Ubiquitous Computing

COMP H4025

Lecturer: Simon McLoughlin

Lecture 3

Lecture Overview

This week:

- AsyncTasks
- Intents
- Passing data between Activities
- Networking
- Persistent Storage

Logging Messages in Android

- Android offers a system-wide logging capability. You can log from anywhere in your code by calling **Log.d(TAG, message),** where TAG and message are some strings.
- TAG should be a tag that is meaningful to you given your code. Typically, a tag would be the name of your app, your class, or some module. Good practice is to define TAG as a Java constant for your entire class, such as:

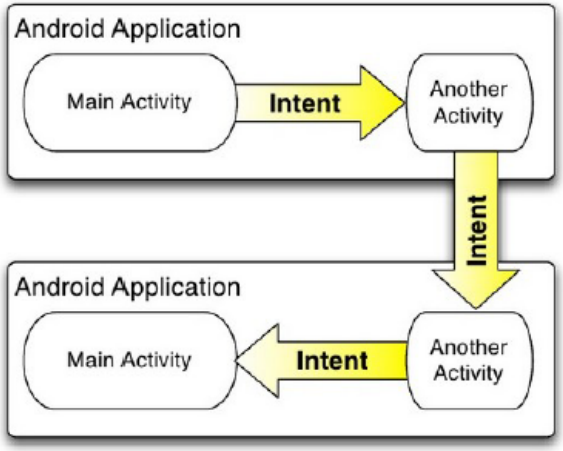
```
private static final String TAG = “MyActivity";
```

- Make sure you import the log class when utilising logging in your app.
- Note that Log takes different **severity levels**. .d() is for debug level, but you can also specify .e() for error, .w() for warning, or .i() for info. There’s also a .wtf() severity level for errors that should never happen.
- The Android system log is outputted to **LogCat**, a standardized system-wide logging mechanism. LogCat is readily available to all Java code. The developer can easily view the logs and filter their output based on severity, such as debug, info, warning, or error,

Intents

• An Intent is a **messaging object** you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use-cases:

1. You can start a **new instance of an Activity** by passing an Intent to startActivity().
2. You can **start a service** to perform a one-time operation (such as download a file) by passing an Intent to startService().
3. To **deliver a broadcast**, a broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to sendBroadcast()



Intents Types

- There are two types of intents: **Explicit and Implicit** Intents

- **Explicit intents specify the component** to start by name (the fully-qualified class name). You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, start a new activity in response to a user action or start a service to download a file in the background.

- Implicit intents do not name a specific component, but instead **declare a general action to perform**, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

Implicit Intents

- When you create an implicit intent, the Android system finds the appropriate component to start by **comparing the contents of the intent to the intent filters declared in the manifest file** of other apps on the device.
- If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

- An intent filter is an expression in an app's manifest file that **specifies the type of intent that the component would like to receive**. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent. Likewise, if you do not declare any intent filters for an activity, then it can be started only with an explicit intent.

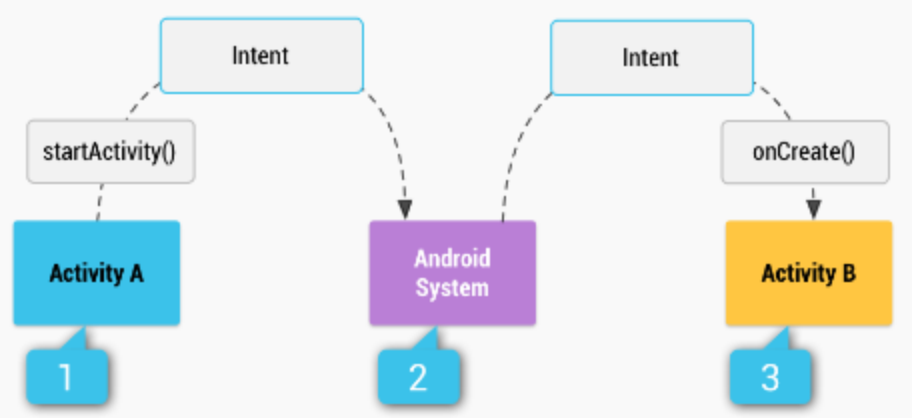


Figure 1. Illustration of how an implicit intent is delivered through the system to start another activity: **[1]** *Activity A* creates an `Intent` with an action description and passes it to `startActivity()`. **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its `onCreate()` method and passing it the `Intent`.

Building an Intent Object

- An **Intent object carries information** that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), plus information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon).The Intent object will send the following kind of information to the components

  - **component name** of the component (class) which needs to process the intent
  - The **action** that needs to be performed
  - The **data** that the action needs to operate on
  - The **type** of data (a MIME type) that is being processed
  - The **category** that this processing falls under
  - Any **flags** and **extras** that are needed to further define this processing that needs to be performed.

Example Explicit Intent

- If you built a service in your app, named DownloadService, designed to download a file from the web, you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context

//The fileUrl is a string URL, such as
//http://www.example.com/image.png

Intent downloadIntent = new Intent(this,DownloadService.class);

downloadIntent.setData(Uri.parse(fileUrl));

startService(downloadIntent);
```

- The Intent(Context, Class) constructor supplies the app Context and the component Class object. As such, this intent explicitly starts the DownloadService class in the app.

Example Implicit Intent

- Using an implicit intent is useful when **your app cannot perform the action**, but other apps probably can and you'd like the user to pick which app to use.
- For example, if you have **content you want the user to share** with other people, create an intent with the ACTION_SEND action and add extras that specify the content to share. When you call startActivity() with that intent, the user can pick an app through which to share the content.
- Caution: It's possible that a user won't have any apps that handle the implicit intent you send to startActivity(). If that happens, the call will fail and your app will crash. **To verify that an activity will receive the intent**, call resolveActivity() on your Intent object. If the result is non-null, then there is at least one app that can handle the intent and it's safe to call startActivity(). If the result is null, you should not use the intent.

```
// Create the text message with a string
Intent sendIntent = newIntent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType(HTTP.PLAIN_TEXT_TYPE); // "text/plain" MIME type
// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
startActivity(sendIntent);
}
```

Passing Data between Activities

- We seen already how we launch one Activity from another. But it is common to want to pass data between them too.
- The Intent object allows us to do this through its **Intent.putExtra()** series of methods in the calling Activity and the Intent.get<type>Extra() method in the new Activity.
- So imagine you want to send a String from one Activity to another.

```
//from sending Activity
String key = "message";
String value = "Greetings from Main Activity";
Intent sendIntent = newIntent(this, NewActivity.class);
sendIntent.putExtra(key, value);
// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) !=
null) {    startActivity(sendIntent);
//in receiving Activity
String value =
getIntent().getExtras().getString("message");
```

Passing Data between Activities

- There are many other **overloaded variants** of putExtra() within the Intent class. For example, there is one for ints, floats, arrays etc but what if you want to send a **custom object**, an object you have created in your app for example.
- For this you can use **Serializeable objects**, i.e. objects that implement the serializable interface.
- Serializing an object basically means it is packaged into a **bytestream** so it can be transported somewhere, in this case to a new activity. When it reaches its destination it can be deserialized.
- Doing this in Android is quite straightforward. Assume you have a custom class called Person that you want to send to a new Activity.

Passing Data between Activities

- 1. You must first make this Person class implement the Serializable interface

```
import java.io.Serializable;

public class Person implements Serializable {..}
```

- 2. Use the **putExtra()** method of the Intent to add this object.

```
Person p = new Person();

Intent i = new Intent(this, NewActivity.class);
i.putExtra("PersonObject", p);

startActivity(i);
```

3. Then in the new Activity, retrieve the object by

```
Intent i = getIntent();

Person p = (Person)i.getSerializableExtra("PersonObject");
```

Passing Data between Activities

- Another option is to use Application variables (like global variables). Here you must extend the application class and add in some global variables.

```
public class MyApplication extends Application {
        private String someVariable;
        public String getSomeVariable() {
                return someVariable;
        }
        public void setSomeVariable(String someVariable)
        {
         this.someVariable = someVariable;
        }
}
```

- Then in your android manifest you must declare the class implementing the Application (android.app.Application) by adding the 'android:name="myApplication"' attribute to the existing application tag.

```
<application android:name="MyApplication" android:icon="@drawable/icon"
android:label="@string/app_name">
```

- Then within your Activity you can call

```
 ((MyApplication) this.getApplication()).setSomeVariable("foo");  or
String s = ((MyApplication) this.getApplication()).getSomeVariable();
```

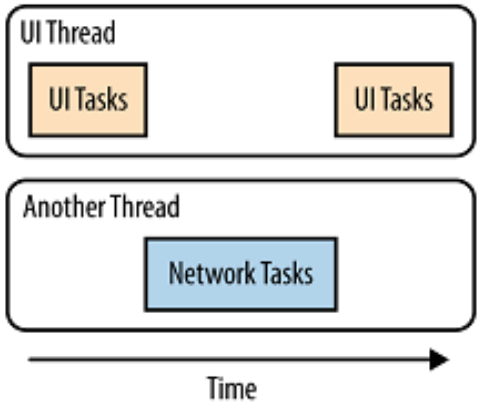
Threading in Android

- A thread is a **sequence of instructions executed in order**. Although each CPU core can process only one instruction at a time, most operating systems are capable of handling multiple threads on multiple CPU cores, or interleaving them on a single CPU.
- Different threads need different priorities, so the operating system determines how much time to give each one if they have to share a CPU.
- By default, an **Android application runs on a single thread**. Single-threaded applications run all commands serially, meaning that no command starts until the previous one is done. Another way of saying this is that each call is blocking.
- This single thread is also known as the **UI thread because** it's the thread that processes all the user interface commands. The UI thread is responsible for **drawing all the elements** on the screen as well as **processing all the user events**, such as touches on the screen, clicks of the button, and so on.
- The problem with running everything on a single thread comes when we need to do lengthy operations like network calls. As with all network calls, the time it takes to execute is outside of our control and subject to all the network availability and latency issues.

Multithreading

- A much better solution is to **run the potentially long operations on a separate thread**. When multiple tasks run on multiple threads at the same time, the operating system slices the available CPU time so that no one task dominates the execution.
- For example, we could put the a **network call i**n a **separate thread**. That way our main UI thread will not block while we're waiting for the network, and the application will appear much more responsive.

- Android provides a utility class called **AsyncTask** for running background threads that will report back to the user interface.
- You could use the standard Java threading library but there is more work involved (you have to synchronise threads to make sure the app is thread-safe).



AsyncTask

- To take advantage of this class, we need to create a new subclass of AsyncTask and implement its **doInBackground(), onProgressUpdate(), and onPostExecute()** methods.
- In other words, we are going to fill in the blanks for what to do in the background, what to do when there's some progress, and what to do when the task completes.

Perform Network Operation on Separate Thread

```
// Uses AsyncTask to create a task away from the main UI thread. This task takes
// URL string and uses it to create an HttpUrlConnection. Once the connection
// has been established, the AsyncTask downloads the contents of the webpage as
// an InputStream. Finally, the InputStream is converted into a string, which is
// displayed in the UI by the AsyncTask's onPostExecute method.
private class DownloadWebpageTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... urls) {

        // params comes from the execute() call: params[0] is the url.
        try {
            return downloadUrl(urls[0]);
        } catch (IOException e) {
            return "Unable to retrieve web page. URL may be invalid.";
        }
    }
    // onPostExecute displays the results of the AsyncTask.
    @Override
    protected void onPostExecute(String result) {
        textView.setText(result);
    }
}
```

AsyncTask

- To run the **AsyncTask we instantiate it and call execute()** on it. The argument that we pass in is what goes into the doInBackground() call.
- Notice the use of Java generics to describe the data types that this AsyncTask will use in its methods. The first data type is used by doInBackground(), the second by onProgressUpdate(), and the third by onPostExecute().
- doInBackground() is the callback that **specifies the actual work to be done** on the separate thread, acting as if it's executing in the background. The argument String… is the first of the three data types that we defined in the list of generics for this AsyncTask class. The three dots indicate that this is an array of Strings, and you have to declare it that way, even though you want to pass only a single one.
- onProgressUpdate() is used to **display any form of progress** in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
- onPostExecute() is invoked on the UI thread **after the background computation finishes**. The result of the background computation is passed to this step as a parameter
- Lets take a look at some of the networking now …

Networking in Android

- Most network-connected Android apps use **HTTP to send and receive data**. Android includes two HTTP clients: HttpURLConnection and Apache HttpClient. Both support HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling.

- To perform network operations in android make sure your application manifest includes the following permissions

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

- The steps invloved are typically,

1. Check the network connection to make sure there is one
2. Create a new thread using AsyncTask to perform the network operations
3. Connect and download the data
4. Convert the input stream (to a string or an image etc).

Check the network connection

- Before attempting to connect to the network, we should **check to see if a network connection exists** by using getActiveNetworkInfo() and isConnected().

```
...
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
if (networkInfo != null && networkInfo.isConnected()) {
    // fetch data
} else {
    // display error
}
```

- Note that we need to create a ConnectivityManager first and from this we can get a NetworkInfo object to reveal information about the network connection.
- The **ConnectivityManager is responsible for monitoring network connections**, switching networks and sending out broadcast intents when network connectivity changes.
- Note: the networking classes will be in java.net and android.net so you must import these packages.

Perform Network Operation on Separate Thread

```
// Uses AsyncTask to create a task away from the main UI thread. This task takes
// URL string and uses it to create an HttpUrlConnection. Once the connection
// has been established, the AsyncTask downloads the contents of the webpage as
// an InputStream. Finally, the InputStream is converted into a string, which is
// displayed in the UI by the AsyncTask's onPostExecute method.
private class DownloadWebpageTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... urls) {

        // params comes from the execute() call: params[0] is the url.
        try {
            return downloadUrl(urls[0]);
        } catch (IOException e) {
            return "Unable to retrieve web page. URL may be invalid.";
        }
    }
    // onPostExecute displays the results of the AsyncTask.
    @Override
    protected void onPostExecute(String result) {
        textView.setText(result);
    }
}
```

```
String stringUrl = urlText.getText().toString();
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
if (networkInfo != null && networkInfo.isConnected()) {
    new DownloadWebpageTask().execute(stringUrl);
} else {
    textView.setText("No network connection available.");
}
```

Connect and Download

• In the previous example we seen how doInBackground() executes the method downloadUrl(). It passes the web page URL as a parameter. The method downloadUrl() fetches and processes the web page content. When it finishes, it passes back a result string. onPostExecute() takes the returned string and displays it in the UI.

•Now we will have a look at downloadUrl, this is where the networking action happens.

Connect and Download

```
// Given a URL, establishes an HttpUrlConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

    // Makes sure that the InputStream is closed after the app is
    // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
```

- We use HttpURLConnection to perform a GET request and download the data. After we call connect(), we get an InputStream to the data by calling getInputStream().

- Note that the method getResponseCode() returns the connection's status code. This is a useful way of getting additional information about the connection. A status code of 200 indicates success.

- The conn.setDoInput(T/F) method specifies whether this URLConnection allows receiving data

Convert the InputStream

• An InputStream is a readable source of bytes. Once you get an InputStream, it's common to decode or convert it into a target data type. For example, if you were downloading image data, you might decode and display it like this:

```
InputStream is = null;
...
Bitmap bitmap = BitmapFactory.decodeStream(is);
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

• And if we want to convert to a simple String ..

```
// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len) throws IOException,UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

Posting Content

• HttpURLConnection uses the GET method by default. It will use POST if setDoOutput(true) has been called. For example, to perform an upload:

```
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
try {
  urlConnection.setDoOutput(true);
  urlConnection.setChunkedStreamingMode(0);

  OutputStream out = new BufferedOutputStream(urlConnection.getOutputStream());
  writeStream(out);

  InputStream in = new BufferedInputStream(urlConnection.getInputStream());
  readStream(in);
 finally {
  urlConnection.disconnect();
}
}
```

• setChunkedStreamingMode(int) is used to stream the body in chunks as apposed to buffering the complete request body in memory before it is transmitted, wasting (and possibly exhausting) heap and increasing latency. Note the zero argument here means use a default chunk size (e.g. 1024 bytes)

Persistent Data

- Most Android apps need to save data, even if only to save information about the app state during onPause() so the user's progress is not lost.

- Most non-trivial apps also need to save user settings, and some apps must manage large amounts of information in files and databases.

- We will now look at the principal data storage options in Android:

1. Saving key-value pairs of simple data types in a shared preferences file

2. Saving arbitrary files in Android's file system

3. Using databases managed by SQLite

Saving Key-Value Pairs

• If you have a small collection of key-value pairs you'd like to save, you should use the SharedPreferences APIs.

• A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them.

• You can create a new shared preference file or access an existing one by calling one of two methods:

• getPreferences() — Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

• getSharedPreferences() — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter.

SharedPreferences

• Here is a snippet that creates a shared preferences file that's identified by the resource string R.string.preference_file_key. It does it in private mode so the file is accessible by only your app.

```
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

• Or alternatively if the app will only have one SharedPreferences file

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

• When naming your shared preference files, you should use a name that's uniquely identifiable to your app, such as "com.example.myapp.PREFERENCE_FILE_KEY“

• If you create a shared preferences file with MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE, then any other apps that know the file identifier can access your data.

Wrting to SharedPreferences

• To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on the SharedPreferences object then pass the keys and values you want to write to the methods such as putInt() and putString(). Then call commit() to save the changes.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

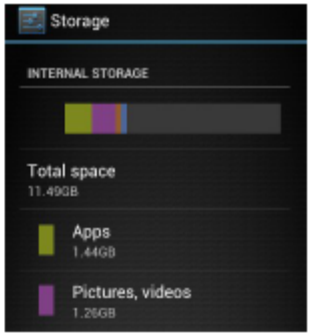
• To read values from a shared preferences file, call methods such as getInt() and getString(), providing the key for the value you want, and optionally a default value to return if the key isn't present.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.string.saved_high_score_default);
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), defaultValue)
```

Internal and External Storage

- Android can read/write files from two locations:
  - **internal** and **external** storage.
  - Both are **persistent** storage; data remains after power-off / reboot.

- **internal storage**: Built into the device.
  - guaranteed to be present
  - typically smaller (~1-4 gb)
  - can't be expanded or removed
  - specific and private to each app
  - wiped out when the app is uninstalled



Files and Streams

- java.io.**File** - Objects that represent a file or directory.
  - methods: canRead, canWrite, create, delete, exists, getName, getParent, getPath, isFile, isDirectory, lastModified, length, listFiles, mkdir, mkdirs, renameTo

- java.io.Input**Stream**, OutputStream - Stream objects represent flows of data bytes from/to a source or destination.
  - Could come from a file, network, database, memory, ...
  - Normally not directly used; they only include low-level methods for reading/writing a byte (character) at a time from the input.
  - Instead, a stream is often passed as parameter to other objects like java.util.**Scanner**, java.io.**BufferedReader**, java.io.**PrintStream** to do the actual reading / writing.

Using Internal Storage

- An activity has methods you can call to read/write files:
  - `getFilesDir()` - returns internal directory for your app
  - `getCacheDir()` - returns a "temp" directory for scrap files
  - `getResources().openRawResource(R.raw.id)` - read an input file from `res/raw/`
  - `openFileInput("name", mode)` - opens a file for reading
  - `openFileOutput("name", mode)` - opens a file for writing

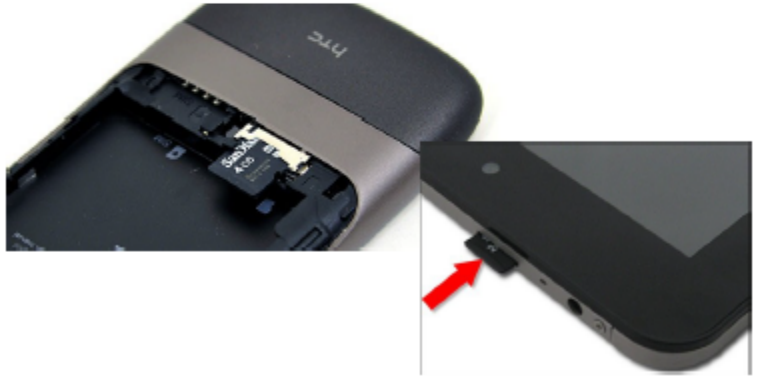
- You can use these to read/write files on the device.
  - many methods return standard java.io.**File** objects
  - some return java.io.**InputStream** or **OutputStream** objects, which can be used with standard classes like Scanner, BufferedReader, and PrintStream to read/write files (see Java API)

Using Internal Storage – Example

```
// write a short text file to the internal storage
PrintStream output = new PrintStream
        openFileOutput("out.txt", MODE_PRIVATE));
output.println("Hello, world!");
output.println("How are you?");
output.close();
...
// read the same file, and put its contents into a TextView
Scanner scan = new Scanner(
        openFileInput("out.txt", MODE_PRIVATE));
String allText = "";   // read entire file
while (scan.hasNextLine()) {
    String line = scan.nextLine();
    allText += line;
}
myTextView.setText(allText);
scan.close();
```

External Storage

- **external storage**: Card that is inserted into the device.
  (such as a MicroSD card)
  - can be much larger than internal storage (~8-32 gb)
  - can be removed or transferred to another device if needed
  - may not be present, depending on the device
  - read/writable by other apps and users; not private to your app
  - *not* wiped when the app is uninstalled, except in certain cases



External Storage - Permission

- If your app needs to read/write the device's external storage, you must explicitly request **permission** to do so in your app's **AndroidManifest.xml** file.
  - On install, the user will be prompted to confirm your app permissions.


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

Using External Storage

- Methods to read/write external storage:
  - `getExternalFilesDir("name")` - returns "private" external directory for your app with the given name
  - `Environment.getExternalStoragePublicDirectory(name)` - returns public directory for common files like photos, music, etc.
    - pass constants for ***name*** such as Environment.DIRECTORY_ALARMS, DIRECTORY_DCIM, DIRECTORY_DOWNLOADS, DIRECTORY_MOVIES, DIRECTORY_MUSIC, DIRECTORY_NOTIFICATIONS, DIRECTORY_PICTURES, DIRECTORY_PODCASTS, DIRECTORY_RINGTONES

- You can use these to read/write files on the external storage.
  - the above methods return standard java.io.**File** objects
  - these can be used with standard classes like Scanner, BufferedReader, and PrintStream to read/write files (see Java API)

External Storage - Example

```
// write short data to app-specific external storage
File outDir = getExternalFilesDir(null);   // root dir
File outFile = new File(outDir, "example.txt");
PrintStream output = new PrintStream(outFile);
output.println("Hello, world!");
output.close();

// read list of pictures in external storage
File picsDir =
        Environment.getExternalStoragePublicDirectory(
                Environment.DIRECTORY_PICTURES);
for (File file : picsDir.listFiles()) {
    ...
}
```

Is Storage Available?

```
/* Checks if external storage is available
 * for reading and writing */
public boolean isExternalStorageWritable() {
    return Environment.MEDIA_MOUNTED.equals(
            Environment.getExternalStorageState());
}

/* Checks if external storage is available
 * for reading */
public boolean isExternalStorageReadable() {
    return isExternalStorageWritable() ||
            Environment.MEDIA_MOUNTED_READ_ONLY.equals(
                Environment.getExternalStorageState());
}
```

External Storage – Tidying Up

• You can query the system to see how much free space there is also before you write files to storage. The methods getFreeSpace() and getTotalSpace() in the File class provide the current available space and the total space in the storage volume, respectively. If this is nearing capacity you should not write your data.

• You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call delete() on itself i.e. myFile.delete(); where myFile is a File object.

Saving Data in SQL Databases

• Saving data to a database is ideal for repeated or structured data, such as contact information. The APIs you'll need to use on Android are available in the android.database.sqlite package.

• To create a database to store Student Records in Android we can use

```
SQLiteDatabase db=openOrCreateDatabase("StudentDB",
Context.MODE_PRIVATE, null);
```

• The first parameter of this method specifies the name of the database to be opened or created. The second parameter, Context.MODE_PRIVATE indicates that the database file can only be accessed by the calling application. The third parameter is a Cursor factory object which can be left null if not required.

• To execute the SQL we use the db.execSQL() method. Here it is used to create the student table if it does not already exist in the database.

```
db.execSQL("CREATE TABLE IF NOT EXISTS student(rollno VARCHAR,name
VARCHAR,marks VARCHAR);");
```

SQL commands

- Here we have some of the common SQL commands (INSERT, DELETE UPDATE) all executed using the execSQL method.

- Note here we are assuming the existence of some EditText views where the data is coming from, like editName, editRollno, editMarks etc.

```
db.execSQL("INSERT INTO student VALUES('"+editRollno.getText()+"','"+
editName.getText()+"','"+editMarks.getText()+"');");

db.execSQL("DELETE FROM student WHERE rollno='"+editRollno.getText()+"'");

db.execSQL("UPDATE student SET name='"+editName.getText()+"',marks='"+
editMarks.getText()+"' WHERE rollno='"+editRollno.getText()+"'");
```

The SELECT query

• To view a student record we use the select statement but with a different method in the SQLiteDatabase class, rawQuery()

```
Cursor c = db.rawQuery("SELECT * FROM student WHERE
rollno='"+editRollno.getText()+"'", null);
if(c.moveToFirst())
{
        editName.setText(c.getString(1));
        editMarks.setText(c.getString(2));
}
```

• The above code uses the rawQuery() method of the SQLiteDatabase class to execute the SELECT statement to select the record of the student, whose roll number is specified.

• The Cursor object is a **record pointer in the database**, it points to the current row or record of the returned result set. When the moveToFirst() method of the cursor class is called it places the read position at the first entry in the results. Of course if this returns false then the Cursor is empty.

Selecting multiple records

• To view all or multiple student records we use the select statement with the '*' wildcard and then use the moveToNext() method of the Cursor class to move through the records. showMessage(..) here is simply a native method to show a message like a Toast.

```
Cursor c=db.rawQuery("SELECT * FROM student", null);
if(c.getCount()==0)
{
        showMessage("Error", "No records found");
        return;
}
StringBuffer buffer=new StringBuffer();
while(c.moveToNext())
{
        buffer.append("Rollno: "+c.getString(0)+"\n");
        buffer.append("Name: "+c.getString(1)+"\n");
        buffer.append("Marks: "+c.getString(2)+"\n\n");
}
showMessage("Student Details", buffer.toString());
```

A SQLiteDatabase App - from Code Project