Reading Files

Android supports all the standard Java file I/O APIs for create, read, update, and delete (CRUD) operations, along with some additional helpers to make accessing those files in specific locations a little more convenient. There are three main locations in which an application can work with files:

- *Internal storage*: Protected directory space to read and write file data.
- *External storage*: Externally mountable space to read and write file data. Requires the `WRITE_EXTERNAL_STORAGE` permission in API Level 4+. Often, this is a physical SD card in the device.
- *Assets*: Protected read-only space inside the APK bundle. Good for local resources that can't or shouldn't be compiled.

While the underlying mechanism to work with file data remains the same, we will look at the details that make working with each destination slightly different.

How It Works

As we stated earlier, the traditional Java `FileInputStream` and `FileOutputStream` classes constitute the primary method of accessing file data. In fact, you can create a `File` instance at any time with an absolute path location and use one of these streams to read and write data. However, with root paths varying on different devices and certain directories being protected from your application, we recommend some slightly more efficient ways to work with files.

Internal Storage – IO Files

In order to create or modify a file's location on internal storage, utilize the `Context.openFileInput()` and `Context.openFileOutput()` methods. These methods require only the name of the file as a parameter, instead of the entire path, and will reference the file in relation to the application's protected directory space, regardless of the exact path on the specific device. See Listing 5-14.

Listing 5-14. CRUD a File on Internal Storage

```
public class InternalActivity extends Activity {

    private static final String FILENAME = "data.txt";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        setContentView(tv);

        //Create a new file and write some data
        try {
            FileOutputStream mOutput = openFileOutput(FILENAME, Activity.MODE_PRIVATE);
            String data = "THIS DATA WRITTEN TO A FILE";
            mOutput.write(data.getBytes());
            mOutput.flush();
            mOutput.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Read the created file and display to the screen
        try {
            FileInputStream mInput = openFileInput(FILENAME);
            byte[] data = new byte[128];
            mInput.read(data);
            mInput.close();

            String display = new String(data);
            tv.setText(display.trim());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Delete the created file
        deleteFile(FILENAME);
    }
}
```

This example uses `Context.openFileOutput()` to write some simple string data out to a file. When using this method, the file will be created if it does not already exist. It takes two parameters: a file name and an operating mode. In this case, we use the default operation by defining the mode as `MODE_PRIVATE`. This mode will overwrite the file with each new write operation; use `MODE_APPEND` if you prefer that each write append to the end of the existing file.

After the write is complete, the example uses `Context.openFileInput()`, which requires only the file name again as a parameter to open an `InputStream` and read the file data. The data will be read into a byte array and displayed to the user interface through a `TextView`. Upon completing the operation, `Context.deleteFile()` is used to remove the file from storage.

■ **Note** Data is written to the file streams as bytes, so higher-level data (even strings) must be converted into and out of this format.

This example leaves no traces of the file behind, but we encourage you to try the same example without running `deleteFile()` at the end in order to keep the file in storage. Using the SDK's DDMS tool with an emulator or unlocked device, you may view the filesystem and can find the file this application creates in its respective application data folder.

Because these methods are a part of `Context`, and not bound to an activity, this type of file access can occur anywhere in an application that you require, such as a `BroadcastReceiver` or even a custom class. Many system constructs either are a subclass of `Context` or will pass a reference to one in their callbacks. This allows the same open/close/delete operations to take place anywhere.

External Storage

The key differentiator between internal and external storage is that external storage is mountable. This means that the user can connect his or her device to a computer and have the option of mounting that external storage as a removable disk on the PC. Often, the storage itself is physically removable (such as an SD card), but this is not a requirement of the platform.

■ **Important** Writing to the external storage of the device will require that you add a declaration for `android.permission.WRITE_EXTERNAL_STORAGE` to the application manifest. Reading from external storag requires `android.permission.READ_EXTERNAL_STORAGE` as well on API Level 19+.

During periods when the device's external storage is either mounted externally or physically removed, it is not accessible to an application. Because of this, it is always prudent to check whether external storage is ready by checking `Environment.getExternalStorageState()`.

Let's modify the file example to do the same operation with the device's external storage. See Listing 5-16.

Listing 5-15. CRUD a File on External Storage

```
public class ExternalActivity extends Activity {

    private static final String FILENAME = "data.txt";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        setContentView(tv);

        //Create the file reference
        File dataFile = new File(Environment.getExternalStorageDirectory(), FILENAME);

        //Check if external storage is usable
        if(!Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
            Toast.makeText(this, "Cannot use storage.", Toast.LENGTH_SHORT).show();
            finish();
            return;
        }

        //Create a new file and write some data
        try {
            FileOutputStream mOutput = new FileOutputStream(dataFile, false);
            String data = "THIS DATA WRITTEN TO A FILE";
            mOutput.write(data.getBytes());
            mOutput.flush();
            //With external files, it is often good to sync the file
            mOutput.getFD().sync();
            mOutput.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Read the created file and display to the screen
        try {
            FileInputStream mInput = new FileInputStream(dataFile);
            byte[] data = new byte[128];
            mInput.read(data);
            mInput.close();

            String display = new String(data);
            tv.setText(display.trim());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Delete the created file
        dataFile.delete();
    }
}
```

With external storage, we utilize a little more of the traditional Java file I/O. The key to working with external storage is calling `Environment.getExternalStorageDirectory()` to retrieve the root path to the device's external storage location.

Before any operations can take place, the status of the device's external storage is first checked with `Environment.getExternalStorageState()`. If the value returned is anything other than `Environment.MEDIA_MOUNTED`, we do not proceed because the storage cannot be written to, so the activity is closed. Otherwise, a new file can be created and the operations may commence.

The input and output streams must now use default Java constructors, as opposed to the `Context` convenience methods. The default behavior of the output stream will be to overwrite the current file or to create it if it does not exist. If your application must append to the end of the existing file with each write, change the Boolean parameter in the `FileOutputStream` constructor to `true`.

Often, it makes sense to create a special directory on external storage for your application's files. We can accomplish this simply by using more of Java's file API. See Listing 5-16.

Listing 5-16. CRUD a File Inside a New Directory

```
public class ExternalActivity extends Activity {

    private static final String FILENAME = "data.txt";
    private static final String DNAME = "myfiles";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        setContentView(tv);
        //Create a new directory on external storage
        File rootPath = new File(Environment.getExternalStorageDirectory(), DNAME);
        if(!rootPath.exists()) {
            rootPath.mkdirs();
        }
        //Create the file reference
        File dataFile = new File(rootPath, FILENAME);

        //Create a new file and write some data
        try {
            FileOutputStream mOutput = new FileOutputStream(dataFile, false);
            String data = "THIS DATA WRITTEN TO A FILE";
            mOutput.write(data.getBytes());
            mOutput.flush();
            //With external files, it is often good to wait for the write
            mOutput.getFD().sync();

            mOutput.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Read the created file and display to the screen
        try {
            FileInputStream mInput = new FileInputStream(dataFile);
            byte[] data = new byte[128];
            mInput.read(data);
            mInput.close();

            String display = new String(data);
            tv.setText(display.trim());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Delete the created file
        dataFile.delete();
    }
}
```

In this example, we created a new directory path within the external storage directory and used that new location as the root location for the data file. Once the file reference is created using the new directory location, the remainder of the example is the same.