Android Services

Services are important Android application components that can greatly enhance an application. An **Android Service** might be used as follows:

- to perform functions in the background that do not require user input, or
- to supply information to other applications

In this module, you will learn the ability to create and interact with an Android Service. Then you will learn how:

- to define a remote interface using the **Android Interface Definition Language** (AIDL)
- to pass objects through this interface by creating a class that implements a Parcelable object

Determining when to use Services

A Service in the Android SDK can mean one of two things:

1. A Service can mean a background process that performs some useful operation at regular intervals, or
2. A Service can be an interface for a remote object, called from within an application.

In both cases, the Service object extends the Service class from the Android SDK, and it can be a standalone component **or** part of an application with a complete user interface.
Certainly, not all applications require or use services. However, you might want to consider a Service if your application meets certain criteria, such as the following:

- The application performs lengthy or resource-intensive processing that does not require input from the user.
- The application must perform certain functions routinely, or at regular intervals, such as uploading or downloading fresh content or logging the current location.
- The application performs a lengthy operation that, if canceled because the application exits, would be wasteful to restart. An example of this is downloading large files.
- The application performs a lengthy operation while the user might be doing multiple activities. A Service can be used to span processing across the bounds of Activity lifecycles.
- The application needs to expose and provide data or information services (think web services) to other Android applications without the need of a user interface.

Understanding the lifecycle of a Service

Before we get into the details of how to create a **Service**, let's look at how services interact with the Android operating system. First, it should be noted that a Service implementation must be registered in an application's manifest file using the <service> tag. The Service implementation might also define and enforce any permissions needed for starting, stopping, and binding to the Service, as well as make specific Service calls.

After it has been implemented, an Android Service can be started using the Context.startService() method. If the Service was already running when the **startService()** method was called, these subsequent calls don't start further instances of the Service. The Service continues to run until either the **Context.stopService()** method is called or the Service completes its tasks and stops itself using the **stopSelf()** method.

To connect to a Service, interested applications use the **Context.bindService()** method to obtain a connection. If that Service is not running, the Service is created at that time. After the connection is established, the interested applications can begin making requests of that Service, if the applications have the appropriate permissions.

For example, a Magic Eight Ball application might have an underlying Service that can receive yes-or-no questions and provide Yoda-style answers. Any interested application can connect to the Magic Eight Ball Service, ask a question ("Will my app flourish in the Google Play store?"), and receive the result ("Signs point to Yes."). The application can disconnect from the Service when finished using the **Context.unbindService()** method.

Warning:
Like applications, services can be killed by the Android operating system under low-memory conditions. Also like applications, services have a main thread that can be blocked, causing the system to become unresponsive. Always offload intensive processing to worker threads using whatever methodology you like, even when implementing a Service.

Creating a Service

Creating an Android Service involves extending the Service class and adding a <service> block to the AndroidManifest.xml permissions file. For example, the GPXService class, discussed later in this section, overrides the **onCreate()**, **onStart()**, **onStartCommand()**, and **onDestroy()** methods to begin with. Defining the Service name enables other applications to start the Service that runs in the background and stop it. Both the **onStart()** and **onStartCommand()** methods are essentially the same, with the exception that **onStart()** is deprecated in API Level 5 and above. (The default implementation of the onStartCommand() on API Level 5 or greater is to call **onStart()**, which returns an appropriate value so that behavior is compatible with previous versions.) In the following example, both methods are implemented.

For this example, we implement a simple Service that listens for Global Positioning System (GPS) changes, displays notifications at regular intervals, and then provides access to the most

recent location data via a remote interface. The following code gives a simple definition of the Service class called GPXService:

```
public class GPXService extends Service {
    public static final String GPX_SERVICE =
        "com.company.android.GPXService.SERVICE";

    private LocationManager location = null;
    private NotificationManager notifier = null;

    @Override
    public void onCreate() {
        super.onCreate();
    }
    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
    }

    @Override
    public void onStartCommand(Intent intent, int flags, int startId) {
        super.onStart(intent, startId);
    }

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

You need to understand the lifecycle of a Service because it's different from that of an Activity. If a Service is started by the system with a call to the **Context.startService()** method, the **onCreate()** method is called just before the **onStart()** or onStartCommand() methods. However, if the Service is bound to with a call to the **Context.bindService()** method, the **onCreate()** method is called just before the **onBind()** method. The **onStart()** and **onStartCommand()** methods are not called in this case. We talk more about binding to a Service later in this module.

Finally, when the Service is finished—that is, it is stopped and no other process is bound to it—the **onDestroy()** method is called. Everything for the Service must be cleaned up in this method.
With this in mind, here is the full implementation of the **onCreate()** method for the GPXService class previously introduced:

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

    location = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    notifier = (NotificationManager)
```

```
                    getSystemService(Context.NOTIFICATION_SERVICE);
}
```

Because the object doesn't yet know if the next call is to either of the start methods or the **onBind()** method, we make a couple of quick initialization calls, but no background processing is started. Even this might be too much if neither of these objects is used by the interface provided by the binder.

Because we can't always predict what version of Android our code will run on, we can simply implement both the **onStart()** and **onStartCommand()** methods and have them call a third method that provides a common implementation. This enables us to customize behaviors on later Android versions while being compatible with earlier versions. To do this, the project needs to be built for an SDK of Level 5 or higher while having a **minSdkValue** of whatever earlier versions are supported. Of course, we highly recommend testing on multiple platform versions to verify that the behavior is as you expect. Here are sample implementations of the onStartCommand() and onStart() methods:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId ) {
    Log.v(DEBUG_TAG, "onStartCommand() called, must be on L5 or later");

    if (flags != 0) {
        Log.w(DEBUG_TAG, "Redelivered or retrying service start: "+flags);
    }

    doServiceStart(intent, startId);
    return Service.START_REDELIVER_INTENT;
}

@Override
public void onStart(Intent intent, int startId) {
    super.onStart(intent, startId);
    Log.v(DEBUG_TAG, "onStart() called, must be on L3 or L4");
    doServiceStart(intent,startId);
}
```

Next, let's look at the implementation of the **doServiceStart()** method in greater detail:

```
@Override
public void doServiceStart(Intent intent, int startId) {
    updateRate = intent.getIntExtra(EXTRA_UPDATE_RATE, -1);
    if (updateRate == -1) {
        updateRate = 60000;
    }

    Criteria criteria = new Criteria();
    criteria.setAccuracy(Criteria.NO_REQUIREMENT);
    criteria.setPowerRequirement(Criteria.POWER_LOW);
```

```
        location = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

        String best = location.getBestProvider(criteria, true);

        location.requestLocationUpdates(best, updateRate, 0, trackListener);

        Intent toLaunch = new Intent(getApplicationContext(),
            ServiceControlActivity.class);
        PendingIntent intentBack = PendingIntent.getActivity(
            getApplicationContext(), 0, toLaunch, 0);

        NotificationCompat.Builder builder = new NotificationCompat.Builder(
            getApplicationContext());
        builder.setTicker("Builder GPS Tracking");
        builder.setSmallIcon(android.R.drawable.stat_notify_more);
        builder.setWhen(System.currentTimeMillis());
        builder.setContentTitle("Builder GPS Tracking");
        builder.setContentText("Tracking start at " + updateRate
            + "ms intervals with [" + best + "] as the provider.");
        builder.setContentIntent(intentBack);
        builder.setAutoCancel(true);
        Notification notify = builder.build();

        notifier.notify(GPS_NOTIFY, notify);
    }
```

The background processing starts in the two start methods. In this example, though, the background processing is actually just registering for an update from another Service.

Tip:
The use of a callback to receive updates is recommended over doing background processing to poll for updates. Most mobile devices have limited battery life. Continual running in the background, or even just polling, can use a **substantial amount of battery power**. In addition, implementing callbacks for the users of your Service is more efficient for the same reasons.

In this case, we turn on the GPS for the duration of the process, which might affect battery life even though we request a lower-power method of location determination. Keep power management in mind when developing services so that your Service does not drain the user's battery.

The **Intent** extras object retrieves data passed in by the process requesting the Service. Here, we retrieve one value, **EXTRA_UPDATE_RATE**, for determining the length of time between updates. The string for this, **update-rate**, must be published externally, either in developer documentation or in a publicly available class file, so that users of the Service know about it. The implementation details of the LocationListener object, **trackListener**, are not relevant to the discussion on services. However, processing should be kept to a minimum to avoid interrupting what the user is doing in the foreground. Some testing might be required to determine how much processing a particular phone can handle before the user notices performance issues.

There are two common methods of communicating data to the user:

1. To use notifications.
    a. This is the least intrusive method and can be used to drive users to the application for more information. It also means the users don't need to be actively using their phone at the time of the notification because it is queued. For instance, a weather application might use notifications to provide weather updates every hour.
2. To use **Toast** messages. From some services, this might work well, especially if the user expects frequent updates and those updates work well overlaid briefly on the screen, regardless of what the user is currently doing. For instance, a background music player could briefly overlay the current song title when the song changes.

The **onDestroy()** method is called when no clients are bound to the Service and a request for the Service to be stopped has been made via a call to the **Context.stopService()** method, or a call has been made to the **2** method from within the Service. At this point, everything should be gracefully cleaned up because the Service ceases to exist.

Here is an example of the **onDestroy()** method:

```
@Override
public void onDestroy() {
    if (location != null) {
        location.removeUpdates(trackListener);
        location = null;
    }

    Intent toLaunch = new Intent(getApplicationContext(),
        ServiceControlActivity.class);
    PendingIntent intentBack = PendingIntent.getActivity(
        getApplicationContext(), 0, toLaunch, 0);

    NotificationCompat.Builder builder = new NotificationCompat.Builder(
        getApplicationContext());
    builder.setTicker("Builder GPS Tracking");
    builder.setSmallIcon(android.R.drawable.stat_notify_more);
    builder.setWhen(System.currentTimeMillis());
    builder.setContentTitle("Builder GPS Tracking");
    builder.setContentText("Tracking stopped");
    builder.setContentIntent(intentBack);
    builder.setAutoCancel(true);

    Notification notify = builder.build();

    notifier.notify(GPS_NOTIFY, notify);
    super.onDestroy();
}
```

Here, we stop updates to the **LocationListener** object. This stops all our background processing. Then, we notify the user that the Service is terminating. Only a single call to the **onDestroy()** method happens, regardless of how many times the start methods are called.

The system does not know about a Service unless it is defined within the **AndroidManifest.xml** permissions file using the `<service>` tag. Here is the <service> tag we must add to the Android manifest file:

```
<service
    android:enabled="true"
    android:name="GPXService">
    <intent-filter>
        <action android:name=
            "com.company.android.GPXService.SERVICE" />
    </intent-filter>
</service>
```

This block of XML defines the Service name, GPXService, and that the Service is enabled. Then, using an intent filter, we use the same string that we defined within the class. This is the string that is used later on when controlling the Service. With this block of XML inside the application section of the manifest, the system now knows that the Service exists and can be used by other applications.

Controlling a Service

At this point, the example code has a complete implementation of a Service. Now we write code to control the Service we previously defined:

```
Intent service = new Intent("com.company.android.GPXService.SERVICE");
service.putExtra("update-rate", 5000);
startService(service);
```

Starting a Service is as straightforward as creating an Intent with the Service name and calling the **startService()** method. In this example, we also set the Intent extra parameter called **update-rate** to 5 seconds. That rate is quite frequent but works well for testing. For practical use, we probably want this set to 60 seconds or more. This code triggers a call to the **onCreate()** method, if the Service isn't bound to or running already. It also triggers a call to the **onStart()** or **onStartCommand()** methods, even if the Service is already running.

Later, when we finish with the Service, it needs to be stopped using the following code:

```
Intent service = new Intent("com.company.android.GPXService.SERVICE");
stopService(service);
```

This code is essentially the same as the code for starting the Service but with a call to the **stopService()** method. This calls the **onDestroy()** method if there are no bindings to it. However, if there are bindings, **onDestroy()** is not called until those are also terminated. This means background processing might continue despite a call to the **stopService()** method. If there is a need to control the background processing separate from these system calls, a remote interface is required.

Implementing a Remote Interface

Sometimes it is useful to have more control over a Service than just system calls to start and stop its activities. However, before a client application can bind to a Service for making other method calls, you need to define the interface. The Android SDK includes a useful tool and file format for remote interfaces for this purpose.

To define a remote interface, you must declare the interface in an AIDL file, implement the interface, and then return an instance of the interface when the **onBind()** method is called.

Using the example GPXService service we already built in this module, we now create a remote interface for it. This remote interface has a method, which can be called especially for returning the last location logged. You can use only primitive types and objects that implement the Parcelable (android.os.Parcelable) protocol with remote Service calls. This is because these calls may cross process boundaries where memory can't be shared. The AIDL compiler handles the details of crossing these boundaries when the rules are followed. The Location object implements the Parcelable interface so it can be used.

Here is the AIDL file for this interface, IRemoteInterface:

```
package com.company.android.services;

interface IRemoteInterface {
    Location getLastLocation();
}
```

When using the Android integrated development environment (IDE), you can add this AIDL file, **IRemoteInterface.aidl**, to the project under the appropriate package and the Android SDK does the rest. Now we must implement the code for the interface. Here is an example implementation:

```
private final IRemoteInterface.Stub
    mRemoteInterfaceBinder = new IRemoteInterface.Stub() {
        public Location getLastLocation() {
            Log.v("interface", "getLastLocation() called");
            return lastLocation;
        }
    };
```

The Service code has already stored the last location received as a member variable, so we can simply return that value. With the interface implemented, it needs to be returned from the **onBind()** method of the Service:

```
@Override
public IBinder onBind(Intent intent) {
    // we only have one, so no need to check the intent
    return mRemoteInterfaceBinder;
}
```

If multiple interfaces are implemented, the Intent passed in can be checked within the **onBind()** method to determine what action is to be taken and which interface should be returned. In this example, though, we have only one interface and don't expect any other information within the Intent, so we simply return the interface.

We also add the class name of the binder interface to the list of actions supported by the intent filter for the Service within the AndroidManifest.xml file. Doing this isn't required but is a useful convention to follow and allows the class name to be used. The following block is added to the `<service>` tag definition:

```
<action android:name ="com.company.android.services.IRemoteInterface" />
```

The Service can now be used through this interface. This is done by implementing a ServiceConnection object and calling the **bindService()** method. When finished, the **unbindService()** method must be called so the system knows that the application has finished using the Service. The connection remains even if the reference to the interface is gone.

Here is an implementation of a **ServiceConnection** object's two main methods, **onServiceConnected()** and **onServiceDisconnected()**:

```
public void onServiceConnected(ComponentName name,
    IBinder service) {

    mRemoteInterface = IRemoteInterface.Stub.asInterface(service);
    Log.v("ServiceControl", "Interface bound.");
}

public void onServiceDisconnected(ComponentName name) {
    mRemoteInterface = null;
    Log.v("ServiceControl", "Remote interface no longer bound");
}
```

When the **onServiceConnected()** method is called, an **IRemoteInterface** instance that can be used to make calls to the interface we previously defined is retrieved. A call to the remote interface looks like any call to an interface now:

```
Location loc = mRemoteInterface.getLastLocation();
```

Remember that remote interface calls operate across process boundaries and are completed synchronously. As such, you should place them within a separate thread, as any lengthy call would be.

To use this interface from another application, you should place the AIDL file in the project and appropriate package. The call to **onBind()** triggers a call to **s** after the call to the Service's **onCreate()** method. Remember, the **onStart()** and **onStartCommand()** methods are not called in this case.

```
bindService(new Intent(IRemoteInterface.class.getName()),
    this, Context.BIND_AUTO_CREATE);
```

In this case, the Activity we call from also implements the ServiceConnection interface. This code also demonstrates why it is a useful convention to use the class name as an intent filter. Because we have both intent filters and we don't check the action on the call to the **onBind()** method, we can also use the other intent filter, but the code here is clearer.

When done with the interface, a call to **unbindService()** disconnects the interface. However, a callback to the **onServiceDisconnected()** method does not mean that the Service is no longer bound; the binding is still active at that point, just not the connection.

Implementing a Parcelable Class

In the example so far, we have been lucky in that the Location class implements the Parcelable interface. What if a new object needs to be passed through a remote interface?

Let's take the following class, **GPXPoint**, as an example:

```
public final class GPXPoint {
    public int latitude;
    public int longitude;
    public Date timestamp;
    public double elevation;

    public GPXPoint() {}
}
```

The GPXPoint class defines a location point that is similar to a GeoPoint but also includes the time the location was recorded and the elevation. This data is commonly found in the popular GPX file format. On its own, the GPX file format is not a basic format that the system recognizes for passing through to a remote interface. However, if the class implements the Parcelable interface and we then create an AIDL file from it, the object can be used in a remote interface.

To fully support the Parcelable type, we need to implement a few methods and a **Parcelable.Creator<GPXPoint>**. The following is the same class now modified to be a Parcelable class:

```
public final class GPXPoint implements Parcelable {
    public int latitude;
    public int longitude;
    public Date timestamp;
    public double elevation;

    public static final Parcelable.Creator<GPXPoint>
        CREATOR = new Parcelable.Creator<GPXPoint>() {

        public GPXPoint createFromParcel(Parcel src) {
            return new GPXPoint(src);
        }

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

    public GPXPoint() {}

    private GPXPoint(Parcel src) {
        readFromParcel(src);
    }

    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(latitude);
        dest.writeInt(longitude);
        dest.writeDouble(elevation);
        dest.writeLong(timestamp.getTime());
    }

    public void readFromParcel(Parcel src) {
        latitude = src.readInt();
        longitude = src.readInt();
        elevation = src.readDouble();
        timestamp = new Date(src.readLong());
    }

    public int describeContents() {
        return 0;
    }
}
```

The **writeToParcel()** method is required and flattens the object in a particular order using supported primitive types within a Parcel. When the class is created from a Parcel, the Creator is called, which in turn calls the private constructor. For readability, we also created a **readFromParcel()** method that reverses the flattening, reading the primitives in the same order

that they were written and creating a new Date object.

Now you must create the AIDL file for this class. You should place it in the same directory as the Java file and name it GPXPoint.aidl to match. You should make the contents look like the following:

```
package com.company.android.services;
parcelable GPXPoint;
```

Now the GPXPoint class can be used in remote interfaces. This is done in the same way as any other native type or Parcelable object. You can modify the **IRemoteInterface.aidl** file to look like the following:

```
package com.company.android.services;
import com.company.android.services.GPXPoint;

interface IRemoteInterface {
    Location getLastLocation();
    GPXPoint getGPXPoint();
}
```

Additionally, we can provide an implementation for this method within the interface, as follows:

```
public GPXPoint getGPXPoint() {
    if (lastLocation == null) {
        return null;
    } else {
        Log.v("interface", "getGPXPoint() called");
        GPXPoint point = new GPXPoint();

        point.elevation = lastLocation.getAltitude();
        point.latitude = (int) (lastLocation.getLatitude() * 1E6);
        point.longitude = (int) (lastLocation.getLongitude() * 1E6);
        point.timestamp = new Date(lastLocation.getTime());

        return point;
    }
}
```

As can be seen, nothing particularly special needs to happen. Just by making the object Parcelable, it can now be used for this purpose.

Using the IntentService Class

Offloading regularly performed tasks to a work queue is an easy and efficient way to process multiple requests without the cumbersome overhead of creating a full Service. The **IntentService class** (android.app.IntentService) is a simple type of Service that can be used to

handle such tasks asynchronously by way of Intent requests. Each Intent is added to the work queue associated with that IntentService and is handled sequentially.

You can send data back to the application by simply

- broadcasting the result as an Intent object, and
- using a broadcast receiver to catch the result and use it within the application.

Certainly, not all applications require or use a Service, or more specifically an IntentService. However, you may want to consider one if your application meets certain criteria, such as:

- The application routinely performs the same or similar blocking or resource-intensive processing operations that do not require input from the user in which requests for such operations can "pile up," requiring a queue to handle the requests in an organized fashion. Image processing and data downloading are examples of such processing.
- The application performs certain blocking operations at regular intervals but does not need to perform these routine tasks so frequently as to require a permanent, "always on" Service. Examples of such operations include but are not limited to
    - accessing local storage content providers,
    - accessing application databases,
    - accessing a network, or
    - processing large image processing
- The application routinely dispatches "work" but doesn't need an immediate response. For example, an email application might use an IntentService work queue to queue up each message to be packaged up and sent out to a mail server. All networking code would then be separated from the main user interface of the application.

Now let's look at an example of how you might use IntentService.

Let's assume you have an application with a screen that performs some processing each time the user provides some input, and then displays the result. For example, you might have an EditText control for taking some textual input, a Button control to commit the text and start the processing, and a TextView control for displaying the result. The code in the Button click handler within the Activity class would look something like this:

```
EditText input = (EditText) findViewById(R.id.txt_input);
String strInputMsg = input.getText().toString();
SystemClock.sleep(5000);
TextView result = (TextView) findViewById(R.id.txt_result);
result.setText(strInputMsg + " "
    + DateFormat.format("MM/dd/yy h:mmaa", System.currentTimeMillis()));
```

All this click handler does is retrieve some text from an EditText control on the screen, hang around doing nothing for 5 seconds, and then generate some information to display in the TextView control as a result. In reality, your application would probably not just sit around sleeping but would do some real work. As written, the click processing runs on the main UI

thread. This means that every time the user clicks on the Button control, the entire application becomes unresponsive for at least 5 seconds. The user must wait for the task to finish before continuing to use the application because the task is being completed on the main thread.

Wouldn't it be great if we could dispatch the processing request each time the user clicked the Button, but let the user interface remain responsive so the user can go about his or her business? Let's implement a simple IntentService that does just that. Here's our simple IntentService implementation:

```
public class SimpleIntentService extends IntentService {
    public static final String PARAM_IN_MSG = "imsg";
    public static final String PARAM_OUT_MSG = "omsg";

    public SimpleIntentService() {
        super("SimpleIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        String msg = intent.getStringExtra(PARAM_IN_MSG);
        SystemClock.sleep(5000);
        String resultTxt = msg + " " + DateFormat.format("MM/dd/yy h:mmaa",
              System.currentTimeMillis());
        Intent broadcastIntent = new Intent();
        broadcastIntent.setAction(ResponseReceiver.ACTION_RESP);
        broadcastIntent.addCategory(Intent.CATEGORY_DEFAULT);
        broadcastIntent.putExtra(PARAM_OUT_MSG, resultTxt);
        sendBroadcast(broadcastIntent);
    }
}
```

We use **Intent** extras (a Bundle of additional information to send to the component) to send some data associated with the specific task request, in a manner similar to passing data between Activity classes. In this case, we take the incoming EditText text value and package it into the PARAM_IN_MSG extra. Once the processing is complete, we use a broadcast Intent to tell anyone interested that the Service has finished the task. Your IntentService needs to do this only if the user interface needs to be updated. If the task simply updates the underlying application database or the shared preferences or what have you, your application would not need to be informed directly, as Cursor objects and such would be updated automatically when some underlying data changed.

Now, turn your attention back to the Activity class that hosts your application user interface with the Button control. Update the Button click handler to send a new task request to the **SimpleIntentService**. The request is packaged as an Intent, the incoming parameter is set (the data associated with the task), and the request is fired off using the **startService()** method.

```
EditText input = (EditText) findViewById(R.id.txt_input);
String strInputMsg = input.getText().toString();
```

```
Intent msgIntent = new Intent(this, SimpleIntentService.class);
msgIntent.putExtra(SimpleIntentService.PARAM_IN_MSG, strInputMsg);
startService(msgIntent);
```

Finally, define a BroadcastReceiver object for use by the application Activity, to listen for the results of each task completing and update the user interface accordingly:

```
public class ResponseReceiver extends BroadcastReceiver {
    public static final String ACTION_RESP =
        "com.mamlambo.intent.action.MESSAGE_PROCESSED";

    @Override
    public void onReceive(Context context, Intent intent) {
        TextView result = (TextView) findViewById(R.id.txt_result);
        String text = intent.getStringExtra(SimpleIntentService.PARAM_OUT_MSG);
        result.setText(text);
    }
}
```

The BroadcastReceiver class's **onReceive()** callback method does the work of reacting to a new broadcast from your **SimpleIntentService**. It updates the TextView control based upon the Intent extra data, which is the "result" from the task processing. Your application should register the broadcast receiver only when it needs to listen for results, and then unregister it when it's no longer needed. To manage this, first add a private member variable to your Activity, like this:

```
private ResponseReceiver receiver;
```

Activities typically register for broadcasts in their **onCreate()** or **onResume()** methods by creating an IntentFilter, like this:

```
IntentFilter filter = new IntentFilter(ResponseReceiver.ACTION_RESP);
filter.addCategory(Intent.CATEGORY_DEFAULT);
receiver = new ResponseReceiver();
registerReceiver(receiver, filter);
```

Similarly, it is typical to unregister the receiver when the Activity class no longer needs to react to results, such as in the **onPause()** or **onDestroy()** methods:

```
unregisterReceiver(receiver);
```

Finally, don't forget to register your SimpleIntentService in your Android manifest file, like this:

```
<service android:name="SimpleIntentService"/>
```

That's the complete implementation of our example IntentService:

- The Activity shoots off requests to the **SimpleIntentService** each time the Button control is clicked.

- The Service handles the queuing, processing, and broadcasting of the result of each task asynchronously.
- The Service shuts itself down when there's nothing left to do and starts back up if a new request comes in.
- Meanwhile, the application Activity remains responsive because it is no longer processing each request on the same thread that handles the UI.
- The user interface is responsive throughout all processing, allowing the user to continue to use the application.
- The user can hit the Button control five times in succession and trigger five tasks to be sent to the IntentService without having to wait 5 seconds between each click.

Summary

The Android SDK provides the Service mechanism that can be used to implement background tasks and to share functionality across multiple applications. By creating an interface through the use of AIDL, a Service can expose functionality to other applications without having to distribute libraries or packages. Creating objects with the Parcelable interface enables developers to extend the data that can also be passed across process boundaries.

Care should be taken when creating a background Service. Poorly designed background services might have a substantial negative impact on handset performance and battery life. In addition to standard testing, you should test a Service implementation with respect to these issues.

Prudent creation of a Service, though, can dramatically enhance the appeal of an application or service you might provide. Service creation is a powerful tool provided by the Android SDK for designing applications that are simply not possible on other mobile platforms. The IntentService class can be used to create a simple service that acts as a work queue.