Binding an activity to a service	Use the `Binder` class in your service and implement the `ServiceConnection` class in your calling activity.
Updating the UI from a `Runnable` block	Use the `post()` method of a view to update the UI. Alternatively, you can also use a `Handler` class. The recommended way is to use the `AsyncTask` class.