Chapter 5 File I/O and JNI

Since Android uses Java as its programming language, it inherits plenty of the Java classes for accessing files and I/O resources, though many of the Java I/O fields and methods are not needed in practice and might have made the language less attractive. In addition, Android defines other means of specifying and accessing resources. We have seen that Android generates the resource directory *res*, which contains other files and subdirectories to define layouts, resources, assets, and parameters, often in *xml* format. The URL

http://developer.android.com/reference/android/content/Context.html

that describes the functionalities of the class *Context*, presents a lot of methods for accessing I/O resources.

Android also supports JNI (Java Native Interface) for Java programs to interact with native code written in C/C++. JNI is vendor-neutral and has support for loading code from dynamic shared libraries, which can be efficient.

5.1 Read Raw Data From File

If we do not want to specify our data in the *xml* format but to keep them as raw data, we can create our own subdirectories and files under the *res* directory. Let us consider a very simple example to illustrate the technique. Suppose in our example, we want to read in the data saved in a file, say *res/raw/hello*, as unformated strings. We call our project and application *ReadRaw*, and the package, *data.readraw*. Suppose like what we did before, we have used the defaults of the Eclipse Android settings to create the project. We continue to do the following in Eclipse IDE:

1. **Create Directory** *res/raw*: Click **File** > **New** > **Folder**. Select the **parent folder** to be *ReadRaw/res* and enter *raw* for **Folder name**. Then click **Finish**, which creates the directory *res/raw*.
2. **Create File** *res/raw/hello*: Click **File** > **New** > **File**. Select the **parent folder** to be *ReadRaw/res/raw* and enter *hello* for **File name**. Then click **Finish**, which creates the file *res/raw/hello*. You may enter any text in *hello*. For example, we enter *Android The Beautiful!* in the file and save it.
3. **Modify** *MainActivity.java*: Modify the main program *MainActivity.java* to the following code, which simply reads in the data in the file *res/raw/hello* and prints it out as a string.

```
--------------------------------------------------------------------
package data.readraw;

import java.io.IOException;
import java.io.InputStream;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.os.Build;

public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    String str = null;
    InputStream inputStream = null;
    try {
      inputStream = getResources().openRawResource(R.raw.hello);
      byte[] reader = new byte[inputStream.available()];
      while (inputStream.read(reader) != -1) {}
      str = new String ( reader );
      System.out.println ( str );
    }  catch(IOException e) {
      Log.e(Tag", e.getMessage());
    }
  }
}
--------------------------------------------------------------------
```

In the code, **getResources**() is an abstract method declared in the class *Context*. It returns a *Resources* instance for the application's package. Note that our class *MainActivity* extends *Activity*, which extends the class *ContextThemeWrapper*; one of the intermediate classes in the class hierarchy *android.view.ContextThemeWrapper* provides an implementation for the method **getResources**. The returned *Resources* object calls its method **OpenRawResources** to open a data stream for reading a raw resource, which in the example is specified by the resource name *R.raw.hello* and will be translated to an integer indentifier by the Android tools. The code creates a byte array, named *reader*, large enough to hold all the data of the input data stream using the *InputStream* class method **available**(). The raw bytes read into the array *reader* is converted to a string usng the *String* constructor and is printed out as a log message.

When we run the application, we will see the message *Android The Beautiful!* appeared in the *Text* column of the Eclipse IDE **LogCat** output.

5.2 Read and Write Files

5.2.1 Read Assets and Display Files

For each application, Android tools generate an *assets* directory for users to put customized resources there. In contrast to the *res* directory, which is accessible from *R.class*, *assets* behaves like a file system, In the *res* directory, each file is assigned an integer identifier, which can be accessed easily through **R.id.***res_id*, providing convenient ways to access images, sounds, icons, xml files and some commonly used resources. On the other hand, the *assets* directory provides users more freedom to put any file there, which can be then accessed like a file in a Java file system. Android does not generate IDs for assets content, and thus we need to specify relative paths and names for files inside the directory *assets*.

The Android public final class *AssetManager*, which extends class *Object* provides access to an application's raw asset files. The class presents a lower-level API that allows one to open and read any raw files lying inside the **Assets** directory or its subdirectories. The APIs could read the raw asset files that have been bundled with the application as a simple stream of bytes.

To illustrate the technique, we again consider a simple example that reads data from an asset file as raw streams. We use the **getAssets**() method of public class *Resources* to obtain an *AssetManager* object, from which we use its **open**() method to open the asset file as an input stream. Suppose we have used Eclipse IDE defaults to create the the project and application *ReadAsset*, and we call the package *data.readasset*, and the asset file *hello_world.txt*. We do the following to finish building the project that reads data from the asset file and prints out its data as a string:

1. **Create File** *assets/hello_world.txt*: Click **File** > **New** > **File**. Select the **parent folder** to be *ReadAsset/assets* and enter *hello_world.txt* for **File name**. Then click **Finish**, which creates the file *assets/hello_world.txt*. You may enter any text in *hello_world.txt*. For example, we enter *Hello World, Democracy the Beautiful!* in the file and save it.
2. **Modify** *MainActivity.java*: Modify the main program *MainActivity.java* to the following code, which simply reads the data from the file *assets/hello_world.txt* as a simple stream and prints it out as a string.

```
------------------------------------------------------------------
package data.readassest;

import java.io.IOException;
import java.io.InputStream;
import android.os.Bundle;
import android.app.Activity;
import android.content.res.AssetManager;

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

    try{
      InputStream inputStream=this.getAssets().open("hello_world.txt");
      int nBytes = inputStream.available();
      byte data[] = new byte[nBytes];
      String str = "";

      //read all data from file "hello_world.txt"
      while( inputStream.read(data) != -1);

      str = new String ( data );
      System.out.println ( str );
      inputStream.close();
    } catch(IOException e) {
      e.printStackTrace();
    }
  }
}
------------------------------------------------------------------
```

In the code, the statement

InputStream inputStream = this.getAssets().open("hello_world.txt");

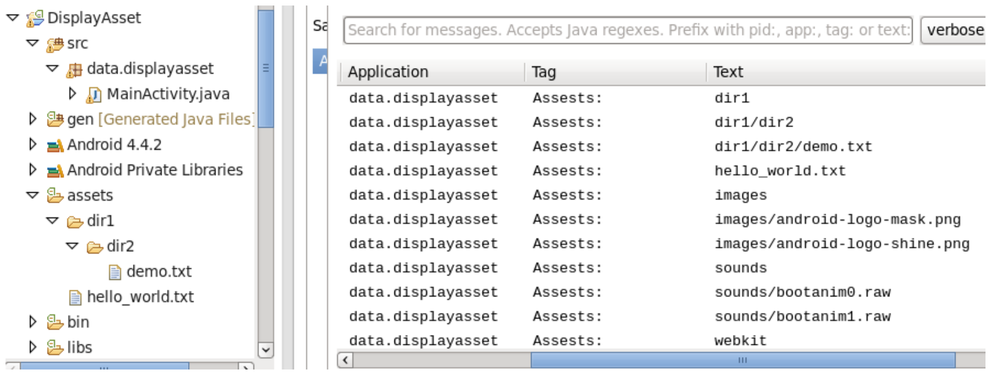
finds the file location and opens it as an input stream. The code finds the number of bytes the file has using the method **available**() of the class *InputStream*. It then allocates memory for an array, reads the data as bytes to the array, converts the bytes to a string using a constructor of the class *String* and prints out the string. When we run the application, we should see the text *Hello World, Democracy the Beautiful!* appear in the *Text* column of the entry with **Tag** *System.out* of the Eclipse IDE **LogCat** output.

We can also make use of the *AssetManager* class to display files, including subdirectories in the *assets* directory. We can use the **list**() method of *AssetManager* to find all the files inside the *assets* directory. The method takes a string as the input parameter, which specifies a relative path like *docs/home.html* within *assets*; it returns a *String* array of all the assets at the given path.

Suppose in our project, in addition to the file *hello_world.txt*, we have created a subdirectory, *dir1*, inside *assets* and another subdirectory, *dir2* under *dir1*. We create another file, called *demo.txt* inside *assets/dir1/dir2*. The following code shows using this method to display all assets files and directories:

```
---------------------------------------------------------------------
package data.displayasset;

import java.io.File;
import android.util.Log;
import android.os.Bundle;
import java.io.IOException;
import android.app.Activity;
import android.content.res.AssetManager;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    final AssetManager mgr = getAssets();
    displayFiles(mgr, "");
  }

  void displayFiles (AssetManager mgr, String path) {
    try {
      String list[] = mgr.list(path);
      if (list != null)
        for (int i=0; i<list.length; ++i) {
          if ( path.length() >  0 ) {
            Log.v("Assests:", path + "/" + list[i]);
            displayFiles(mgr, path  + "/" + list[i] );
          }else{
            Log.v("Assests:", path  + list[i]);
            displayFiles(mgr, path  + list[i] );
          }
        }
      else
        Log.v("List:", "empty!");
    } catch (IOException e) {
        Log.v("List error:", "can't list" + path);
    }
  }
}
---------------------------------------------------------------------
```

In the code, the method **displayFiles**() is called recursively to display all the assets files and directories. Initially, the empty string is passed in as the *path* parameter for the **list**() method of the class *AssetManager*, which returns the files and directories in the root directory of the asset file system. A directory name is then appended to *path* to search for the next level recursively. At deeper levels the symbol '/' is added to construct a path in the hierarchical form *dir1/dir2*. Figure 5-1 shows a screen shot of a portion of the Eclipse IDE, showing the project of the outputs at *LogCat* when the program is executed. As one can see from the outputs, besides the files in the *assets* directory, the system also consists of other assets such as images, and sounds.


DisplayAsset
src
data.displayasset
MainActivity.java
gen [Generated Java Files]
Android 4.4.2
Android Private Libraries
assets
dir1
dir2
demo.txt
hello_world.txt
bin
libs
Search for messages. Accepts Java regexes. Prefix with pid:, app:, tag: or text:
verbose

```
Application         Tag          Text
data.displayasset   Assests:     dir1
data.displayasset   Assests:     dir1/dir2
data.displayasset   Assests:     dir1/dir2/demo.txt
data.displayasset   Assests:     hello_world.txt
data.displayasset   Assests:     images
data.displayasset   Assests:     images/android-logo-mask.png
data.displayasset   Assests:     images/android-logo-shine.png
data.displayasset   Assests:     sounds
data.displayasset   Assests:     sounds/bootanim0.raw
data.displayasset   Assests:     sounds/bootanim1.raw
data.displayasset   Assests:     webkit
```

Figure 5-1 Assets

5.2.2 Write to Files

We can easily use a Java class of file I/O to save data in a file. The interested question is: *where is the written file located?* To answer this question, let us again consider a simple example, in which we write some text to a file named *sampleFile.txt*, and read it back, printing the text on the **LogCat** output. Suppose we call the project of this example *WriteFile*. The following is the complete code of the *MainActivity* of *WriteFile*:

```
---------------------------------------------------------------------
package data.writefile;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.IOException;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    try {
      final String helloString = new String("Hello, Friend!");
      FileOutputStream fos = openFileOutput("sampleFile.txt",
                                    MODE_WORLD_READABLE);
      OutputStreamWriter osw = new OutputStreamWriter( fos );

      // Write the string to the file
      osw.write ( helloString );

      // Flush out anything in buffer
      osw.flush();
      osw.close();

      // Read the file back...
```

```
        FileInputStream fis = openFileInput( "sampleFile.txt" );
        InputStreamReader isr = new InputStreamReader( fis );

        // Prepare a char-Array that to read data back
        char[] inputBuffer = new char[fis.available()];

        // Fill the Buffer with data from the file
        isr.read(inputBuffer);

        // Transform the chars to a String
        String readString = new String(inputBuffer);

        // Check if we read back the same chars that we had written out
        boolean writeReadEqual = helloString.equals ( readString );
        Log.i( "String read:", readString );
        Log.i( "File Reading:", "success = " + writeReadEqual );

      } catch (IOException ioe)  { ioe.printStackTrace(); }
    }
  }
  ---------------------------------------------------------------------
```

In the code, we use the methods **openFileOutput**() and **openFileInput**() of the class *Context*, which are implemented in *ContextWrapper*, to access files. For **openFileOutput**, the first parameter is file name and the second is access mode, which specifies who has the right to access the file. In the example, the mode is *MODE_WORLD_READABLE* implying that everyone can access the file.

When we run this code in Eclipse IDE, we will see in the **LogCat** a portion of the output similar to following:

Application	**Tag**	**Text**
data.writefile	String read:	Hello, Friend!
data.writefile	File Reading:	success = true

The log indicates that we have successfully written the data to the file *sampleFile.txt* and read the data back. However, if we examine the directory tree of the project *WriteFile*, we will not find in any of the subdirectories. This is because the file has been saved somewhere else, in an emulated file system, which is compressed and is in the path of our home directory. We can find this out by issuing a listing command such as

```
$ ls ~/.android/
```

In this command, the symbol '∼' means home directory and the dot '.' preceding a directory name means that the directory is hidden. This command displays the files, including directories under *.android*, similar to the following:

```
adbkey          androidwin.cfg  debug.keystore    repositories.cfg
adbkey.pub      avd             default.keyset    sites-settings.cfg
adb_usb.ini     cache           modem-nv-ram-5554
androidtool.cfg ddms.cfg        modem-nv-ram-5556
```

Suppose we examine deeper directories:

$ ls /.android/avd/*avd_name*/

This command would list files with names like the following:

```
cache.img       emulator-user.ini      sdcard.img      userdata-qemu.img
cache.img.lock  hardware-qemu.ini      sdcard.img.lock userdata-qemu.img.lock
config.ini      hardware-qemu.ini.lock userdata.img
```

Our data file *sampleFile.txt* is saved in the compressed file image *userdata-qemu.img*. (We can verify this by a command like **strings userdata-qemu.img | grep sampleFile**, which would display *sampleFile.txt*.) The compressed image can be decompressed by the Android extra utility *simg2img*. Alternatively, we can extract the data files using the Android *adb* command with the **pull** option. (The command *adb –help* lists all options of the utility *adb*.) We can first make a directory, say *uncompressed* inside our project:

```
    $ mkdir uncompressed
    $ cd uncompressed
```

While extracting the data, the emulator should be running. To check the device status, we can issue the command,

$ adb devices

which lists the attached devices, displaying a message similar to the following:

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

Now we can extract the data using the command,

$ adb pull data

If we use *ls* to list the files in *uncompressed*, we will see the following list:

```
  anr  backup       data  nativebenchmark  property  tombstones
  app  dalvik-cache misc  nativetest       system
```

Our file *sampleFile.txt* is in the path *data*, which can be displayed by:

$ ls data/data.writefile/files/

We can use the *cat* command to see its content:

$ cat data/data.writefile/files/sampleFile.txt

which displays the message

```
  Hello, Friend!
```

which is indeed the text we have written to *sampleFile.txt* in our applicaiton.

For more information about internal storage, one can refer to the site:

```
http://developer.android.com/guide/topics/data/data-storage.html#filesInternal
```

5.2.3 External Storage

Normally, an Android device supports a shared external storage that we can store data. It can be a removable storage media such as an SD card. Files saved in the external storage are world-readable. They can be accessed and modified by the user if the USB mass storage has been enabled to transfer files on a computer.

In order to read or write files on the external storage, the application must acquire system permissions *READ_EXTERNAL_STORAGE* or *WRITE_EXTERNAL_STORAGE*, which can be done by adding the permission statements in the manifest file *AndroidManifest.xml* like the following:

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

External storage will become unavailable if the user removes the media. There is no enforcing security on files that we save in the external storage; any application can read or write the files and the user can remove them.

We can setup external storage in the emulator by clicking in the Eclipse IDE:

Window > **Android Virtual Device Manager** > **Android Virtual Devices**

Then we choose our AVD and click **Edit**, which displays an I/O panel showing the information of our AVD. In the entry **SD Card**, we may check **Size** and enter 200 for 200 MB storage.

5.3 Android JNI

5.3.1 Installing Android NDK

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to interface with native applications that may be specific to a hardware and operating system platform. Through JNI, Java programs can call libraries written in other languages such as C/C++ and assembly.

In Android, two key data structures, *JavaVM* and *JNIEnv* are defined by JNI. They are essentially pointers to pointers to function tables. The *JavaVM* structure provides *invocation interface* functions, which allow us to create and destroy a *JavaVM* object. Though in principle a process can have multiple *JavaVM* objects, Android only allows one per process. The *JNIEnv* structure provides most of the JNI functions and our native functions all receive a *JNIEnv* object as the first argument. It is used for thread-local storage and thus we cannot share a *JNIEnv* object threads.

To use the JNI, we must first install the Android NDK, which can be downloaded from the site:
https://developer.android.com/tools/sdk/ndk/index.html
We have to unzip and unpack the package. Suppose we unpack it into the directory
/apps/android/android-ndk-r10
To associate ths package with the Eclipse IDE, click **Window** > **Preferences** > **Android** > **NDK**. Then enter the root directory of the NDK for **NDK Location** (e.g. */apps/android/android-ndk-r10*), and click **Apply** > **OK**. (You may need to restart Eclipse.)

5.3.2 A Simple Example of JNI

We present a simple example here to illustrate how an Android Java program can call a routine written in C/C++ through JNI. Suppose we call our project and application *JniDemo* and the package *data.jnidemo*. As usual, we have created the project using Eclipse and the default main program is *MainActivity.java*. The steps of writing a C program and calling its function are exaplained below. In our example, our C program (*sum-jni.c*) has a function named **sum**(**int** n), which adds the integers from 1 to n and returns the sum of them.

1. Add folder *jni*: Click **File** > **New** > Folder. Enter *JniDemo* for the parent folder name and *jni* for the folder name. Then click **Finish** to create the folder named *jni*.
2. Add *Android.mk* inside folder *jni*: Click **File** > **New** > File. Enter *JniDemo/jni* for the parent folder name and *Android.mk* for the file name and click **Finish** to create the file.

This file acts as a *Makefile* for linking Java and C programs. We write it with the followng statements:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE    := sum-jni
LOCAL_SRC_FILES := sum-jni.c
include $(BUILD_SHARED_LIBRARY)
```

This *Makefile* says that our C file, which contains the C routines is named *sum-jni.c*.

3. Create a Java source wrapper, *SumWrapper.java*: Click **File** > **Class**. Enter *JniDemo/src* for **Source Foler** and *SumWrapper* for **Name**. Click **Finish** to create the class. Modify the file to the following:

```
package data.jnidemo;

public class SumWrapper
{
  // Declare native method public to expose it
  public static native int sum ( int n );

  public static int getSum ( int n) {
    int s = sum ( n );  //call native method
    return s;
  }
  // Load library
  static {
    System.loadLibrary( "sum-jni");  // C-file is sum-jni.c
  }
}
```

This wrapper‘s job is to load the library, and expose any functions in the C program we will use in other Java programs.

4. Create the C header: We compile the source *SumWrapper.java* to obtain its class file and use it to create a C header file, which contains the function prototypes of the native methods. We do this in a terminal via the javac command, assuming our workspace is in the directory */workspace*:

```
$ cd /workspace/JniDemo/src #change into the source directory
$ javac -d /tmp data/jnidemo/SumWrapper.java
```

The switch *-d* specifies an output directory, and here we simply put the class in the directory */tmp*. We can list the class with the command *ls /tmp/data/jnidemo/*, which will display the file name *SumWrapper.class*. The C header file can be created by the commands:

```
$ cd /tmp
$ javah -jni data.jnidemo.SumWrapper
```

This creates the C header file *data_jnidemo_SumWrapper.h* in the directory */tmp*, which looks like the following:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class data_jnidemo_SumWrapper */
```

```
#ifndef _Included_data_jnidemo_SumWrapper
#define _Included_data_jnidemo_SumWrapper
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     data_jnidemo_SumWrapper
 * Method:    sum
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_data_jnidemo_SumWrapper_sum
  (JNIEnv *, jclass, jint);
#ifdef __cplusplus
}
#endif
#endif
```

5. Create C source code: Using the prototype generated by *javah*, we can implement our C source code. We first create the file *sum-jni.c* inside the directory *jni* by clicking **File** > **New** > **File** and entering file name *sum-jni.c*. Modify this file to the following:

```
#include <jni.h>
JNIEXPORT jint JNICALL Java_data_jnidemo_SumWrapper_sum
  (JNIEnv * je, jclass jc, jint n)
{
  int i, sum = 0;

  for ( i = 1; i <= n; i++ )
    sum += i;

  return sum;
}
```

6. Build a shared library: We use the *ndk-build* script to build a shared library from *sum-jni.c*. Suppose we have installed the Android NDK in the directory */apps/android/android-nkk-r10*. To compile, we first go to the project directory */workspace/JniDemo/jni/*, which we created earlier, then run the *ndk-build* script provided by the Android NDK:

```
$ cd /workspace/JniDemo/jni/
$ /apps/android/android-ndk-r10/ndk-build
```

The script makes use of the information in *jni/Android.mk* to create the dynamic-linked library *libsum-jni.so* from *sum-jni.c* and install it in the directory *libs/armeabi* of the project. (i.e. */workspace/JniDemo/libs/armeabi*) The script generates the following messages, reflecting what it has done:

```
[armeabi] Compile thumb :sum-jni <= sum-jni.c
[armeabi] SharedLibrary :libsum-jni.so
[armeabi] Install       :libsum-jni.so=>libs/armeabi/libsum-jni.so
```

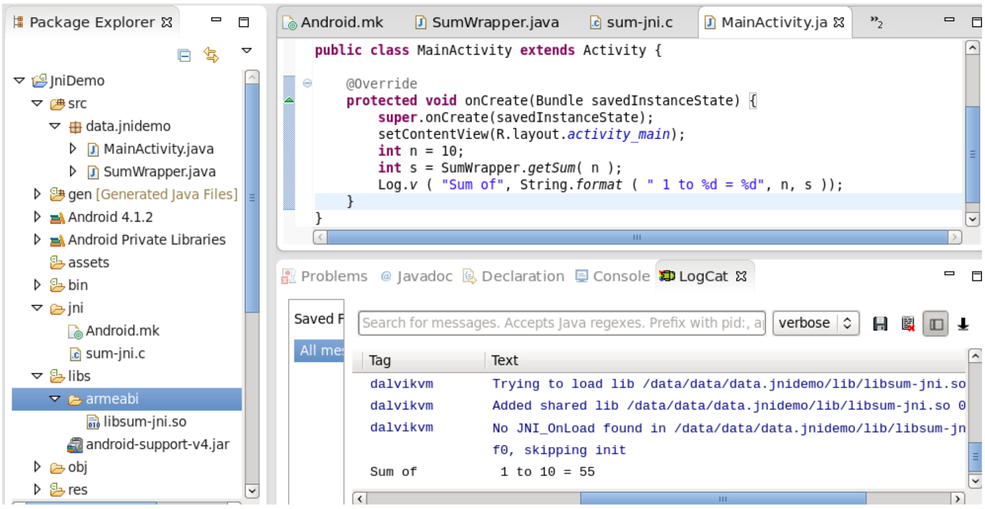
7. Modify *MainActivity.java*: Modify the default main program to the following:

```
package data.jnidemo;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends Activity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  int n = 10;
  int s = SumWrapper.getSum( n );
  Log.v ( "Sum of", String.format ("1 to %d = %d",n,s));
 }
}
```

This main program deos not do much. It simply calls the *SumWrapper*'s **getSum**(n) method, which calls the native method **sum**(n) to calculate the sum $1 + 2 + ... + n$. It sends the result to the *LogCat*.

8. Run the program: We run the progam as usual, clicking **Run** > **Run**. The emulator will not show any output but the Eclipse LogCat will show the sum output. Figure 5-2 shows a caputred image of the Eclipse environment when running the program; we can see from it the file struture and log output of the project.



```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        int n = 10;
        int s = SumWrapper.getSum( n );
        Log.v ( "Sum of", String.format ( " 1 to %d = %d", n, s ));
    }
}
```

```
Tag          Text
dalvikvm     Trying to load lib /data/data/data.jnidemo/lib/libsum-jni.so
dalvikvm     Added shared lib /data/data/data.jnidemo/lib/libsum-jni.so 0
dalvikvm     No JNI_OnLoad found in /data/data/data.jnidemo/lib/libsum-jn
             f0, skipping init
Sum of        1 to 10 = 55
```

Figure 5-2 A Simple JNI Example

5.3.3 A Simple JNI Example with UI

We can easily add a UI to the above example. Suppose we call this new project *JniDemo1*, and have gone through the above steps. To create a UI we first modify *res/layout/activity_main.xml* to:

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

```
    android:id="@+id/linearLayout0"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="12pt"
    android:layout_marginRight="12pt"
    android:layout_marginTop="4pt" >

    <TextView
      android:id="@+id/info"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginLeft="6pt"
      android:layout_marginRight="6pt"
      android:layout_marginTop="4pt"
      android:gravity="center_horizontal"
      android:text="Enter a postive number:"
      android:textSize="12pt" >
    </TextView>
  </LinearLayout>

  <LinearLayout
    android:id="@+id/linearLayout1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="12pt"
    android:layout_marginRight="12pt"
    android:layout_marginTop="4pt" >

    <EditText
      android:id="@+id/n"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_marginRight="6pt"
      android:layout_weight="1"
      android:inputType="numberDecimal" >
    </EditText>

    <Button   android:id="@+id/getsum"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Get Sum"
      android:textSize="10pt"
      android:onClick="onClick"/>
  </LinearLayout>

  <TextView
    android:id="@+id/sum"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="6pt"
    android:layout_marginRight="6pt"
    android:layout_marginTop="4pt"
    android:gravity="center_horizontal"
    android:textSize="12pt" >
  </TextView>
```

```
</LinearLayout>
```

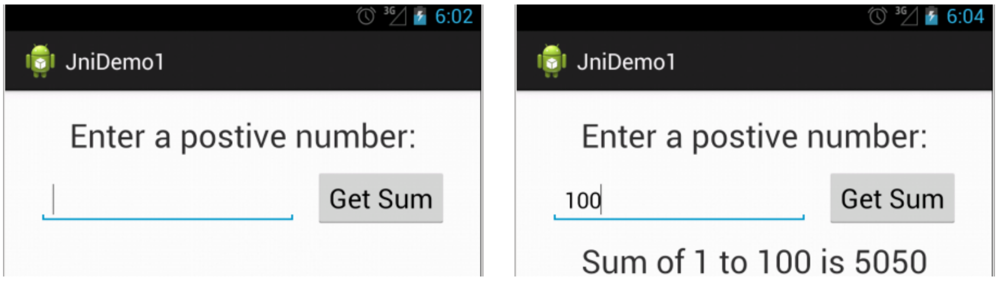
In this layout, we define an *EditText* for the user to enter an integer. We define a *Button* named *getsum* for the user to click upon that calls the summing routine. The result is didsplayed in *TextView sum*.

We also need to modify *MainActivity.java* to the following:

```
package data.jnidemo1;

import android.os.Bundle;
import android.view.View;
import android.widget.*;
import android.app.Activity;
import android.text.TextUtils;

public class MainActivity extends Activity
{

  private EditText n;
  private TextView sum;
  private Button getsum;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  n = (EditText) findViewById(R.id.n);
  getsum = (Button) findViewById(R.id.getsum);
  sum = (TextView) findViewById(R.id.sum);
  TextView info = (TextView) findViewById(R.id.info);
}

// @Override
public void onClick(View view) {
  // check if the fields are empty
  if (TextUtils.isEmpty(n.getText().toString()) )
     return;

  // read numbers from EditText
  String str = n.getText().toString();

  int n1 = Integer.parseInt( str );
  int s = SumWrapper.getSum( n1 );

  // display result
  str = "Sum of 1 to " + str + " is ";
  str += Integer.toString ( s );;
  sum.setText ( str );
}
}
```

The method **onClick**() is called when the user clicks on the *Button getsum*. It reads the integer entered and saves it in *EditText n*. The *EditText n* is converted to a *String*, which in turn is converted to an integer. The method then calls **getSum**() of the class *SumWrapper*, which in turn calls the native routine **sum** to calculate the sum of the integers from 1 to n. The returned result is displayed by *TextView sum*.

When we run the program we will see a UI like that of Figure 5-3(a). Figure 5-3(b) shows the UI after we have entered the number 100 and clicked the button.



Sum of 1 to 100 is 5050

(a) (b)

Figure 5-3 UI of Project *JniDemo1*