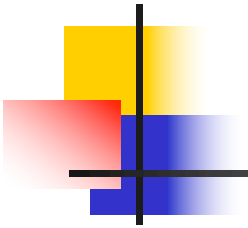
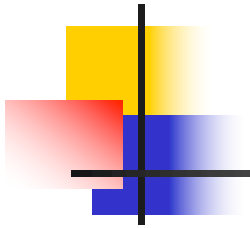
Accessing Files and Directories

Expanded by KJK



Overview

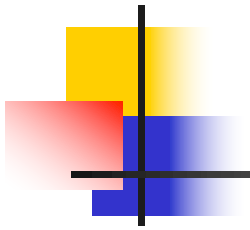
- Work with application data on the device
- Practice good file management
- Understand Android file permissions
- Work with files and directories
- Explore the Android application directories
- Create and write to files in the default application directory
- Read from files in the default application directory
- Read raw files byte by byte (already know this)
- Read XML files
- Work with other directories and files on the Android file system
- Create and write files to external storage



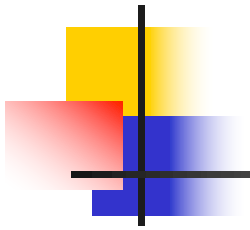
Android Apps and Data

- Many Android apps need to access and/or maintain data to fulfill their functions.
- The data ranges from background images and help-like text information, stock prices, game supporting data, national weather forecasts, to music and video streams of TV shows, to mention just a few.
- Some of the data can and should be accessed and maintained locally, i.e., within the Android device, while other data can be obtained or stored remotely, as needed.

Working with Application Data on the Device



- Many applications require a robust solution for storing data persistently.
- Some types of data that an application might want to store include:
  - Multimedia content such as images, sounds, video, and other complex information
  - Content downloaded from a network
  - Complex content generated by the application



Android Apps and Data

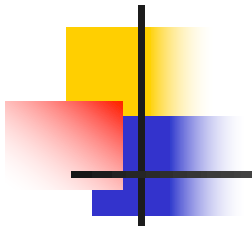
- Data can be accessed and stored locally in a variety of ways in an Android app:
  - as files included as raw resources or included as app's assets (read only)
  - as files in the internal (private) area of the app
  - as files in the external files storage of the device
  - as files in the internal cache area
  - as files in the external storage cache area
  - as shared preferences (properties) of the app.

Android Apps and Data

- *Structured data* can be accessed and stored either locally or externally (not on the device):
  - as records in a local SQLite database
  - as records in a remote Relational Database (RDB).
- Other data, unstructured, structured, or semi-structured can be handled, as well, by reading/writing to external sources via a network connection.

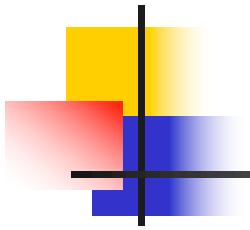
Working with Application Data on the Device (Cont'd)

- Android applications can create and use directories and files to store their data in a variety of ways, most commonly:
  - Storing private application data in the application directory
  - Caching data under the application's cache directory
  - Storing shared application data on external storage devices or shared device directory areas



Practicing Good File Management

- Here are a few of the most important best practices for working with files on the Android file system:
  - Anytime you read or write data to disk, you are performing intensive blocking operations and using valuable device resources; therefore, in most cases, file access functionality should not be performed on the main UI thread of an application.

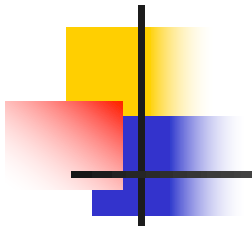


Practicing Good File Management

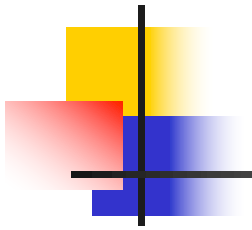
- Instead, these operations should be handled asynchronously using threads, AsyncTask* objects, or other asynchronous methods.
- Even working with small files can slow down the UI thread due to the nature of the underlying file system and hardware.

* AsyncTask has been deprecated in API level 30; we will talk about the alternatives later.

Practicing Good File Management



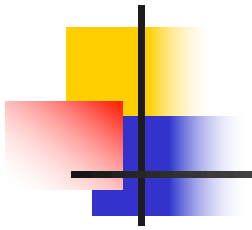
- More best practices:
  - Android devices have limited storage capacity; therefore, to free up space on the device, store only what you need to store, and clean up old data when it is no longer needed.
  - Use external storage whenever it is appropriate to give the user more flexibility.



Practicing Good File Management)

- More best practices:
  - Be a good citizen on the device.
    - Check for availability of resources such as disk space and external storage opportunities prior to using them and causing errors or crashes.
  - Do not forget to set appropriate file permissions for new files.
  - Release resources when you're not using them!
    - In other words, if you open them, close them, and so on.

Practicing Good File Management

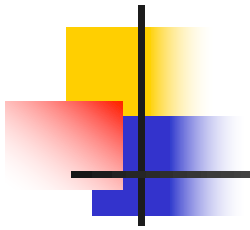


- More best practices:
  - Implement efficient file access algorithms for reading, writing, and parsing file contents.
  - Use the many profiling tools available as part of the Android SDK to identify and improve the performance of your code.
  - A good place to start is with the StrictMode API:
    - `android.os.StrictMode`

Practicing Good File Management

```
private static boolean DEVELOPER_MODE = true;
public void onCreate() {
    if( DEVELOPER_MODE ) {
        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                .detectDiskReads()
                .detectDiskWrites()
                .detectNetwork()   // or .detectAll() for all detectable problems
                .penaltyLog()
                .build());
        StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                .detectLeakedSqlLiteObjects()
                .detectLeakedClosableObjects()
                .penaltyLog()
                .penaltyDeath()
                .build());
    }
    super.onCreate();
}
```

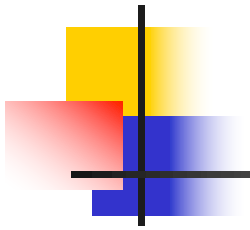
Practicing Good File Management



- It is possible to decide what should happen when a violation is detected.
- For example, using

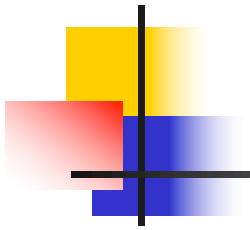
  StrictMode.ThreadPolicy.Builder.penaltyLog()

  one can watch the output of the logcat while ones uses the application to see the violations as they happen.
- StrictMode.ThreadPolicy.Builder.penaltyDeath() will crash the whole process, if a violation happens.



Practicing Good File Management

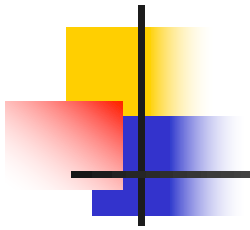
- If the data the application needs to store is well structured, consider using an SQLite database to store it.
- Test your application on real devices; different devices have different processor speeds.
- Do not assume that because your application runs smoothly on the emulator it will run that way on real devices.
- If you're using external storage, test when external storage is not available.



Android File Permissions

- Each Android application is its own user on the underlying Linux operating system; it has its own application directory and files.
- Files created in the application's directory are private to that application by default.
- Files can be created on the Android file system with different permissions specifying how the files are accessed.
- Permission modes are most commonly used when creating files.

Android File Permissions

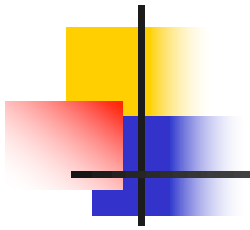


- These permission modes are defined in the `Context` class (`android.content.Context`):
  - `MODE_PRIVATE` (the default) is used to create a file that can be accessed only by the "owner" application itself.
    - From a Linux perspective, this means the specific user identifier.
    - The constant value of `MODE_PRIVATE` is 0, so you may see this used in legacy code.
  - `MODE_APPEND` is used to append data to the end of an existing file.
    - The constant value of MODE_APPEND is 32768.

Android File Permissions

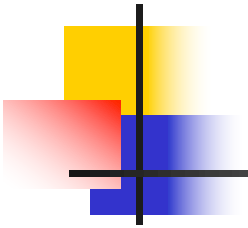
- An application does not need any special Android manifest file permissions to access its own private file system area.
- However, in order for your application to access external storage, the app will need to register for the `WRITE_EXTERNAL_STORAGE` permission in the AndroidManifest:

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



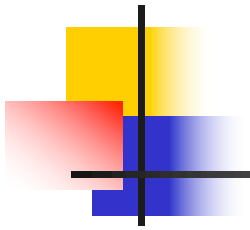
Working with Files and Directories

- Within the Android SDK, you can also find a variety of standard Java file utility classes (such as `java.io`) for handling different types of files, such as:
  - Text files
  - Binary files
  - XML files



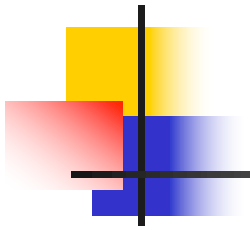
Working with Files and Directories

- Retrieving the file handle to a resource file is performed slightly differently from accessing files on the device file system, but once you have a file handle, either method allows you to perform read and other operations.



Working with Files and Directories

- Android application file resources are part of the application package and are therefore accessible only to the application itself.
- Android application files are stored in a standard directory hierarchy on the Android file system.
- Any data written to a file in the app's directory must be written by the app. It is impossible to "give the app" some data this way.



Working with Files and Directories

- Applications access the Android device file system using methods within the Context class:
  - `android.content.Context`
- The application, or any Activity class, can use the application `Context` to access its private application file directory or cache directory.
- From here, you can add, remove, and access files associated with your application.
- By default, these files are private to the application and cannot be accessed by other applications or by the user.

Exploring with the Android Application Directories

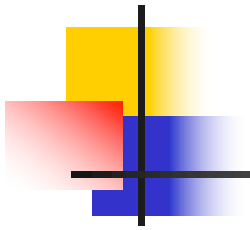
- Android application data is stored on the Android file system in the following top-level directory:
  - `/data/data/<package name>/`
- Several default subdirectories are created for storing databases, preferences, and files, as necessary.
- The actual location of these directories may vary, depending on the device.
- You can also create other custom directories as needed.
- File operations all begin by interacting with the application's Context object.

Exploring with the Android Application Directories (Cont'd)

Method	Purpose
`Context.getFilesDir()`	Retrieves the application `/files` subdirectory
`Context.openFileInput()`	Opens a private application file for reading
`Context.openFileOutput()`	Opens a private application file for writing
`Context.getFileStreamPath()`	Returns the absolute file path to a file in the application's `/files` subdirectory
`Context.deleteFile()`	Deletes a private application file by name
`Context.fileList()`	Gets a list of all files in the `/files` subdirectory
`Context.getCacheDir()`	Retrieves the application `/cache` subdirectory
`Context.getDir()`	Creates or retrieves a named application subdirectory
`Context.getExternalCacheDir()`	Retrieves the `/cache` subdirectory on the external file system (API Level 8)
`Context.getExternalFilesDir()`	Retrieves the `/files` subdirectory on the external file system (API Level 8)

The above methods are ***not static*** – need a Context object.

Creating and Writing to Files in the Default Application Directory



- Android applications that require only the occasional file to be created should rely on the helpful `Context` class method called `openFileOutput()`.
- Use this method to create files in the default location under the application data directory:

  `/data/data/<package_name>/files/`

- To view this directory in AS:

  View->Tool Windows->Device File Explorer

Creating/Writing to Files in the Default Application Directory (Cont'd)

- The following code snippet creates and opens a file called `Filename.txt`.
- We write a single line of text to the file and then close the file:

```
import java.io.FileOutputStream;
...
FileOutputStream fos;
String strFileContents = "Some text to write to the file.";
fos = openFileOutput( "Filename.txt", MODE_PRIVATE );
fos.write(strFileContents.getBytes());
fos.close();
```

Creating/Writing to Files in the Default Application Directory (Cont'd)

- We can append data to the file by opening it with the mode set to `MODE_APPEND`:

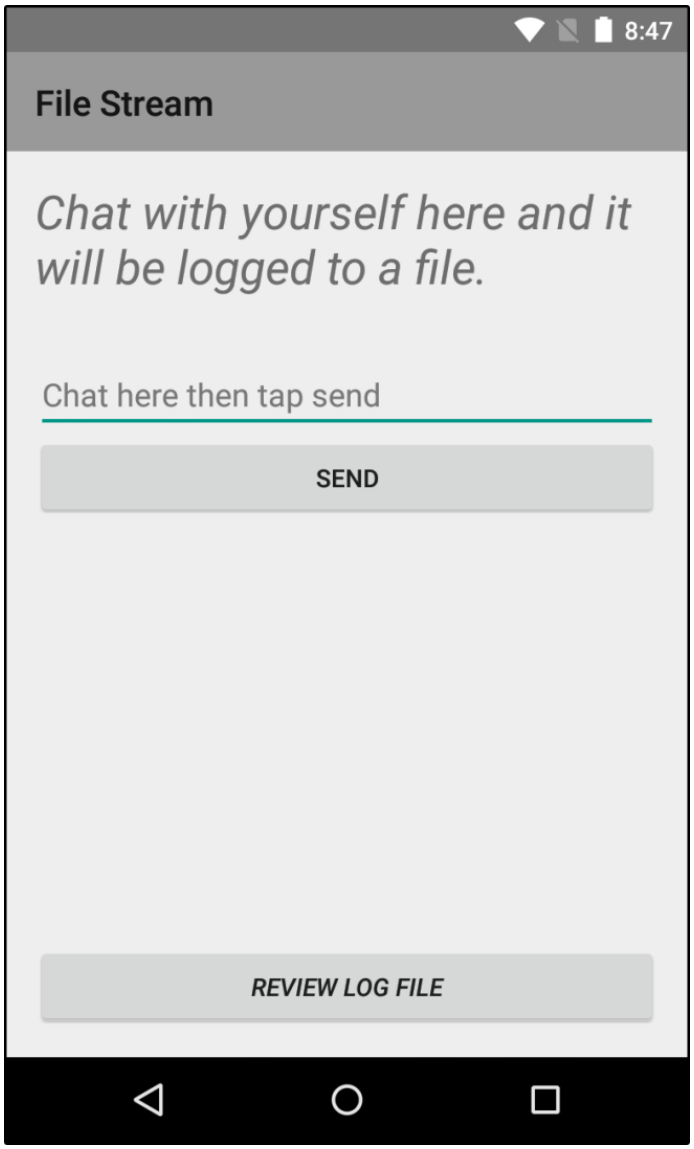
```
import java.io.FileOutputStream;
...
FileOutputStream fos;
String strFileContents = "More text to write to the file.";
fos = openFileOutput("Filename.txt", MODE_APPEND);
fos.write(strFileContents.getBytes());
fos.close();
```

Creating/Writing to Files in the Default Application Directory (Cont'd)

- The file we just created has the following path on the Android file system:

  /data/data/<package name>/files/Filename.txt

Creating and Writing to Files in the Default Application Directory (Cont'd)

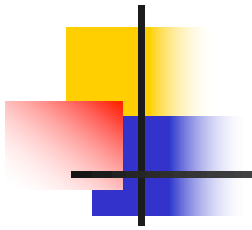


Reading from Files in the Default Application Directory

- Again, we have a shortcut for reading files stored in the default /files subdirectory.
- The following code snippet opens a file called `Filename.txt` for read operations:

```
import java.io.FileInputStream;

...

String strFileName = "Filename.txt";

FileInputStream fis = openFileInput(strFileName);
```

Reading from Files in the Default Application Directory (Cont'd)

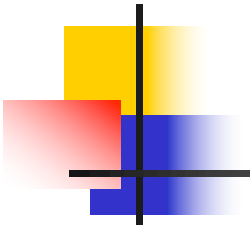


Reading Raw Files Byte by Byte

- Handle file-reading and file-writing operations using standard Java methods.
- The java.io.InputStreamReader and java.io.BufferedReader are used for reading bytes and characters from different types of primitive file types.

Reading Raw Files Byte by Byte

```
FileInputStream fis = openFileInput(filename);
StringBuffer sBuffer = new StringBuffer();
BufferedReader dataIO = new BufferedReader (new
                                        InputStreamReader(fis));
String strLine = null;
while ((strLine = dataIO.readLine()) != null) {
    sBuffer.append(strLine + "\n");
}
dataIO.close();
fis.close();
```



Reading XML Files

Package or Class	Purpose
`android.sax.*`	Framework to write standard SAX handlers
`android.util.Xml`	XML utilities, including the `XMLPullParser` creator
`org.xml.sax.*`	Core SAX functionality (project: http://www.saxproject.org/)
`javax.xml.*`	SAX and limited DOM, Level 2 Core support
`org.w3c.dom`	Interfaces for DOM, Level 2 Core
`org.xmlpull.*`	`XmlPullParser` and `XMLSerializer` interfaces as well as a SAX2 Driver class (project: http://www.xmlpull.org/)

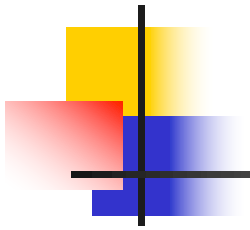
Reading XML Files (Cont'd)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Our pet list -->
<pets>
   <pet type="Bunny" name="Bit"/>
   <pet type="Bunny" name="Nibble"/>
   <pet type="Bunny" name="Stack"/>
   <pet type="Bunny" name="Queue"/>
   <pet type="Bunny" name="Heap"/>
   <pet type="Bunny" name="Null"/>
   <pet type="Fish" name="Nigiri"/>
   <pet type="Fish" name="Sashimi II"/>
   <pet type="Lovebird" name="Kiwi"/>
</pets>
```

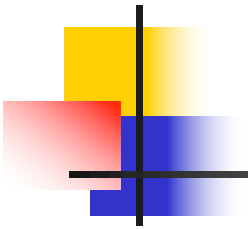
Reading XML Files (Cont'd)

```
XmlResourceParser myPets = getResources().getXml( R.xml.my_pets );
int eventType = -1;
while (eventType != XmlResourceParser.END_DOCUMENT) {
    if(eventType == XmlResourceParser.START_DOCUMENT) {
        Log.d(DEBUG_TAG, "Document Start");
    } else if(eventType == XmlResourceParser.START_TAG) {
        String strName = myPets.getName();
        if(strName.equals("pet")) {
            Log.d(DEBUG_TAG, "Found a PET");
            Log.d(DEBUG_TAG, "Name: "+ myPets.getAttributeValue(null,
                "name"));
            Log.d(DEBUG_TAG, "Species: "+ myPets.getAttributeValue(null,
                "type"));
        }
    }
    eventType = myPets.next();
}
Log.d(DEBUG_TAG, "Document End");
```

Supporting Adoptable Storage Devices

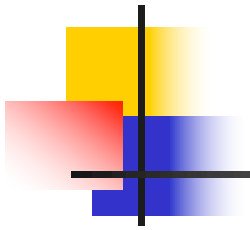


- Android Marshmallow introduced a new feature for device users that allows them to adopt their external storage device — usually an SD card — by encrypting and formatting the SD card to function just like internal storage.
- This will allow users to transfer their applications and associated private files among the internal storage and the SD card, which functions just like internal storage.



Supporting Adoptable Storage Devices

- Need to make sure not to hard-code file path names with this new feature, since the file path names will dynamically change when a user moves your application and associated files from one storage device to the other.
- For safety, use only the Context methods for determining path names — methods like getFilesDir() and getDir().



Working with Other Directories and Files on Android's File System

- Using `Context.openFileOutput()` and `Context.openFileInput()` method calls is great if you have a few files and you want them stored in the application's private files subdirectory.
- If you have more sophisticated file management needs, you need to set up your own directory structure.
- To do this, you must interact with the Android file system using the standard `java.io.File` class methods.

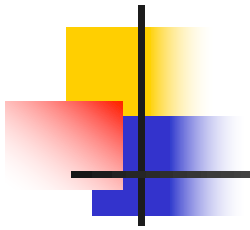
Working with Other Directories and Files on Android's File System

```
import java.io.File;

...

File pathForAppFiles = getFilesDir();

String[] fileList = pathForAppFiles.list();
```

Working with Other Directories and Files on Android's File System

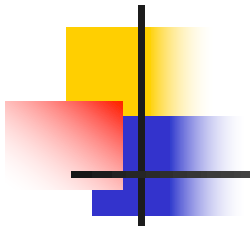
```
import java.io.File;

import java.io.FileOutputStream;
...
File fileDir = getFilesDir();

String strNewFileName = "myFile.dat";

String strFileContents = "Some data for our file";

File newFile = new File(fileDir, strNewFileName);

newFile.createNewFile();

FileOutputStream fo = new
FileOutputStream(newFile.getAbsolutePath());

fo.write(strFileContents.getBytes());

fo.close();
```

Working with Other Directories and Files on Android's File System



- You can use File objects to manage files within a desired directory and create subdirectories.
- You might want to store "track" files within "album" directories or create a file in a directory other than the default.
- Let's say you want to cache some data to speed up your application's performance and how often it accesses the network.
  - In this instance, you might want to create a cache file.

Working with Other Directories and Files on Android's File System

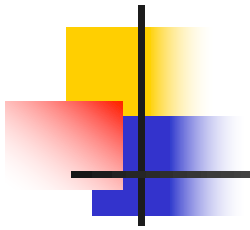


- There is a special application directory for storing cache files on the Android file system, retrievable with a call to the `getCacheDir()` method:
  - /data/data/<package name>/cache/
- The external cache directory, found via a call to the `getExternalCacheDir()` method, is not treated the same in that files are not automatically removed from it.
- Cache files are temporary, and Android may remove some or all of them if low on space.

Working with Other Directories and Files on Android's File System

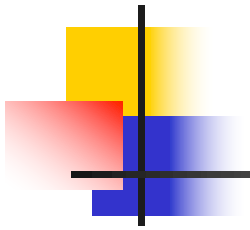
```
File pathCacheDir = getCacheDir();
String strCacheFileName = "myCacheFile.cache";
String strFileContents = "Some data for our file";

File newCacheFile =
        new File(pathCacheDir, strCacheFileName);
newCacheFile.createNewFile();

FileOutputStream foCache =
        new FileOutputStream(newCacheFile.getAbsolutePath());
foCache.write(strFileContents.getBytes());
foCache.close();
newCacheFile.delete();
```

Creating and Writing Files to External Storage



- Applications should store large amounts of data on external storage (using an SD card) rather than limited internal storage.
- You can also access external file storage, such as the SD card, from within your application.
- This is a little more involved than working within the confines of the application directory, as SD cards are removable.
- As a result, you need to check to see if the storage is mounted before use.

Creating and Writing Files to External Storage

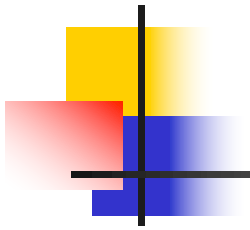


- You can access external storage on the device using the `Environment` class:

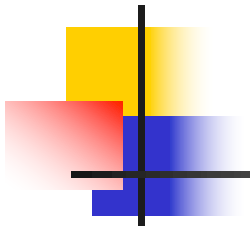
```
android.os.Environment
```

- The Environment class provides many static methods to work with external storage
- Begin by using the `getExternalStorageState()` method to check the mount status of external storage.
- You can store private application files on external storage, or you can store public shared files such as media.

Creating and Writing Files to External Storage

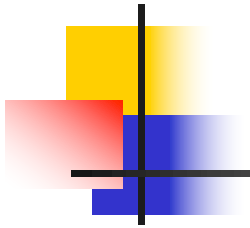


- If you want to store private application files, use the `getExternalFilesDir()` method of the `Context` class.
  - These files will be cleaned up if the application is uninstalled later.
- The external cache is accessed using the `getExternalCacheDir()` method.



Summary

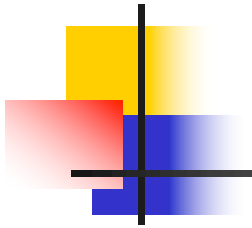
- We have learned a variety of ways to store and manage application data using the Android platform.
- We have learned that the method for storing data depends on what type of data you want to store.
- We are now able to store private files that are readable only by our application.
- We have learned that performing disk operations asynchronously is important when working with the Android file system.



References and More Information

- Android SDK Reference regarding the java.io package:
  - http://d.android.com/reference/java/io/package-summary.html
- Android SDK Reference regarding the Context interface:
  - http://d.android.com/reference/android/content/Context.html
- Android SDK Reference regarding the File class:
  - http://d.android.com/reference/java/io/File.html
- Android SDK Reference regarding the Environment class:
  - http://d.android.com/reference/android/os/Environment.html

References and More Information



- Android Training: “Saving Files”:
  - http://d.android.com/training/basics/data-storage/files.html
- Android API Guides: “Using the Internal Storage”:
  - http://d.android.com/guide/topics/data/data-storage.html#filesInternal
- Android API Guides: “Using the External Storage”:
  - http://d.android.com/guide/topics/data/data-storage.html#filesExternal
- Android API Guides: “App Install Location”:
  - http://d.android.com/guide/topics/data/install-location.html
- Android API Guides: “<manifest>”:
  - http://d.android.com/guide/topics/manifest/manifest-element.html
- Android SDK Reference regarding the ContextCompat class:
  - http://d.android.com/reference/android/support/v4/content/ContextCompat.html