

An Overview of Android Intents

- By this stage of the book, it should be clear that Android applications are comprised, among other things, of one or more activities.
- An area that has yet to be covered in extensive detail, however, is the mechanism by which one activity can trigger the launch of another activity.
- As outlined briefly in a Chapter 6 - Android Architecture, Application Anatomy and View Binding, this is achieved primarily by using Intents.
- Prior to working through some Android Studio based example implementations of intents in the following chapters, the goal of this chapter is to provide an overview of intents in the form of explicit intents and implicit intents together with an introduction to intent filters.

An Overview of Intents

- Intents (`android.content.Intent`) are the messaging system by which one activity is able to launch another activity.
- An activity can, for example, issue an intent to request the launch of another activity contained within the same application.
- Intents also, however, go beyond this concept by allowing an activity to request the services of any other appropriately registered activity on the device for which permissions are configured.
- Consider, for example, an activity contained within an application that requires a web page to be loaded and displayed to the user.
- Rather than the application having to contain a second activity to perform this task, the code can simply send an intent to the Android runtime requesting the services of any activity that has registered the ability to display a web page.
- The runtime system will match the request to available activities on the device and either launch the activity that matches or, in the event of multiple matches, allow the user to decide which activity to use.
- Intents also allow for the transfer of data from the sending activity to the receiving activity.
- In the previously outlined scenario, for example, the sending activity would need to send the URL of the web page to be displayed to the second activity.
- Similarly, the receiving activity may also be configured to return data to the sending activity when the required tasks are completed.
- Though not covered until later chapters, it is also worth highlighting the fact that, in addition to launching activities, intents are also used to launch and communicate with services and broadcast receivers.
- Intents are categorized as either explicit or implicit.

Explicit Intents

- An explicit intent requests the launch of a specific activity by referencing the component name (which is actually the class name) of the target activity.
- This approach is most common when launching an activity residing in the same application as the sending activity (since the class name is known to the application developer).
- An explicit intent is issued by creating an instance of the Intent class, passing through the activity context and the component name of the activity to be launched.
- A call is then made to the startActivity() method, passing the intent object as an argument.

- For example, the following code fragment issues an intent for the activity with the class name ActivityB to be launched:

```
Intent i = new Intent(this, ActivityB.class); startActivity(i);
```

- Data may be transmitted to the receiving activity by adding it to the intent object before it is started via calls to the putExtra() method of the intent object.
- Data must be added in the form of key-value pairs.
- The following code extends the previous example to add String and integer values with the keys “myString” and “myInt” respectively to the intent:

```
Intent i = new Intent(this, ActivityB.class);  
i.putExtra("myString", "This is a message for ActivityB");  
i.putExtra("myInt", 100);  
  
startActivity(i);
```

- The data is received by the target activity as part of a Bundle object which can be obtained via a call to getIntent(). getExtras().
- The getIntent() method of the Activity class returns the intent that started the activity, while the getExtras() method (of the Intent class) returns a Bundle object containing the data.
- For example, to extract the data values passed to ActivityB:

```
Bundle extras = getIntent().getExtras();  
if (extras != null)  
{  
    String myString = extras.getString("myString");  
    int myInt = extras.getInt("myInt");  
}
```

- When using intents to launch other activities within the same application, it is essential that those activities be listed in the application manifest file.

- The following AndroidManifest.xml contents are correctly configured for an application containing activities named ActivityA and ActivityB:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.niu.your_Z-ID.intent1.intent1" >

    <application
        android:icon="@mipmap/ic_launcher" android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name="edu.niu.your_Z-ID.intent1.intent1.ActivityA" >

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="ActivityB"
            android:label="ActivityB" >
        </activity>
    </application>
</manifest>
```

Returning Data from an Activity

- As the example in the previous section stands, while data is transferred to ActivityB, there is no way for data to be returned to the first activity (which we will call ActivityA).
- This can, however, be achieved by launching ActivityB as a sub-activity of ActivityA.
- An activity is started as a sub-activity by starting the intent with a call to the startActivityForResult() method instead of using startActivity().
- In addition to the intent object, this method is also passed a request code value which can be used to identify the return data when the sub-activity returns.
- For example:

```
startActivityForResult(i, REQUEST_CODE);
```

- In order to return data to the parent activity, the sub-activity must implement the finish() method, the purpose of which is to create a new intent object containing the data to be returned, and then calling the setResult() method of the enclosing activity, passing through a result code and the intent containing the return data.
- The result code is typically RESULT_OK, or RESULT_CANCELED, but may also be a custom value subject to the requirements of the developer.

- In the event that a sub-activity crashes, the parent activity will receive a `RESULT_CANCELED` result code.
- The following code, for example, illustrates the code for a typical sub-activity `finish()` method:

```
public void finish()
{
    Intent data = new Intent();

    data.putExtra("returnString1", "Message to parent activity");
    setResult(RESULT_OK, data);
    super.finish();
}
```

- In order to obtain and extract the returned data, the parent activity must implement the `onActivityResult()` method, for example:

```
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data)
{
    String returnString;
    if (requestCode == REQUEST_CODE && resultCode == RESULT_OK)
    {
        if (data.hasExtra("returnString1"))
        {
            returnString = data.getExtras().getString("returnString1");
        }
    }
}
```

- Note that the above method checks the returned request code value to make sure that it matches that passed through to the `startActivityForResult()` method.
- When starting multiple sub-activities it is especially important to use the request code to track which activity is currently returning results, since all will call the same `onActivityResult()` method on exit.

Implicit Intents

- Unlike explicit intents, which reference the class name of the activity to be launched, implicit intents identify the activity to be launched by specifying the action to be performed and the type of data to be handled by the receiving activity.
- For example, an action type of `ACTION_VIEW` accompanied by the URL of a web page in the form of a URI object will instruct the Android system to search for, and subsequently launch, a web browser capable activity.
- The following implicit intent will, when executed on an Android device, result in the designated web page appearing in a web browser activity:

```
Intent intent = new Intent(Intent.ACTION_VIEW,
                           Uri.parse("https://www.ebookfrenzy.com"));

startActivity(intent);
```

An Overview of Android Intents

- When the above implicit intent is issued by an activity, the Android system will search for activities on the device that have registered the ability to handle ACTION_VIEW requests on http scheme data using a process referred to as intent resolution.
- In the event that a single match is found, that activity will be launched.
- If more than one match is found, the user will be prompted to choose from the available activity options.

Using Intent Filters

- Intent filters are the mechanism by which activities “advertise” supported actions and data handling capabilities to the Android intent resolution process.
- Continuing the example in the previous section, an activity capable of displaying web pages would include an intent filter section in its manifest file indicating support for the ACTION_VIEW type of intent requests on http scheme data.
- It is important to note that both the sending and receiving activities must have requested permission for the type of action to be performed.
- This is achieved by adding <uses-permission> tags to the manifest files of both activities.
- For example, the following manifest lines request permission to access the internet and contacts database:

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

- The following AndroidManifest.xml file illustrates a configuration for an activity named WebViewActivity with intent filters and permissions enabled for internet access:

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

    <uses-sdk android:minSdkVersion="10" />

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

    <application
        android:icon="@mipmap/ic_launcher"
```

```

        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.BROWSABLE" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="https" android:host="www.ebookfrenzy.com"/>
            </intent-filter>
        </activity>
    </application>

</manifest>

```

Checking Intent Availability

- It is generally unwise to assume that an activity will be available for a particular intent, especially since the absence of a matching action will typically result in the application crashing.
- Fortunately, it is possible to identify the availability of an activity for a specific intent before it is sent to the runtime system.
- The following method can be used to identify the availability of an activity for a specified intent action type:

```

public static boolean isIntentAvailable(Context context, String action)
{
    final PackageManager packageManager = context.getPackageManager();
    final Intent intent = new Intent(action);

    List<ResolveInfo> list =
        packageManager.queryIntentActivities(intent,
                                            PackageManager.MATCH_DEFAULT_ONLY);
    return list.size() > 0;
}

```

Android Explicit Intents – A Worked Example

- We will now the theory of using intents to launch activities into practice through the creation of an example application.
- The example Android Studio application project created in this chapter will demonstrate the use of an explicit intent to launch an activity, including the transfer of data between sending and receiving activities.
- Don't worry. We will very soon demonstrate the use of implicit intents.

(continued)

Creating the Explicit Intent Example Application

- Select the Start a new Android Studio project quick start option from the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.
- Enter ExplicitIntent into the Name field and specify edu.niu.your_Z-ID.explicitintent as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

Designing the User Interface Layout for MainActivity

- The user interface for MainActivity will consist of a ConstraintLayout view containing EditText (Plain Text), TextView and Button views named editText1, textView1 and button1 respectively.
- Using the Project tool window, locate the activity_main.xml resource file for MainActivity (located under app > res > layout) and double-click on it to load it into the Android Studio Layout Editor tool.
- Select and delete the default “Hello World!” TextView.
- Drag a TextView widget from the palette and drop it so that it is centered within the layout and use the Attributes tool window to assign an ID of textView1.
- Drag a Button object from the palette and position it so that it is centered horizontally and located beneath the bottom edge of the TextView.
- Change the text property so that it reads “Ask Question” and configure the onClick property to call a method named askQuestion.
- Next, add a Plain Text object so that it is centered horizontally and positioned above the top edge of the TextView.
- Using the Attributes tool window, remove the “Name” string assigned to the text property and set the ID to editText1. With the layout completed, click on the toolbar Infer constraints button to add appropriate constraints:

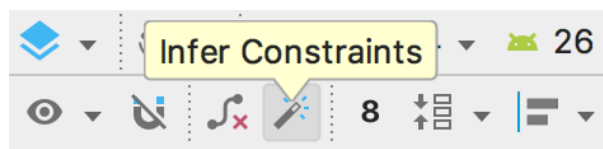


Figure 13-1

- Finally, click on the red warning button in the top right-hand corner of the Layout Editor window and use the resulting panel to extract the “Ask Question” string to a resource named ask_question.
- Once the layout is complete, the user interface should resemble that illustrated in Figure 13-2:

(continued)

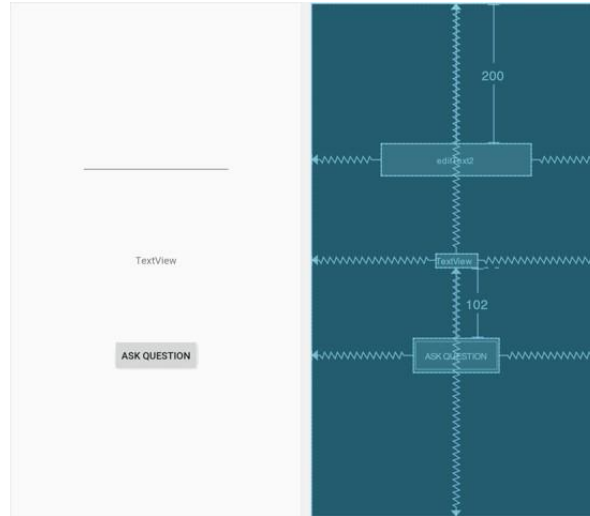


Figure 13-2

Creating the Second Activity Class

- When the “Ask Question” button is touched by the user, an intent will be issued requesting that a second activity be launched into which an answer can be entered by the user.
- The next step, therefore, is to create the second activity.
- Within the Project tool window, right-click on the edu.niu.your_Z-ID.explicitintent packagepackage name located in app > java and select the New > Activity > Empty Activity menu option to display the New Android Activity dialog as shown in Figure 13-3:

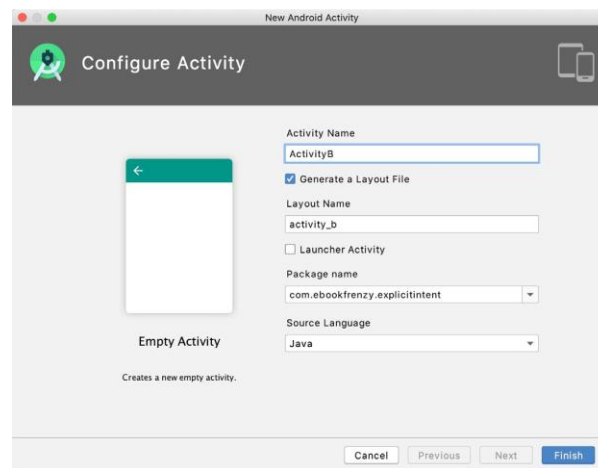


Figure 13-3

- Enter ActivityB into the Activity Name and Title fields and name the layout file activity_b and change the Language menu to Java.
- Since this activity will not be started when the application is launched (it will instead be launched via an intent by MainActivity when the button is pressed), it is important to make sure that the Launcher Activity option is disabled before clicking on the Finish button.

Designing the User Interface Layout for ActivityB

- The elements that are required for the user interface of the second activity are a Plain Text EditText, TextView and Button view.
- With these requirements in mind, load the activity_b.xml layout into the Layout Editor tool, and add the views.
- During the design process, note that the onClick property on the button view has been configured to call a method named answerQuestion, and the TextView and EditText views have been assigned IDs textView1 and editText1 respectively.
- Once completed, the layout should resemble that illustrated in Figure 13-4.
- Note that the text on the button (which reads “Answer Question”) has been extracted to a string resource named answer_question.
- With the layout complete, click on the Infer constraints toolbar button to add the necessary constraints to the layout:

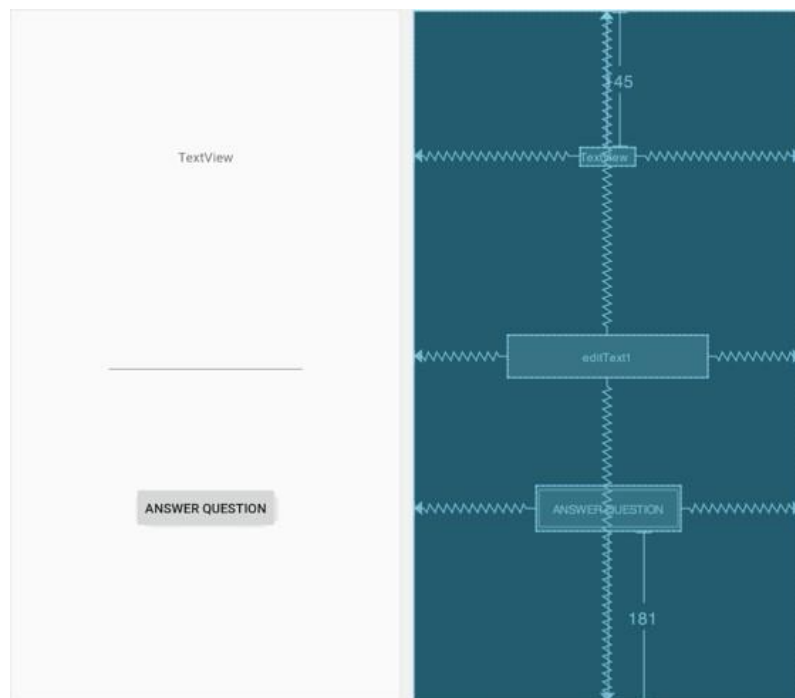


Figure 13-4

Reviewing the Application Manifest File

- In order for MainActivity to be able to launch ActivityB using an intent, it is necessary that an entry for ActivityB be present in the AndroidManifest.xml file.
- Locate this file within the Project tool window (app > manifests), double-click on it to load it into the editor and verify that Android Studio has automatically added an entry for the activity:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.niu.your_Z-ID.explicitintent">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true" android:theme="@style/AppTheme">
        <activity android:name=".ActivityB"></activity>
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

- With the second activity created and listed in the manifest file, it is now time to write some code in the MainActivity class to issue the intent.

Creating the Intent

- The objective for MainActivity is to create and start an intent when the user touches the “Ask Question” button.
- As part of the intent creation process, the question string entered by the user into the EditText view will be added to the intent object as a key-value pair.
- When the user interface layout was created for MainActivity, the button object was configured to call a method named askQuestion() when “clicked” by the user.
- This method now needs to be added to the MainActivity class MainActivity.java source file as follows:

```
package edu.niu.your_Z-ID.explicitintent;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
```

```
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_a);
}

public void askQuestion(View view)
{
    Intent i = new Intent(this, ActivityB.class);

    final EditText editText1 = findViewById(R.id.editText1);
    String myString = editText1.getText().toString();
    i.putExtra("qString", myString);
    startActivity(i);
}
}
```

- The code for the askQuestion() method follows the techniques outlined in “An Overview of Android Intents”.
- First, a new Intent instance is created, passing through the current activity and the class name of ActivityB as arguments.
- Next, the text entered into the EditText object is added to the intent object as a key-value pair and the intent started via a call to startActivity(), passing through the intent object as an argument.
- Compile and run the application and touch the “Ask Question” button to launch ActivityB and the back button (located in the toolbar along the bottom of the display) to return to MainActivity.

Extracting Intent Data

- Now that ActivityB is being launched from MainActivity, the next step is to extract the String data value included in the intent and assign it to the TextView object in the ActivityB user interface.
- This involves adding some code to the onCreate() method of ActivityB in the ActivityB.java source file:

```
package edu.niu.your_Z-ID.explicitintent;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;
import android.widget.TextView;
import android.widget.EditText;

public class ActivityB extends AppCompatActivity
{
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activityb);
    }
}
```

```
Bundle extras = getIntent().getExtras();

if (extras == null)
{
    return;
}

String qString = extras.getString("qString");

final TextView textView = findViewById(R.id.textView1);
textView.setText(qString);
}
}
```

- Compile and run the application either within an emulator or on a physical Android device.
- Enter a question into the text box in MainActivity before touching the “Ask Question” button.
- The question should now appear on the TextView component in the ActivityB user interface.

Launching ActivityB as a Sub-Activity

- In order for ActivityB to be able to return data to MainActivity, ActivityB must be started as a sub-activity of MainActivity.
- This means that the call to startActivity() in the MainActivity askQuestion() method needs to be replaced with a call to startActivityForResult().
- Unlike the startActivity() method, which takes only the intent object as an argument, startActivityForResult() requires that a request code also be passed through.
- The request code can be any number value and is used to identify which sub-activity is associated with which set of return data.
- For the purposes of this example, a request code of 5 will be used, giving us a modified MainActivity class that reads as follows:

```
public class MainActivity extends AppCompatActivity
{

    private static final int request_code = 5;

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

    public void askQuestion(View view)
```

```
{  
  
    Intent i = new Intent(this, ActivityB.class);  
  
    final EditText editText1 = findViewById(R.id.editText1);  
    String myString = editText1.getText().toString();  
    i.putExtra("qString", myString);  
    startActivity(i);  
    startActivityForResult(i, request_code);  
}  
}
```

- When the sub-activity exits, the `onActivityResult()` method of the parent activity is called and passed as arguments the request code associated with the intent, a result code indicating the success or otherwise of the sub-activity and an intent object containing any data returned by the sub-activity.
- Remaining within the `MainActivity` class source file, implement this method as follows:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)  
{  
    if ((requestCode == request_code) && (resultCode == RESULT_OK))  
    {  
        TextView textView1 = findViewById(R.id.textView1);  
        String returnString = data.getExtras().getString("returnData");  
  
        textView1.setText(returnString);  
    }  
}
```

- The code in the above method begins by checking that the request code matches the one used when the intent was issued and ensuring that the activity was successful.
- The return data is then extracted from the intent and displayed on the `TextView` object.

Returning Data From a Sub-Activity

- `ActivityB` is now launched as a sub-activity of `MainActivity`, which has, in turn, been modified to handle data returned from `ActivityB`.
- All that remains is to modify `ActivityB.java` to implement the `finish()` method and to add code for the `answerQuestion()` method, which is called when the “Answer Question” button is touched.
- The `finish()` method is triggered when an activity exits (for example when the user selects the back button on the device):

```
public void answerQuestion(View view)  
{  
    finish();  
}
```

```
@Override
public void finish()
{
    Intent data = new Intent();

    EditText editText1 = findViewById(R.id.editText1);

    String returnString = editText1.getText().toString();
    data.putExtra("returnData", returnString);

    setResult(RESULT_OK, data); super.finish();
}
```

- All that the finish() method needs to do is create a new intent, add the return data as a key-value pair and then call the setResult() method, passing through a result code and the intent object.
- The answerQuestion() method simply calls the finish() method.

Testing the Application

- Compile and run the application, enter a question into the text field on MainActivity and touch the “Ask Question” button.
- When ActivityB appears, enter the answer to the question and use either the back button or the “Submit Answer” button to return to MainActivity where the answer should appear in the text view object.

Android Implicit Intents – A Worked Example

- In this section, an example application will be created in Android Studio designed to demonstrate a practical implementation of implicit intents.
- The goal will be to create and send an intent requesting that the content of a particular web page be loaded and displayed to the user.
- Since the example application itself will not contain an activity capable of performing this task, an implicit intent will be issued so that the Android intent resolution algorithm can be engaged to identify and launch a suitable activity from another application.
- This is most likely to be an activity from the Chrome web browser bundled with the Android operating system.
- Having successfully launched the built-in browser, a new project will be created that also contains an activity capable of displaying web pages.
- This will be installed onto the device or emulator and used to demonstrate what happens when two activities match the criteria for an implicit intent.

(continued)

Creating the Android Studio Implicit Intent Example Project

- Select the Start a new Android Studio project quick start option from the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.
- Enter ImplicitIntent into the Name field and specify edu.niu.your_Z-ID.implicitintent as the package name.
- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

Designing the User Interface

- The user interface for the MainActivity class is very simple, consisting solely of a ConstraintLayout and a Button object.
- Within the Project tool window, locate the app > res > layout > activity_main.xml file and double-click on it to load it into the Layout Editor tool.
- Delete the default TextView and, with Autoconnect mode enabled, position a Button widget so that it is centered within the layout.
- Note that the text on the button (“Show Web Page”) has been extracted to a string resource named show_web_page.

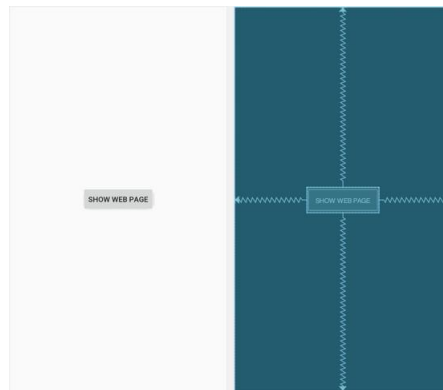


Figure 13-5

- With the Button selected use the Attributes tool window to configure the onClick property to call a method named showWebPage().

Creating the Implicit Intent

- As outlined above, the implicit intent will be created and issued from within a method named showWebPage() which, in turn, needs to be implemented in the MainActivity class, the code for which resides in the MainActivity.java source file.
- Locate this file in the Project tool window and double-click on it to load it into an editing pane. Once loaded, modify the code to add the showWebPage() method together with a few requisite imports:

```
package edu.niu.your_Z-ID.implicitintent;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.net.Uri;
import android.content.Intent;
import android.view.View;

public class MainActivity extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_implicit_intent);
    }

    public void showWebPage(View view)
    {
        Intent intent = new Intent(Intent.ACTION_VIEW,
                                   Uri.parse("https://www.ebookfrenzy.com"));

        startActivity(intent);
    }
}
```

- The tasks performed by this method are actually very simple.
- First, a new intent object is created. Instead of specifying the class name of the intent, however, the code simply indicates the nature of the intent (to display something to the user) using the ACTION_VIEW option.
- The intent object also includes a URI containing the URL to be displayed.
- This indicates to the Android intent resolution system that the activity is requesting that a web page be displayed.
- The intent is then issued via a call to the startActivity() method.
- Compile and run the application on either an emulator or a physical Android device and, once running, touch the Show Web Page button.
- When touched, a web browser view should appear and load the web page designated by the URL.
- A successful implicit intent has now been executed.

(continued)

Adding a Second Matching Activity

- The remainder of this chapter will be used to demonstrate the effect of the presence of more than one activity installed on the device matching the requirements for an implicit intent.
- To achieve this, a second application will be created and installed on the device or emulator.
- Begin, therefore, by creating a new project within Android Studio with the application name set to MyWebView, using the same SDK configuration options used when creating the ImplicitIntent project earlier in this chapter and once again selecting an Empty Activity.

Adding the Web View to the UI

- The user interface for the sole activity contained within the new MyWebView project is going to consist of an instance of the Android WebView widget.
- Within the Project tool window, locate the activity_main.xml file, which contains the user interface description for the activity, and double-click on it to load it into the Layout Editor tool.
- With the Layout Editor tool in Design mode, select the default TextView widget and remove it from the layout by using the keyboard delete key.
- Drag and drop a WebView object from the Widgets section of the palette onto the existing ConstraintLayout view as illustrated in Figure 13-6:

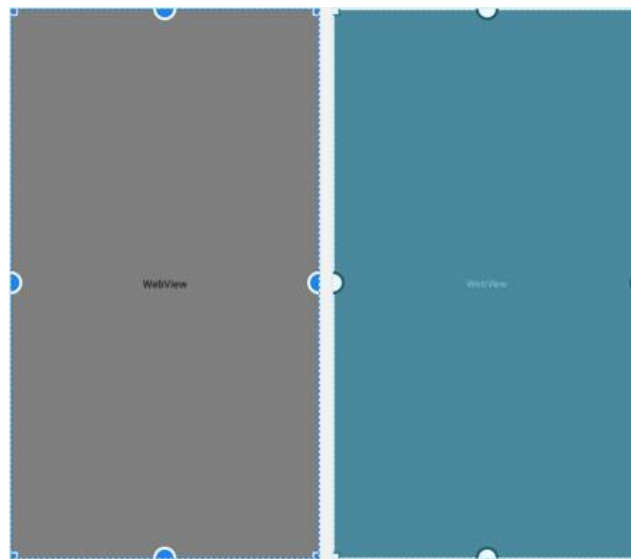


Figure 13-6

- Before continuing, change the ID of the WebView instance to webView1 and set the layout_width and layout_height properties to match_constraints.

Obtaining the Intent URL

- When the implicit intent object is created to display a web browser window, the URL of the web page to be displayed will be bundled into the intent object within a Uri object.

- The task of the onCreate() method within the MainActivity class is to extract this Uri from the intent object, convert it into a URL string and assign it to the WebView object.
- To implement this functionality, modify the MainActivity.java file so that it reads as follows:
package edu.niu.your_Z-ID.mywebview;

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import java.net.URL;
import android.net.Uri;
import android.content.Intent;
import android.webkit.WebView;

public class MainActivity extends AppCompatActivity
{

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my_web_view);

        handleIntent();
    }

    private void handleIntent()
    {
        Intent intent = getIntent();
        Uri data = intent.getData();
        URL url = null;

        try
        {
            url = new URL(data.getScheme(), data.getHost(), data.getPath());
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        WebView webView = findViewById(R.id.webView1);
        webView.loadUrl(url.toString());
    }
}
```

- The new code added to the onCreate() method performs the following tasks:
 - Obtains a reference to the intent which caused this activity to be launched
 - Extracts the Uri data from the intent object
 - Converts the Uri data to a URL object
 - Obtains a reference to the WebView object in the user interface

- Loads the URL into the web view, converting the URL to a String in the process
- The coding part of the MyWebView project is now complete.
- All that remains is to modify the manifest file.

Modifying the MyWebView Project Manifest File

- There are a number of changes that must be made to the MyWebView manifest file before it can be tested.
- In the first instance, the activity will need to seek permission to access the internet (since it will be required to load a web page).
- This is achieved by adding the appropriate permission line to the manifest file:

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

- Further, a review of the contents of the intent filter section of the AndroidManifest.xml file for the MyWebView project will reveal the following settings:

```
<intent-filter>  
    <action android:name="android.intent.action.MAIN" />  
    <category android:name="android.intent.category.LAUNCHER" />  
</intent-filter>
```

- In the above XML, the android.intent.action.MAIN entry indicates that this activity is the point of entry for the application when it is launched without any data input.
- The android.intent.category.LAUNCHER directive, on the other hand, indicates that the activity should be listed within the application launcher screen of the device.
- Since the activity is not required to be launched as the entry point to an application, cannot be run without data input (in this case a URL) and is not required to appear in the launcher, neither the MAIN nor LAUNCHER directives are required in the manifest file for this activity.
- The intent filter for the MainActivity activity does, however, need to be modified to indicate that it is capable of handling ACTION_VIEW intent actions for http data schemes.
- Android also requires that any activities capable of handling implicit intents that do not include MAIN and LAUNCHER entries, also include the so-called browsable and default categories in the intent filter.
- The modified intent filter section should therefore read as follows:

```
<intent-filter>  
    <action android:name="android.intent.action.VIEW" />  
    <category android:name="android.intent.category.BROWSABLE" />  
    <category android:name="android.intent.category.DEFAULT" />  
    <data android:scheme="http" android:host="www.ebookfrenzy.com" />  
</intent-filter>
```

- Note that the android:host entry above references a specific URL.
- Wildcards may also be used, for example to support all URLs:

```
<data android:scheme="http" android:host="*" />
```

- Bringing these requirements together results in the following complete AndroidManifest.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.niu.your_Z-ID.mywebview" >

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category
                    android:name="android.intent.category.BROWSABLE" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="https" android:host="www.ebookfrenzy.com"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

- Load the AndroidManifest.xml file into the manifest editor by double-clicking on the file name in the Project tool window.
- Once loaded, modify the XML to match the above changes.
- Having made the appropriate modifications to the manifest file, the new activity is ready to be installed on the device.

Installing the MyWebView Package on a Device

- Before the MyWebView main activity can be used as the recipient of an implicit intent, it must first be installed onto the device.
- This is achieved by running the application in the normal manner.

- Because the manifest file contains neither the `android.intent.action.MAIN` nor the `android.intent.category.LAUNCHER` settings, Android Studio needs to be instructed to install, but not launch, the app.
- To configure this behavior, select the app > Edit configurations... menu from the toolbar as illustrated in Figure 13-7:



Figure 13-7

- Within the Run/Debug Configurations dialog, change the Launch option located in the Launch Options section of the panel to Nothing and click on Apply followed by OK:



Figure 13-8

- With this setting configured run the app as usual. Note that the app is installed on the device, but not launched.

Testing the Application

- In order to test MyWebView, simply re-launch the `ImplicitIntent` application created earlier in this chapter and touch the Show Web Page button.
- This time, however, the intent resolution process will find two activities with intent filters matching the implicit intent.
- As such, the system will display a dialog (Figure 13-9) providing the user with the choice of activity to launch.

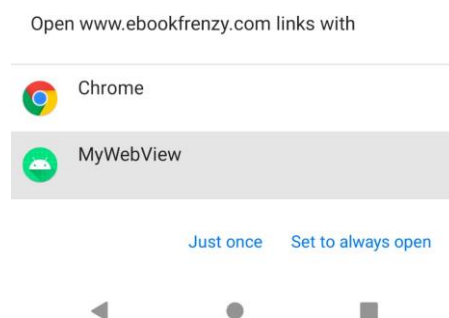


Figure 13-9

- Selecting the MyWebView option followed by the Just once button should cause the intent to be handled by our new MyWebView main activity, which will subsequently appear and display the designated web page.
- If the web page loads into the Chrome browser without the above selection dialog appearing, it may be that Chrome has been configured as the default browser on the device.
- This can be changed by going to Settings > Apps & notifications on the device followed by App info.
- Scroll down the list of apps and select Chrome.
- On the Chrome app info screen, tap the Open by default option followed by the Clear Defaults button.

Android Broadcast Intents and Broadcast Receivers

- In addition to providing a mechanism for launching application activities, intents are also used as a way to broadcast system wide messages to other components on the system.
- This involves the implementation of Broadcast Intents and Broadcast Receivers, both of which are the topic of this chapter.

An Overview of Broadcast Intents

- Broadcast intents are Intent objects that are broadcast via a call to the `sendBroadcast()`, `sendStickyBroadcast()` or `sendOrderedBroadcast()` methods of the Activity class (the latter being used when results are required from the broadcast).
- In addition to providing a messaging and event system between application components, broadcast intents are also used by the Android system to notify interested applications about key system events (such as the external power supply or headphones being connected or disconnected).
- When a broadcast intent is created, it must include an action string in addition to optional data and a category string.
- As with standard intents, data is added to a broadcast intent using key-value pairs in conjunction with the `putExtra()` method of the intent object.
- The optional category string may be assigned to a broadcast intent via a call to the `addCategory()` method.
- The action string, which identifies the broadcast event, must be unique and typically uses the application's package name syntax.
- For example, the following code fragment creates and sends a broadcast intent including a unique action string and data:

```
Intent intent = new Intent();  
intent.setAction("com.example.Broadcast");  
intent.putExtra("MyData", 1000);  
sendBroadcast(intent);
```

- The above code would successfully launch the corresponding broadcast receiver on a device running an Android version earlier than 3.0.
- On more recent versions of Android, however, the intent would not be received by the broadcast receiver.
- This is because Android 3.0 introduced a launch control security measure that prevents components of stopped applications from being launched via an intent.
- An application is considered to be in a stopped state if the application has either just been installed and not previously launched, or been manually stopped by the user using the application manager on the device.
- To get around this, however, a flag can be added to the intent before it is sent to indicate that the intent is to be allowed to start a component of a stopped application.
- This flag is `FLAG_INCLUDE_STOPPED_PACKAGES` and would be used as outlined in the following adaptation of the previous code fragment:

```
Intent intent = new Intent();  
intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);  
intent.setAction("com.example.Broadcast");  
intent.putExtra("MyData", 1000);  
sendBroadcast(intent);
```

An Overview of Broadcast Receivers

- An application listens for specific broadcast intents by registering a broadcast receiver.
- Broadcast receivers are implemented by extending the Android `BroadcastReceiver` class and overriding the `onReceive()` method.
- The broadcast receiver may then be registered, either within code (for example within an activity), or within a manifest file.
- Part of the registration implementation involves the creation of intent filters to indicate the specific broadcast intents the receiver is required to listen for.
- This is achieved by referencing the action string of the broadcast intent.
- When a matching broadcast is detected, the `onReceive()` method of the broadcast receiver is called, at which point the method has 5 seconds within which to perform any necessary tasks before returning.
- It is important to note that a broadcast receiver does not need to be running all the time.
- In the event that a matching intent is detected, the Android runtime system will automatically start up the broadcast receiver before calling the `onReceive()` method.
- The following code outlines a template Broadcast Receiver subclass:

```
package com.example.broadcastdetector;
```

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

public class MyReceiver extends BroadcastReceiver
{
    public MyReceiver()
    {
    }

    @Override
    public void onReceive(Context context, Intent intent)
    {
        // Implement code here to be performed when
        // broadcast is detected
    }
}
```

- When registering a broadcast receiver within a manifest file, a <receiver> entry must be added for the receiver.
- The following example manifest file registers the above example broadcast receiver:

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

    <uses-sdk android:minSdkVersion="17" />

    <application
        android:icon="@mipmap/ic_launcher" android:label="@string/app_name" >
        <receiver android:name="MyReceiver" >
        </receiver>
    </application>
</manifest>
```

- When running on versions of Android older than Android 8.0, the intent filters associated with a receiver can be placed within the receiver element of the manifest file as follows:

```
<receiver android:name="MyReceiver" >
    <intent-filter>
        <action android:name="com.example.Broadcast" >
        </action>
    </intent-filter>
</receiver>
```

- On Android 8.0 or later, the receiver must be registered in code using the registerReceiver() method of the Activity class together with an appropriately configured IntentFilter object:


```
IntentFilter filter = new IntentFilter("com.example.Broadcast");  
MyReceiver receiver = new MyReceiver();  
registerReceiver(receiver, filter);
```

- When a broadcast receiver registered in code is no longer required, it may be unregistered via a call to the `unregisterReceiver()` method of the activity class, passing through a reference to the receiver object as an argument.
- For example, the following code will unregister the above broadcast receiver:

```
unregisterReceiver(receiver);
```

- It is important to keep in mind that some system broadcast intents can only be detected by a broadcast receiver if it is registered in code rather than in the manifest file.
- Check the Android Intent class documentation for a detailed overview of the system broadcast intents and corresponding requirements online at:

<https://developer.android.com/reference/android/content/Intent>

Obtaining Results from a Broadcast

- When a broadcast intent is sent using the `sendBroadcast()` method, there is no way for the initiating activity to receive results from any broadcast receivers that pick up the broadcast.
- In the event that return results are required, it is necessary to use the `sendOrderedBroadcast()` method instead.
- When a broadcast intent is sent using this method, it is delivered in sequential order to each broadcast receiver with a registered interest.
- The `sendOrderedBroadcast()` method is called with a number of arguments including a reference to another broadcast receiver (known as the result receiver) which is to be notified when all other broadcast receivers have handled the intent, together with a set of data references into which those receivers can place result data.
- When all broadcast receivers have been given the opportunity to handle the broadcast, the `onReceive()` method of the result receiver is called and passed the result data.

Sticky Broadcast Intents

- By default, broadcast intents disappear once they have been sent and handled by any interested broadcast receivers.
- A broadcast intent can, however, be defined as being “sticky”.
- A sticky intent, and the data contained therein, remains present in the system after it has completed. The data stored within a sticky broadcast intent can be obtained via the return value of a call to the `registerReceiver()` method, using the usual arguments (references to the broadcast receiver and intent filter object).

- Many of the Android system broadcasts are sticky, a prime example being those broadcasts relating to battery level status.
- A sticky broadcast may be removed at any time via a call to the `removeStickyBroadcast()` method, passing through as an argument a reference to the broadcast intent to be removed.

The Broadcast Intent Example

- The remainder of this chapter will work through the creation of an Android Studio based example of broadcast intents in action.
- In the first instance, a simple application will be created for the purpose of issuing a custom broadcast intent.
- A corresponding broadcast receiver will then be created that will display a message on the display of the Android device when the broadcast is detected.
- Finally, the broadcast receiver will be modified to detect notification by the system that external power has been disconnected from the device.

Creating the Example Application

- Select the Start a new Android Studio project quick start option from the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.
- Enter `SendBroadcast` into the Name field and specify `edu.niu.your_Z-ID.sendbroadcast` as the package name.
- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.
- Once the new project has been created, locate and load the `activity_main.xml` layout file located in the Project tool window under `app > res > layout` and, with the Layout Editor tool in Design mode, replace the `TextView` object with a `Button` view and set the text property so that it reads “Send Broadcast”.
- Once the text value has been set, follow the usual steps to extract the string to a resource named `send_broadcast`.
- With the button still selected in the layout, locate the `onClick` property in the Attributes panel and configure it to call a method named `broadcastIntent`.

Creating and Sending the Broadcast Intent

- Having created the framework for the `SendBroadcast` application, it is now time to implement the code to send the broadcast intent.
- This involves implementing the `broadcastIntent()` method specified previously as the `onClick` target of the `Button` view in the user interface.

- Locate and double-click on the MainActivity.java file and modify it to add the code to create and send the broadcast intent.
- Once modified, the source code for this class should read as follows:

```
package edu.niu.your_Z-ID.sendbroadcast;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;

public class MainActivity extends AppCompatActivity
{

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_send_broadcast);
    }

    public void broadcastIntent(View view)
    {
        Intent intent = new Intent();
        intent.setAction("edu.niu.your_Z-ID.sendbroadcast");
        intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);
        sendBroadcast(intent);
    }
}
```

- Note that in this instance the action string for the intent is edu.niu.your_Z-ID.sendbroadcast.
- When the broadcast receiver class is created in later sections of this chapter, it is essential that the intent filter declaration match this action string.
- This concludes the creation of the application to send the broadcast intent.
- All that remains is to build a matching broadcast receiver.

Creating the Broadcast Receiver

- In order to create the broadcast receiver, a new class needs to be created which subclasses the BroadcastReceiver superclass.
- Within the Project tool window, navigate to app > java and right-click on the package name.
- From the resulting menu, select the New > Other > Broadcast Receiver menu option, name the class MyReceiver and make sure the Exported and Enabled options are selected.

- These settings allow the Android system to launch the receiver when needed and ensure that the class can receive messages sent by other applications on the device.
- With the class configured, click on Finish.
- Once created, Android Studio will automatically load the new MyReceiver.java class file into the editor where it should read as follows:

```
package edu.niu.your_Z-ID.sendbroadcast;

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

public class MyReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        // TODO: This method is called when the BroadcastReceiver
        // is receiving an Intent broadcast.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

- As can be seen in the code, Android Studio has generated a template for the new class and generated a stub for the onReceive() method.
- A number of changes now need to be made to the class to implement the required behavior. Remaining in the MyReceiver.java file, therefore, modify the code so that it reads as follows:

```
package edu.niu.your_Z-ID.sendbroadcast;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class MyReceiver extends BroadcastReceiver
{
    public MyReceiver()
    {
    }

    @Override
    public void onReceive(Context context, Intent intent)
    {
        // TODO: This method is called when the BroadcastReceiver
        // is receiving an Intent broadcast.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

```

        Toast.makeText(context, "Broadcast Intent Detected.",
                        Toast.LENGTH_LONG).show();
    }
}

```

- The code for the broadcast receiver is now complete.

Registering the Broadcast Receiver

- The project needs to publicize the presence of the broadcast receiver and must include an intent filter to specify the broadcast intents in which the receiver is interested.
- When the BroadcastReceiver class was created in the previous section, Android Studio automatically added a <receiver> element to the manifest file.
- All that remains, therefore, is to add code within the MainActivity.java file to create an intent filter and to register the receiver:

```

package edu.niu.your_Z-ID.sendbroadcast;
.
.
import android.content.BroadcastReceiver;
import android.content.IntentFilter;
.
.
public class MainActivity extends AppCompatActivity
{
    BroadcastReceiver receiver;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_send_broadcast);

        configureReceiver();
    }

    private void configureReceiver()
    {
        IntentFilter filter = new IntentFilter();
        filter.addAction("edu.niu.your_Z-ID.sendbroadcast");
        receiver = new MyReceiver();
        registerReceiver(receiver, filter);
    }
    .
    .
}

```

- It is also important to unregister the broadcast receiver when it is no longer needed:

```
@Override
protected void onDestroy()
{
    super.onDestroy();
    unregisterReceiver(receiver);
}
```

Testing the Broadcast Example

- In order to test the broadcast sender and receiver, run the SendBroadcast app on a device or AVD and wait for it to appear on the display.
- Once running, touch the button, at which point the toast message reading “Broadcast Intent Detected.” should pop up for a few seconds before fading away.

Listening for System Broadcasts

- The final stage of this example is to modify the intent filter for the broadcast receiver to listen also for the system intent that is broadcast when external power is disconnected from the device.
- That action is `android.intent.action.ACTION_POWER_DISCONNECTED`.
- Modify the `onCreate()` method in the `MainActivity.java` file to add this additional filter:

```
private void configureReceiver()
{
    IntentFilter filter = new IntentFilter();
    filter.addAction("edu.niu.your_Z-ID.sendbroadcast");
    filter.addAction("android.intent.action.ACTION_POWER_DISCONNECTED");
    receiver = new MyReceiver();
    registerReceiver(receiver, filter);
}
```

- Since the `onReceive()` method is now going to be listening for two types of broadcast intent, it is worthwhile to modify the code so that the action string of the current intent is also displayed in the toast message.
- This string can be obtained via a call to the `getAction()` method of the intent object passed as an argument to the `onReceive()` method:

```
public void onReceive(Context context, Intent intent)
{
    String message = "Broadcast intent detected " + intent.getAction();

    Toast.makeText(context, message, Toast.LENGTH_LONG).show();
}
```

- Test the receiver by re-installing the modified BroadcastReceiver package.

- Touching the button in the SendBroadcast application should now result in a new message containing the custom action string:

Broadcast intent detected edu.niu.your_Z-ID.sendbroadcast

- Next, remove the USB connector that is currently supplying power to the Android device, at which point the receiver should report the following in the toast message.
- If the app is running on an emulator, display the extended controls, select the Battery option and change the Charger connection setting to None.
- Broadcast intent detected android.intent.action.ACTION_POWER_DISCONNECTED
- To avoid this message appearing every time the device is disconnected from a power supply launch the Settings app on the device and select the Apps & notifications option.
- Select the BroadcastReceiver app from the resulting list and tap the Uninstall button.

Summary

- Intents are the messaging mechanism by which one Android activity can launch another.
- An explicit intent references a specific activity to be launched by referencing the receiving activity by class name.
- Explicit intents are typically, though not exclusively, used when launching activities contained within the same application.
- An implicit intent specifies the action to be performed and the type of data to be handled, and lets the Android runtime find a matching activity to launch.
- Implicit intents are generally used when launching activities that reside in different applications.
- An activity can send data to the receiving activity by bundling data into the intent object in the form of key-value pairs.
- Data can only be returned from an activity if it is started as a sub-activity of the sending activity.
- Activities advertise capabilities to the Android intent resolution process through the specification of intent-filters in the application manifest file.
- Both sending and receiving activities must also request appropriate permissions to perform tasks such as accessing the device contact database or the internet.
- Having covered the basics of intents before, we worked through the creation of an application project in Android Studio.
- The application was designed to demonstrate the use of explicit intents together with the concepts of data transfer between a parent activity and sub-activity.

- Implicit intents provide a mechanism by which one activity can request the service of another, simply by specifying an action type and, optionally, the data on which that action is to be performed.
- In order to be eligible as a target candidate for an implicit intent, however, an activity must be configured to extract the appropriate data from the inbound intent object and be included in a correctly configured manifest file, including appropriate permissions and intent filters.
- When more than one matching activity for an implicit intent is found during an intent resolution search, the user is prompted to make a choice as to which to use.
- Within this chapter an example was created to demonstrate both the issuing of an implicit intent, and the creation of an example activity capable of handling such an intent.
- Broadcast intents are a mechanism by which an intent can be issued for consumption by multiple components on an Android system.
- Broadcasts are detected by registering a Broadcast Receiver which, in turn, is configured to listen for intents that match particular action strings.
- In general, broadcast receivers remain dormant until woken up by the system when a matching intent is detected.
- Broadcast intents are also used by the Android system to issue notifications of events such as a low battery warning or the connection or disconnection of external power to the device.
- In addition to providing an overview of Broadcast intents and receivers, this chapter has also worked through an example of sending broadcast intents and the implementation of a broadcast receiver to listen for both custom and system broadcast intents.