MODULE II

UNDERSTANDING ANDROID RESOURCES

Resources play a key role in Android architecture. A resource in Android is a file or a value that is bound to an executable application. These files and values are bound to the executable in such a way that we can change them or provide alternatives without recompiling the application. Examples of resources include strings, colors, bitmaps, and layouts. Instead of hard-coding strings in an application, resources allow us to use their IDs instead. This indirection lets change the text of the string resource without changing the source code.

STRING RESOURCES

Android allows us to define strings in one or more XML resource files. These XML files containing string-resource definitions reside in the /res/values subdirectory. The names of the XML files are arbitrary, although we commonly see the file name as strings.xml. Listing 3–1 shows an example of a string-resource file.

Listing 3–1. *Example strings.xml*

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

Even the first line of the file indicating that it is an XML file with a certain encoding is optional. When this file is created or updated, the Eclipse ADT plug-in automatically creates or updates a Java class in our application's root package called R.java with unique IDs for the two string resources specified. Regardless of the number of resource files, there is only one R.java file. Notice the placement of this R.java file in the following example.

```
\MyProject
\src
\com\mycompany\android\my-root-package
\com\mycompany\android\my-root-package\another-package
\gen
\com\mycompany\android\my-root-package\R.java
\assets
\res
\AndroidManifest.xml
...etc
```

For the string-resource file in Listing 3–1, the updated R.java file has the entries in Listing 3–2.

Listing 3–2. *Example of R.java*

```
package com.mycompany.android.my-root-package;
public final class R {
...other entries depending on our project and application
public static final class string
{
...other entries depending on our project and application
public static final int hello=0x7f040000;
public static final int app_name=0x7f040001;
...other entries depending on our project and application
}
...other entries depending on our project and application
}
```

The two static final ints defined with variable names hello and app_name are the resource IDs that represent the corresponding string resources. We can use these resource IDs anywhere in the source code through the following code structure: R.string.hello

LAYOUT RESOURCES

In Android, the view for a screen is often loaded from an XML file as a resource. This is very similar to an HTML file describing the content and layout of a web page. These XML files are called layout resources. A *layout resource* is a key resource used in Android UI programming. Consider the code segment in Listing 3–4 for a sample Android activity.

Listing 3–4. *Using a Layout File*

```
public class HelloWorldActivity extends Activity
{
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
TextView tv = (TextView)this.findViewById(R.id.text1);
tv.setText("Try this text instead");
}
...
}
```

The line setContentView(R.layout.main) points out that there is a static class called R.layout, and within that class, there is a constant called main (an integer) pointing to a View defined by an XML layout resource file. The name of the XML file is main.xml, which needs to be placed in the resources' layout subdirectory. In other words, this statement expects the programmer to create the file /res/layout/main.xml and place the necessary layout definition in that file. The contents of the main.xml layout file could look like Listing 3–5.

Listing 3–5. *Example main.xml Layout File*

```
        <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        >
        <TextView android:id="@+id/text1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />
        <Button android:id="@+id/b1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />
</LinearLayout>
```

The layout file in Listing 3–5 defines a root node called LinearLayout, which contains a TextView followed by a Button. A LinearLayout lays out its children vertically or horizontally—vertically, in this example. We need to define a separate layout file for each screen (or activity). More accurately, each layout needs a dedicated file. If we are painting two screens, we probably need two layout files, such as /res/layout/screen1_layout.xml and /res/layout/screen2_layout.xml.

RESOURCE REFERENCE SYNTAX

All Android resources are identified (or referenced) by their IDs in Java source code. The syntax we use to allocate an ID to a resource in the XML file is called *resource reference syntax*. This syntax is not limited to allocating just ids: it is a way to identify any resource such as a string, a layout file, or an image. This resource reference has the following formal structure:

@[package:]type/name

The type corresponds to one of the resource-type namespaces available in R.java, some of which follow:

R.drawable	*R.id*	*R.layout*	*R.string*
R.attr	*R.plural*	*R.array*	

The corresponding types in XML resource-reference syntax are as follows:

Drawable *id* *layout* *stringattr* *plurals* *string-array*

The name part in the resource reference @[package:]type/name is the name given to the resource (for example, text1 in Listing 3–5); it also gets represented as an int constant in R.java. If we don't specify any package in the syntax @[package:]type/name, the pair type/name is resolved based on local resources and the application's local R.java package. If we specify android:type/name, the reference is resolved using the package android and specifically through the android.R.java file. We can use any Java package name in place of the package placeholder to locate the correct R.java file to resolve the reference.

DEFINING OWN RESOURCE IDS

It is possible to create IDs beforehand and use them later in our own packages. If they are resources, they should be allowed to be predefined and made available for later use. The solution is to use a resource tag called item to define an ID without attaching to any particular resource. Listing 3–8 shows an example.

Listing 3–8. *Predefining an ID*

```
<resources>
<item type="id" name="text"/>
</resources>
```

The type refers to the type of resource—id in this case. Once this ID is in place, the View definition in Listing 3–9 will work.

Listing 3–9. *Reusing a Predefined ID*

```
<TextView android:id="@id/text">
..
</TextView>
```

ENUMERATING KEY ANDROID RESOURCES

Let's enumerate some of the other key resources that Android supports, their XML representations, and the way they're used in Java code. To begin, take a quick glance at the types of resources and what they are used for in Table 3–1.

STRING ARRAYS

We can specify an array of strings as a resource in any file under the /res/values subdirectory. To do so, we use an XML node called string-array. This node is a child node of resources just like the string resource node. Listing 3–10 is an example of specifying an array in a resource file.

Listing 3–10. *Specifying String Arrays <resources*

```
.....................>
....Other resources
<string-array name="test_array">
<item>one</item> <item>two</item> <item>three</item> </string-
array>
.........................................................................................Other resources
</resources>
```

Once we have this string-array resource definition, we can retrieve this array in the Java code as shown in Listing 3–11.

Listing 3–11. *Specifying String Arrays*

```
//Get access to Resources object from an Activity Resources res
= our-activity.getResources();
String strings[] = res.getStringArray(R.array.test_array); //Print
strings
for (String s: strings)
{
Log.d("example", s);
```

PLURALS

The resource plurals is a set of strings. These strings are various ways of expressing a numerical quantity, such as how many eggs are in a nest. Consider an example:

There is 1 egg.
There are 2 eggs.
There are 0 eggs.
There are 100 eggs.

Notice how the sentences are identical for the numbers 2, 0, and 100.

However, the sentence for 1 egg is different. Android allows us to represent this variation as a plurals resource. Listing 3–12 shows how we would represent these two variations based on quantity in a resource file.

Listing 3–12. *Specifying String Arrays*

```
<resources...>
<plurals name="eggs_in_a_nest_text">
<item quantity="one">There is 1 egg</item>
<item quantity="other">There are %d eggs</item>
</plurals>
</resources>
```

The two variations are represented as two different strings under one plural. Now we can use the Java code in Listing 3–13 to use this plural resource to print a string given a quantity. The first parameter to the getQuantityString() method is the plurals resource ID. The second parameter selects the string to be used. When the value of the quantity is 1, we use the string as is. When the value is not 1, we must supply a third parameter whose value is to be placed where %d is. We must always have at least three parameters if we use a formatting string in our plurals resource. The second parameter can be confusing; the only distinction in this parameter is whether its value is 1 or other than 1.

Listing 3–13. *Specifying String Arrays*

```
Resources res = our-activity.getResources();
String s1 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 0,0);
String s2 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 1,1);
String s3 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 2,2);
String s4 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 10,10);
```

Given this code, each quantity results in an appropriate string that is suitable for its plurality. However, what other possibilities exist for the quantity attribute of the preceding item node? We strongly recommend that we read the source code of Resources.java and PluralRules.java in the Android source code distribution to truly understand this. Our research link in “Resources” at the end of this chapter has extracts from these source files.

The bottom line is that, for the en (English) locale, the only two possible values are "one" and "other". This is true for all other languages as well, except for cs (Czech), in which case the values are "one" (for 1), "few" (for 2 to 4), and "other" for the rest.

COLOR RESOURCES

As we can with string resources, we can use reference identifiers to indirectly reference colors. Doing this enables Android to localize colors and apply themes. Once we’ve defined and identified colors in resource files, we can access them in Java code through their IDs. Whereas string-resource IDs are available under the <*ourpackage*>. R.string namespace, the color IDs are available under the <*ourpackage*>. R.color namespace.

Android also defines a base set of colors in its own resource files. These IDs, by extension, are accessible through the Android android.R.color namespace. Check out this URL to learn the color constants available in the android.R.color namespace: Listing 3–17 has some examples of specifying color in an XML resource file.

Listing 3–17. *XML Syntax for Defining Color Resources*

```
<resources>
<color name="red">#f00</color>
<color name="blue">#0000ff</color>
<color name="green">#f0f0</color>
<color name="main_back_ground_color">#ffffff00</color>
</resources>
```

The entries in Listing 3–17 need to be in a file residing in the /res/values subdirectory. The name of the file is arbitrary, meaning the file name can be anything we choose. Android reads all the files and then processes them and looks for individual nodes such as resources and color to figure out individual IDs. Listing 3–18 shows an example of using a color resource in Java code.

Listing 3–18. *Color Resources in Java code*

```
int mainBackGroundColor
= activity.getResources.getColor(R.color.main_back_ground_color);
```

Listing 3–19 shows how we can use a color resource in a view definition.

Listing 3–19. *Using Colors in View Definitions*

```
<TextView android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="@color/red"
android:text="Sample Text to Show Red Color"/>
```

DIMENSION RESOURCES

Pixels, inches, and points are all examples of dimensions that can play a part in XML layouts or Java code. We can use these dimension resources to style and localize Android UIs without changing the source code. Listing 3–20 shows how we can use dimension resources in XML.

Listing 3–20. *XML Syntax for Defining Dimension Resources*

```
<resources>
<dimen name="mysize_in_pixels">1px</dimen>
<dimen name="mysize_in_dp">5dp</dimen>
<dimen name="medium_size">100sp</dimen>
</resources>
```

We can specify the dimensions in any of the following units:
px: Pixels
in: Inches
mm: Millimeters
pt: Points
dp: Density-independent pixels based on a 160dpi (pixel density per inch) screen (dimensions adjust to screen density)
sp: Scale-independent pixels (dimensions that allow for user sizing; helpful for use in fonts)

In Java, we need to access our Resources object instance to retrieve a dimension. We can do this by calling getResources on an activity object (see Listing 3–21). Once we have the Resources object, we can ask it to locate the dimension using the dimension ID (again, see Listing 3–21).

Listing 3–21. *Using Dimension Resources in Java Code*

```
float dimen = activity.getResources().getDimension(R.dimen.mysize_in_pixels);
```

As in Java, the resource reference for a dimension in XML uses dimen as opposed to thefull word *dimension* (see Listing 3–22).

Listing 3–22. *Using Dimension Resources in XML*

```
<TextView android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textSize="@dimen/medium_size"/>
```

IMAGE RESOURCES

Android generates resource IDs for image files placed in the /res/drawable subdirectory. The supported image types include .gif, .jpg, and .png. Each image file in this directory generates a unique ID from its base file name. If the image file name is sample_image.jpg, for example, then the resource ID generated is R.drawable.sample_image.

We can reference the images available in /res/drawable in other XML layout definitions, as shown in Listing 3–23.

Listing 3–23. *Using Image Resources in XML*

```
<Button
android:id="@+id/button1"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Dial"
android:background="@drawable/sample_image"
/>
```

We can also retrieve an image programmatically using Java and set it ourself against a UI object like a button (see Listing 3–24).

Listing 3–24. *Using Image Resources in Java*

```
//Call getDrawable to get the image
BitmapDrawable d = ctivity.getResources().getDrawable(R.drawable.sample_image);

//We can use the drawable then to set the background
button.setBackgroundDrawable(d);

//or we can set the background directly from the Resource Id
button.setBackgroundResource(R.drawable.sample_image);
```

Android also supports a special type of image called a *stretchable* image. This is a kind of .png where parts of the image can be specified as static and stretchable. Android provides a tool called the Draw 9-patch tool to specify these regions. Once the .png image is made available, we can use it like any other image. It comes in handy when used as a background for a button where the button has to stretch itself to accommodate the text.

UNDERSTANDING CONTENT PROVIDERS

Android uses a concept called *content providers* for abstracting data into services. A SQLite database on an Android device is an example of a data source that we can encapsulate into a content provider. To retrieve data from a content provider or save data into a content provider, we will need to use a set of REST-like URIs. For example, if we were to retrieve a set of books from a content provider that is an encapsulation of a book database, we would need to use a URI like this:

content://com.android.book.BookProvider/books

To retrieve a specific book from the book database (book 23), we would need to use a URI like this:

content://com.android.book.BookProvider/books/23

The content-provider abstraction is required only if we want to share data externally or between applications. For internal data access, an application can use any data storage/access mechanism that it deems suitable, such as the following:

- *Preferences*: A set of key/value pairs that we can persist to store application preferences
- *Files*: Files internal to applications, which we can store on a removable storage medium
- *SQLite*: SQLite databases, each of which is private to the package that creates that database

- *Network*: A mechanism that lets we retrieve or store data externally through the Internet via HTTP services

ANDROID BUILT IN PROVIDERS

Android comes with a number of built-in content providers, which are documented in the SDK's *android.provider* Java package. The providers include, for example, Contacts and Media Store. These SQLite databases typically have an extension of .db and are accessible only from the implementation package. Any access outside that package must go through the content-provider interface.

EXPLORING DATABASES ON EMULATOR

Because many content providers in Android use SQLite databases, we can use tools provided both by Android and by SQLite to examine the databases. Many of these tools reside in the \android-sdk-install-directory\tools subdirectory; others are in *android-sdk-install-directory**platform-tools.*
Android uses a command-line tool called Android Debug Bridge (adb), which is found here: *platform-tools**adb.exe*
adb is a special tool in the Android toolkit that most other tools go through to get to the device. However, we must have an emulator running or an Android device connected for *adb* to work. We can find out whether we have running devices or emulators by typing this at the command line:

```
adb devices
```

If the emulator is not running, we can start it by typing this at the command line:

```
emulator.exe @avdname
```

The argument *@avdname* is the name of an Android Virtual Device (AVD). To find out what virtual devices we already have, we can run the following command:

```
android list avd
```

This command will list the available AVDs. If we have developed and run any Android applications through Eclipse Android Development Tool (ADT), then we will have configured at least one virtual device. The preceding command will list at least that one. Here is some example output of that list command. (Depending on where our tools directory is and also depending on the Android release, the following printout may vary as to the path or release numbers, such as i:\android.)

We can also start the emulator through the Eclipse ADT plug-in. This automatically happens when we choose a program to run or debug in the emulator. Once the emulator is up and running, we can test again for a list of running devices by typing this:

```
adb devices
```

Now we should see a printout that looks like this:

```
List of devices attached
emulator-5554 device
```

We can see the many options and commands that we can run with adb by typing this at the command line:

```
adb help
```

We can use adb to open a shell on the connected device by typing this:

```
adb shell
```

We can see the available command set in the shell by typing this at the shell prompt: *#ls /system/bin*

The # sign is the prompt for the shell. For brevity, we will omit this prompt in the following examples. To see a list of root-level directories and files, we can type the following in the shell: *ls -l*

We'll need to access this directory to see the list of databases:

```
ls /data/data
```

This directory contains the list of installed packages on the device. Let's look at an example by exploring the com.android.providers.contacts package:

```
ls /data/data/com.android.providers.contacts/databases
```

This will list a database file called contacts.db, which is a SQLite database. (This file and path are still device and release dependent.) If there were a find command in the included ash, we could look at all the **.db* files. But there is no good way to do this with ls alone. The nearest thing we can do is this:

```
ls -R /data/data/*/databases
```

With this command, we will notice that the Android distribution has the following databases (again, a bit of caution; depending on our release, this list may vary):

```
alarms.db
contacts.db
downloads.db
internal.db
settings.db
mmssms.db
telephony.db
```

We can invoke sqlite3 on one of these databases inside the adb shell by typing this:

```
sqlite3 /data/data/com.android.providers.contacts/databases/contacts.db
```

We can exit sqlite3 by typing this:

```
sqlite>.exit
```

Notice that the prompt for adb is # and the prompt for sqlite3 is *sqlite>*. However, we will list a few important commands here so we don't have to make a trip to the Web. We can see a list of tables by typing sqlite> .tables. This command is a shortcut for:

```
SELECT name FROM sqlite_master WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%' UNION ALL

SELECT name FROM sqlite_temp_master WHERE type IN ('table','view') ORDER BY 1
```

As we probably guessed, the table *sqlite_master* is a master table that keeps track of tables and views in the database. The following command line prints out a create statement for a table called people in *contacts.db*: *.schema people*.

This is one way to get at the column names of a table in *SQLite*. This will also print out the column data types. While working with content providers, we should note these column types because access methods depend on them. However, it is pretty tedious to manually parse through this long create statement just to learn the column names and their types. There is a workaround: we can pull *contacts.db* down to our local box and then examine the database using any number of GUI tools for SQLite version 3. We can issue the following command from our OS command prompt to pull down the contacts.db file:

adb pull /data/data/com.android.providers.contacts/databases/contacts.db c:/somelocaldir/contacts.db

We used a free download of Sqliteman (http://sqliteman.com/), a GUI tool for SQLite databases, which seemed to work fine. We experienced a few crashes but otherwise found the tool completely usable for exploring Android SQLite databases.

Quick SQLite Primer

The following sample SQL statements could help we navigate through an SQLite database quickly:

```
//Set the column headers to show in the tool
sqlite>.headers on
//select all rows from a table
select * from table1;
//count the number of rows in a table
select count(*) from table1;
//select a specific set of columns
select col1, col2 from table1;
//Select distinct values in a column
select distinct col1 from table1;
//counting the distinct values
select count(col1) from (select distinct col1 from table1);
//group by
select count(*), col1 from table1 group by col1;
//regular inner join
select * from table1 t1, table2 t2 where t1.col1 = t2.col1;
//left outer join
//Give me everything in t1 even though there are no rows in t2
select * from table t1 left outer join table2 t2 on t1.col1 = t2.col1 where
```

....

ARCHITECTURE OF CONTENT PROVIDERS

The content-provider approach has parallels to the following industry abstractions:

- Web sites
- REST
- Web services
- Stored procedures

Each content provider on a device registers itself like a web site with a string (skin to a domain name, but called an *authority*). This uniquely identifiable string forms the basis of a set of URIs that this content provider can offer. This is not unlike how a web site with a domain offers a number of URLs to expose its documents or content in general. This authority registration occurs in the AndroidManifest.xml file. Here are two examples of how we can register providers in AndroidManifest.xml:

An authority is like a domain name for that content provider. Given the preceding authority registration, these providers will honor URLs starting with that authority prefix:

content://com.our-company.SomeProvider/
content://com.google.provider.NotePad/

We see that a content provider, like a web site, has a base domain name that acts as a starting URL. Content providers also provide REST-like URLs to retrieve or manipulate data. For the preceding registration, the URI to identify a directory or a collection of notes in the *NotePadProvider* database is

content://com.google.provider.NotePad/Notes

The URI to identify a specific note is

content://com.google.provider.NotePad/Notes/#

Where # is the id of a particular note. Here are some additional examples of URIs that some data providers accept:

content://media/internal/images
content://media/external/images
content://contacts/people/
content://contacts/people/23

Notice how these providers' media (content://media) and contacts (content://contacts) don't have a fully qualified structure. This is because these are not third-party providers and are controlled by Android. Content providers exhibit characteristics of web services as well. A content provider, through its URIs, exposes internal data as a service. However, the output from the URL of a content provider is not typed data, as is the case for a SOAP-based web-service call. This output is more like a result set coming from a JDBC statement. Even there, the similarities to JDBC are conceptual. We don't want to give the impression that this is the same as a JDBC ResultSet.

The caller is expected to know the structure of the rows and columns that are returned. Also, as we will see in this chapter's "Structure of Android MIME Types" section, a content provider has a built-in mechanism that allows we to determine the Multipurpose Internet Mail Extensions (MIME) type of the data represented by this URI. In addition to resembling web sites, REST, and web services, a content provider's URIs also resemble the names of stored procedures in a database. Stored procedures present service-based access to the underlying relational data. URIs is similar to stored procedures, because URI calls against a content provider return a cursor. However, content providers differ from stored procedures in that the input to a service call in a content provider is typically embedded in the URI itself.

STRUCTURE OF ANDROID CONTENT URIs

We compared a content provider to a web site because it responds to incoming URIs. So, to retrieve data from a content provider, all we have to do is invoke a URI. The retrieved data in the case of a content provider, however, is in the form of a set of rows and columns represented by an Android cursor object. In this context, we'll examine the structure of the URIs that we could use to retrieve data. Content URIs in Android look similar to HTTP URIs, except that they start with content and have the general form

content:///*/** or

content://authority-name/path-segment1/path-segment2/etc…

Here's an example URI that identifies a note numbered 23 in a database of notes:

content://com.google.provider.NotePad/notes/23

After content:, the URI contains a unique identifier for the authority, which is used to locate the provider in the provider registry. In the preceding example, *com.google.provider.NotePad* is the authority portion of the URI. /notes/23 is the path section of the URI that is specific to each provider. The notes and 23 portions of the path section are called *path segments*. It is the responsibility of the provider to document and interpret the path section and path segments of the URIs.

The developer of the content provider usually does this by declaring constants in a Java class or a Java interface in that provider's implementation Java package. Furthermore, the first portion of the path might point to a collection of objects. For example, /notes indicates a collection or a directory of notes, whereas /23 points to a specific note item. Given this URI, a provider is expected to retrieve rows that the URI identifies. The provider is also expected to alter content at this URI using any of the state-change methods: insert, update, or delete.

READING DATA USING URIs

Now we know that to retrieve data from a content provider, we need to use URIs supplied by that content provider. Because the URIs defined by a content provider are unique to that provider, it is important that these URIs are documented and available to programmers to see and then call. The providers that come with Android do this by defining constants representing these URI strings. Consider these three URIs defined by helper classes in the Android SDK:

MediaStore.Images.Media.INTERNAL_CONTENT_URI
MediaStore.Images.Media.EXTERNAL_CONTENT_URI
ContactsContract.Contacts.CONTENT_URI

The equivalent textual URI strings would be as follows:

content://media/internal/images
content://media/external/images
content://com.android.contacts/contacts/

Given these URIs, the code to retrieve a single row of people from the Contacts provider looks like this:

```
        Uri peopleBaseUri = ContactsContract.Contacts.CONTENT_URI;
        Uri myPersonUri = Uri.withAppendedPath(peopleBaseUri, "23");
//Query for this record.
//managedQuery is a method on Activity class
        Cursor cur = managedQuery(myPersonUri, null, null, null);
```

Notice how the ContactsContract.Contacts.CONTENT_URI is predefined as a constant in the Contacts class. We have named the variable peopleBaseUri to indicate that if our intention is to discover people, we go after the Contacts content URI. Of course, we can call this variable contactsBaseUri if we conceptually think of people as contacts.

In this example, the code takes the root URI, adds a specific person ID to it, and makes a call to the managedQuery method. As part of the query against this URI, it is possible to specify a sort order, the columns to select, and a where clause. These additional parameters are set to null in this example. A content provider should list which columns it supports by implementing a set of interfaces or by listing the column names as constants. However, the class or interface that defines constants for columns should also make the column types clear through a column-naming convention, or comments or documentation, because there is no formal way to indicate the type of a column through constants.

USING ANDROID CURSOR

Here are a few facts about an Android cursor:

- A cursor is a collection of rows.
- We need to use *moveToFirst()* before reading any data because the cursor starts off positioned before the first row.
- We need to know the column names.

- We need to know the column types.
- All field-access methods are based on column number, so we must convert the column name to a column number first.
- The cursor is random (we can move forward, backward and jump). Because the cursor is random, we can ask it for a row count.

An Android cursor has a number of methods that allow we to navigate through it. Listing 4–2 shows how to check if a cursor is empty and how to walk through the cursor row by row when it is not empty.

Listing 4–2. *Navigating Through a Cursor Using a while Loop*

```
if (cur.moveToFirst() == false)
{
//no rows empty cursor
return;
}
//The cursor is already pointing to the first row
//let's access a few columns
int nameColumnIndex = cur.getColumnIndex(Contacts.DISPLAY_NAME_PRIMARY);
String name = cur.getString(nameColumnIndex);
//let's now see how we can loop through a cursor
while(cur.moveToNext())
{
//cursor moved successfully
//access fields
}
```

The assumption at the beginning of Listing 4–2 is that the cursor has been positioned before the first row. To position the cursor on the first row, we use the *moveToFirst()* method on the cursor object. This method returns false if the cursor is empty. We then use the *moveToNext()* method repetitively to walk through the cursor. To help we learn where the cursor is, Android provides the following methods:

isBeforeFirst()
isAfterLast()
isClosed()

Using these methods, we can also use a for loop as in Listing 4–3 to navigate through the cursor instead of the while loop used in Listing 4–2.

Listing 4–3. *Navigating Through a Cursor Using a for Loop*

```
//Get our indexes first outside the for loop
int nameColumn = cur.getColumnIndex(Contacts.DISPLAY_NAME_PRIMARY);
//Walk the cursor now based on column indexes
for(cur.moveToFirst();!cur.isAfterLast();cur.moveToNext())
{
String name = cur.getString(nameColumn);
}
```

The index order of columns seems to be a bit arbitrary. As a result, we advise we to explicitly get the indexes first from the cursor to avoid surprises. To find the number of rows in a cursor, Android provides a method on the cursor object called *getCount().*

WORKING WITH WHERE CLAUSE

Content providers offer two ways of passing a where clause:

- Through the URI
- Through the combination of a string clause and a set of replaceable string-array arguments

1. **Passing a where Clause Through a URI**

Imagine we want to retrieve a note whose ID is 23 from the Google notes database. We'd use the code in Listing 4–4 to retrieve a cursor containing one row corresponding to row 23 in the notes table.

Listing 4–4. *Passing SQL where Clauses Through the URI*

```
Activity someActivity;
//..initialize someActivity
String noteUri = "content://com.google.provider.NotePad/notes/23";
Cursor managedCursor = someActivity.managedQuery( noteUri,
projection, //Which columns to return.
null, // WHERE clause
null); // Order-by clause.
```

We left the where clause argument of the managedQuery method null because, in this case, we assumed that the note provider is smart enough to figure out the id of the book we wanted. This id is embedded in the URI itself. We used the URI as a vehicle to pass the where clause. This becomes apparent when we notice how the notes provider implements the corresponding query method. Here is a code snippet from that query method:

```
//Retrieve a note id from the incoming uri that looks like
//content://.../notes/23
int noteId = uri.getPathSegments().get(1);
//ask a query builder to build a query
//specify a table name
queryBuilder.setTables(NOTES_TABLE_NAME);
//use the noteid to put a where clause
queryBuilder.appendWhere(Notes._ID + "=" + noteId);
```

Notice how the ID of a note is extracted from the URI. The Uri class representing the incoming argument uri has a method to extract the portions of a URI after the root *content://com.google.provider.NotePad*. These portions are called *path segments*; they're strings between / separators such as */seg1/seg3/seg4/,* and they're indexed by their positions. For the URI here, the first path segment would be 23. We then used this ID of 23 to append to the where clause specified to the *QueryBuilder* class. In the end, the equivalent select statement would be

*select * from notes where _id = 23*

2. Using Explicit where Clauses

Now that we have seen how to use a URI to send in a where clause, consider the other method by which Android lets us send a list of explicit columns and their corresponding values as a where clause. To explore this, let's take another look at the *managedQuery* method of the Activity class that we used in **Listing 4–4**. Here's its signature:

public final Cursor managedQuery(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)

Notice the argument named selection, which is of type String. This selection string represents a filter (a where clause, essentially) declaring which rows to return, formatted as a SQL where clause (excluding the WHERE itself). Passing null will return all rows for the given URI. In the selection string we can include ?s, which will be replaced by the values from selectionArgs in the order that they appear in the selection. The values will be bound as Strings. Because we have two ways of specifying a where clause, we might find it difficult to determine how a provider has used these where clauses and which where clause takes precedence if both where clauses are utilized. For example, we can query for a note whose ID is 23 using either of these two methods:

```
//URI method
managedQuery("content://com.google.provider.NotePad/notes/23"
,null
,null
,null
,null);
```

or

```
//explicit where clause
managedQuery("content://com.google.provider.NotePad/notes"
,null
,"_id=?"
,new String[] {23}
,null);
```

The convention is to use where clauses through URIs where applicable and use the explicit option as a special case.

INSERTING RECORDS

Android uses a class called *android.content.ContentValues* to hold the values for a single record that is to be inserted. *ContentValues* is a dictionary of key/value pairs, much like column names and their values. We insert records by first populating a record into *ContentValues* and then asking *android.content.ContentResolver* to insert that record using a URI. Here is an example of populating a single row of notes in ContentValues in preparation for an insert:

```
ContentValues values = new ContentValues();
values.put("title", "New note");
values.put("note","This is a new note");
```

//values object is now ready to be inserted
We can get a reference to ContentResolver by asking the Activity class:

```
ContentResolver contentResolver = activity.getContentResolver();
```

Now, all we need is a URI to tell ContentResolver to insert the row. These URIs are defined in a class corresponding to the Notes table. In the Notepad example, this URI is

Notepad.Notes.CONTENT_URI

We can take this URI and the ContentValues we have and make a call to insert the row:

```
Uri uri = contentResolver.insert(Notepad.Notes.CONTENT_URI, values);
```

This call returns a URI pointing to the newly inserted record. This returned URI would match the following structure:

Notepad.Notes.CONTENT_URI/new_id

UPDATING AND RELETING RECORDS

We have talked about queries and inserts; updates and deletes are fairly straightforward. Performing an update is similar to performing an insert, in which changed column values are passed through a ContentValues object. Here is the signature of an update method on the ContentResolver object:

```
int numberOfRowsUpdated =activity.getContentResolver().update(
Uri uri, ContentValues values, String whereClause, String[] selectionArgs )
```

The *whereClause* argument constrains the update to the pertinent rows. Similarly, the signature for the delete method is

```
int numberOfRowsDeleted = activity.getContentResolver().delete(
Uri uri, String whereClause, String[] selectionArgs )
```

Clearly, a delete method will not require the *ContentValues* argument because we will not need to specify the columns we want when we are deleting a record. Almost all the calls from *managedQuery* and *ContentResolver* are directed eventually to the provider class. Knowing how a provider implements each of these methods gives us enough clues as to how those methods are used by a client. In the next section, we'll cover from scratch the implementation of an example content provider: called *BookProvider.*

IMPLEMENTING CONTENT PROVIDERS

We've discussed how to interact with a content provider for data needs but haven't yet discussed how to write a content provider. To write a content provider, we have to extend android.content.ContentProvider and implement the following key methods:

query
insert
update
delete
getType

We'll also need to set up a number of things before implementing them. We will illustrate all the details of a content-provider implementation by describing the steps we'll need to take:

1) Plan our database, URIs, column names, and so on, and create a metadata class that defines constants for all of these metadata elements.
2) Extend the abstract class ContentProvider.
3) Implement these methods: query, insert, update, delete, and getType.
4) Register the provider in the manifest file.

UNDERSTANDING INTENTS

Android introduced a concept called *intents* to invoke components. The list of components in Android includes activities (UI components), services (background code), broadcast receivers (code that responds to broadcast messages), and content providers (code that abstracts data).

BASICS OF INTENTS

Intent is easily understood as a mechanism to invoke components, Android folds multiple ideas into the concept of intent. We can use intents to invoke external applications from our application. We can use intents to invoke internal or external components from our application. We can use intents to raise events so that others can respond in a manner similar to a publish-and-subscribe model. We can use intents to raise alarms. The short answer may be that intent is an action with its associated data payload. An action that we can tell Android to perform (or *invoke*). The action Android invokes depends on what is registered for that action.

Following is the example of an Activity

```
{
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
setContentView(R.layout.some_view);
}
}//eof-class
```

Android then allows we to register this activity in the manifest file of that application, making it available for other applications to invoke. The registration looks like this:

```
<activity android:name=".BasicViewActivity"
android:label="Basic View Tests">
<intent-filter>
<action android:name="com.androidbook.intent.action.ShowBasicView"/>
<category android:name="android.intent.category.DEFAULT" />
</intent-filter>
</activity>
```

Now that we have specified the activity and its registration against an action, we can use an intent to invoke this BasicViewActivity:

```
public static void invokeMyApplication(Activity parentActivity)
{
String actionName= "com.androidbook.intent.action.ShowBasicView";
Intent intent = new Intent(actionName);
parentActivity.startActivity(intent);
}
```

AVAILABLE INTENTS IN ANDROID

We can give intents a test run by invoking some of the applications that come with Android. This list may change depending on the Android release. The set of available applications could include the following:

1) A browser application to open a browser window
2) An application to call a telephone number
3) An application to present a phone dialer so the user can enter the numbers and make a call through the UI
4) A mapping application to show the map of the world at a given latitude and longitude coordinate
5) A detailed mapping application that can show Google street views

Listing 5–1. *Exercising Android's Prefabricated Applications*

```
public class IntentsUtils
{
public static void invokeWebBrowser(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http:// www.google.com"));
activity.startActivity(intent);
}
```

```
public static void invokeWebSearch(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_WEB_SEARCH);
intent.setData(Uri.parse("http:// www.google.com"));
activity.startActivity(intent);
}
public static void dial(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_DIAL);
activity.startActivity(intent);
}
public static void call(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel:555–555–5555"));
activity.startActivity(intent);
}
public static void showMapAtLatLong(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_VIEW);
//geo:lat,long?z=zoomlevel&q=question-string
intent.setData(Uri.parse("geo:0,0?z=4&q=business+near+city"));
activity.startActivity(intent);
}
public static void tryOneOfThese(Activity activity)
{
IntentsUtils.invokeWebBrowser(activity);
```

EXPLORING INTENT COMPOSITION

An intent has an action, data (represented by a data URI), a key/value map of extra data elements, and an explicit class name (called a *component name*). Almost all of these are optional as long as the intent carries at least one of these. We will explore each of these parts in turn. When an intent carries a component name with it, it is called an *explicit* intent. When an intent doesn't carry a component name but relies on other parts such as action and data, it is called an *implicit* intent.

1. Data URI
2. Generic Action
3. Using Extra information
4. Using Components To Directly Invoke An Activity

1)Data URI

We've covered the simplest of the intents, where all we need is the name of an action. The ACTION_DIAL activity in Listing 5–1 is one of these; to invoke the dialer, all we needed in that listing is the dialer's action and nothing else:

```
public static void dial(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_DIAL);
activity.startActivity(intent);
}
```

Unlike ACTION_DIAL, the intent ACTION_CALL (again referring to Listing 5–1) that is used to make a call to a given phone number takes an additional parameter called Data. This parameter points to a URI, which, in turn, points to the phone number:

```
public static void call(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel:555–555–5555"));
activity.startActivity(intent);
}
```

The action portion of an intent is a string or a string constant, usually prefixed by the Java package name. The data portion of an intent is not really data but a pointer to the data. This data portion is a string representing a URI. An intent's URI can contain arguments that can be inferred as data, just like a web site's URL. The format of this URI could be specific to each activity that is invoked by that action. In this case, the CALL action decides what kind of data URI it would expect. From the URI, it extracts the telephone number. The invoked activity can also use the URI as a pointer to a data source, extract the data from the data source, and use that data instead. This would be the case for media such as audio, video, and images.

2)Generic Action

The actions Intent.ACTION_CALL and Intent.ACTION_DIAL could easily lead us to the wrong assumption that there is a one-to-one relationship between an action and what it invokes. To disprove this, let's consider a counterexample from the IntentUtils code in Listing 5–1:

```
public static void invokeWebBrowser(Activity activity)
{
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http:// www.google.com"));
activity.startActivity(intent);
}
```

Note that the action is simply stated as ACTION_VIEW. How does Android know which activity to invoke in response to such a generic action name? In these cases, Android relies not only on the generic action name but also on the nature of the URI. Android looks at the scheme of the URI, which happens to be http, and questions all the registered activities to see which ones understand this scheme. Out of these, it inquires which ones can handle the VIEW and then invokes that activity. For this to work, the browser activity should have registered a VIEW intent against the data scheme of http.

3)Using Extra information

An intent can include an additional attribute called *extras*. An extra can provide more information to the component that receives the intent. The extra data is in the form of key/value pairs: the key name typically starts with the package name, and the value can be any fundamental data type or arbitrary object as long as it implements the android.os.Parcelable interface. This extra information is represented by an Android class called android.os.Bundle. The following two methods on an Intent class provide access to the extra Bundle:

```
//Get the Bundle from an Intent
    Bundle extraBundle = intent.getExtras();
// Place a bundle in an intent
    Bundle anotherBundle = new Bundle();
//populate the bundle with key/value pairs
    …
//and then set the bundle on the Intent
    intent.putExtras(anotherBundle);
```

getExtras is straightforward: it returns the Bundle that the intent has. *putExtras* checks whether the intent currently has a bundle. If the intent already has a bundle, *putExtras* transfers the additional keys and values from the new bundle to the existing bundle. If the bundle doesn't exist, *putExtras* will create one and copy the key/value pairs from the new bundle to the created bundle.

4)Using Components To Directly Invoke An Activity

Android also provides a more direct way to start an activity: we can specify the activity's ComponentName, which is an abstraction around an object's package name and class name. There are a number of methods available on the Intent class to specify a component:

```
    setComponent(ComponentName name);
    setClassName(String packageName, String classNameInThatPackage);
    setClassName(Context context, String classNameInThatContext);
    setClass(Context context, Class classObjectInThatContext);
```

Ultimately, they are all shortcuts for calling one method:

```
    setComponent(ComponentName name);
```

ComponentName wraps a package name and a class name together. For example, the following code invokes the contacts activity that ships with the emulator:

```
Intent intent = new Intent();
intent.setComponent(new ComponentName(
"com.android.contacts"
,"com.android.contacts.DialContactsEntryActivity");
startActivity(intent);
```

RULES FOR RESOLVING INTENTS TO THEIR COMPONENTS

At the top of the hierarchy is the component name attached to an intent. If this is set, the intent is known as an *explicit intent*. For an explicit intent, only the component name matters; every other aspect or attribute of the intent is ignored. When a component name is not present on an intent, the intent is said to be an *implicit intent*. The rules for resolving targets for implicit intents are numerous. The basic rule is that an incoming intent's action, category, and data characteristics *must match* (or present) those specified in the intent filter. An intent filter, unlike an intent, can specify multiple actions, categories, and data attributes. This means the same intent filter can satisfy multiple intents, which is to say that an activity can respond to many intents. However, the meaning of "match" differs among actions, data attributes, and categories. Let's look the matching criteria for each of the parts of an implicit intent.

Action

If an intent has an action on it, the intent filter must have that action as part of its action list or not have any actions at all. So if an intent filter *doesn't define an action*, that intent filter *is a match* for any incoming intent action. If one or more actions are specified in the intent filter, at least one of the actions must match the incoming intent's action.

Data

If no data characteristics are specified in an intent filter, it does not match an incoming intent that carries any data or data attribute. This means it will only look for intents that have no data specified at all. Lack of data and lack of action (in the filter) work the opposite. If there is no action in the filter, every thing is a match. If there is no data in the filter, every bit of data in the intent is a mismatch.

Data Type

For a data type to match, the incoming intent's data type must be one of the data types that is specified in the intent filter. The data type in the intent must be present in the intent filter. The incoming intent's data type is determined in one of two ways. First, if the data URI is a content or file URI, the content provider or Android will figure out the type. The second way is to look at the explicit data type of the intent. For this to work, the incoming intent should not have a data URI set, because this is automatically taken care of when setType is called on the intent. Android also allows its MIME type specification to have an asterisk (*) as its subtype to cover all possible subtypes. Also, the data type is case sensitive.

Data Scheme

For a data scheme to match, the incoming intent data scheme must be one of those specified in the intent filter. In other words, the incoming data scheme must be present in the intent filter. The incoming intent's scheme is the first part of the data URI. On an intent, there is no method to set the scheme. It is purely derived from the intent data URI that looks like
http://www.somesite.com/somepath.

If the data scheme of the incoming intent URI is content: or file:, it is considered a match regardless of the intent filter scheme, domain, and path. According to the SDK, this is so because every component is expected to know how to read data from content or file URLs, which are essentially local. In other words, all components are expected to support these two types of URLs. The scheme is also case sensitive.

Data Authority

If there are no authorities in the filter, we have a match for any incoming d ta URI authority (or domain name). If an authority is specified in the filter—for example, www.somesite.com—then one scheme and one authority should match the incoming intent's data URI. For example, if we specify www.somesite.com as the authority in the intent filter and the scheme as https, the intent will fail to match http://www.somesite.com/somepath because http is not indicated as the supporting scheme. The authority is case sensitive as well.

Data Path

No data paths in the intent filter means a match for any incoming data URI's path. If a path is specified in the filter—for example, somepath—one scheme, one authority, and one data path should match the incoming intent's data URI. In other words scheme, authority, and path work together to validate an incoming intent URI such as http://www.somesite.com/somepath. So path, authority, and scheme work not in isolation but together. The path, too, is case sensitive.

ACTION_PICK

ACTION_PICK is one such generic action. In ACTION_PICK, we are specifying a URI that points to a collection of items, such as a collection of notes. We will expect the action to pick one of the notes and return it to the caller. The idea of ACTION_PICK is to start an activity that displays a list of items. The activity then should allow a user to pick one item from that list. Once the user picks the item, the activity should return the URI of the picked item to the caller. This allows reuse of the UI's functionality to select items of a certain type. We cannot use *startActivity()*, because *startActivity*() does not return a result. *startActivity*() cannot return a result, because it opens the new activity as a modal dialog in a separate thread and leaves the main thread for attending events. If we want to return data, we can use a variation of *startActivity*() called *startActivityForResult*(), which comes with a callback.

GET_CONTENT

ACTION_GET_CONTENT is similar to ACTION_PICK. In the case of ACTION_PICK, we are specifying a URI that points to a collection of items, such as a collection of notes. We will expect the action to pick one of the notes and return it to the caller. In the case of ACTION_GET_CONTENT, we indicate to Android that we need an item of a particular MIME type. Android searches for either activities that can create one of those items or activities that can choose from an existing set of items that satisfy that MIME type.

PENDING INTENTS

Android has a variation on an intent called a *pending intent*. In this variation, Android allows a component to store an intent for future use in a location from which it can be invoked again. For example, in an alarm manager, we want to start a service when the alarm goes off.