INTRO TO NETWORK PROGRAMMING: USING ANDROID WITH THE INTERNET (Chapter 23)

Lecture Summary

- Getting network permissions
- Working with the HTTP protocol
  - Sending HTTP requests
  - Getting results
- Parsing HTTP results
- Parsing JSON

Network Prerequisites (1)

- The **<uses-permission>** element must be included in the AndroidManifest.xml resource so as to allow the application to connect to the network
- Permissions are used to ask the operating system to access any privileged resource

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

Network Prerequisites (2)

- The **<uses-permission>** tag causes the application to request to use an Android resource that must be authorized
  - The tag must be an immediate child of **<manifest>**

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

Protocols (Introduction)

- Android supports several different network protocols.
  - TCP / IP (through the **`Socket`** class)
  - SMTP (through the **`GMailSender`** class)
  - HTTP
  - And others
- Here we will look at HTTP

Checking for Network Access

- The network might be unavailable for many reasons
- We may have WiFi service or just cell service
- Note that service might be metered so we need to limit bandwidth
- We will only get to the basics in this course related to bandwidth management and power consumption best practices

The **ConnectivityManager**

- The **ConnectivityManager** answers questions about the state of the network
- It can be used to notify applications when the connectivity status changes
- Fail over to other networks when connectivity is lost
- Allow applications to select and query

```
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
```

The **ConnectivityManager**

- The **getSystemService(Context.CONNECTIVITY_SERVICE)** method gets information about the service stored properties of the **ConnectivityManager**

The `NetworkInfo` Class (1)

- The `ConnectivityMannager.getActiveNetworkInfo()` method returns an `NetworkInfo` instance
  - Call the `isConnected` method to determine whether there is a network connection
  - See http://developer.android.com/reference/android/net/NetworkInfo.html

The **NetworkInfo** Class (2)

- Example
- If there is no default network, **NetworkInfo** is null
- Always call **isConnected** to determine whether there is a viable connection

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

(Re)introduction to Multithreading

- A device must:
  - Respond to user requests (your interaction)
  - Listen for and respond to various notifications (from other applications)
- The Android OS services application requests based on processes and threads
  - In a nutshell, the OS services applications with work to do, based on a queue

Introduction to Multithreading

- Service requests can take a while to complete
- Some requests might not complete at all
- A **thread** is a concurrent unit of execution
  - There is a thread for the UI (foreground)
  - We can create other threads (background)
  - There are other threads already running (background and other applications)
- This gets complicated so we'll do just enough here to create a thread and run it

The **`AsyncTask`** Class (Introduction)

- The **`android.os.AsyncTask`** allows background operations to be performed on a background thread and data returned to the UI thread
- It is a helper class that wraps the **`Thread`** class
  - It hides the details of managing a **`Thread`**
  - Use it to perform short-lived background tasks

The **AsyncTask** Class (Creating)

- To use the **AsyncTask** create a subclass with three generic parameters
  - The first contains the data passed to the thread (params…)
  - The second, if used, contains an integer used to report the thread's progress (progress…)
  - The third, if used, contains the data type of the thread's return value (result)
  - If a parameter is void, then it will not be used

The **AsyncTask** Class (Example)

- Extend **AsyncTask**

```
private class DownloadWebpageTask extends AsyncTask<String, Void, String> {
```

  - A **String** is passed to the task
  - There is no progress reporting (the 2nd argument is **Void**)
  - A **String** is returned from the task

When the background task is executed, it goes through four steps

The **AsyncTask** Class (Implementing – Introduction)

- We implement methods that execute
  - Before the thread starts (**onPreExecute**)
  - To start the thread and pass data to the thread (**doInBackground**)
  - To report thread progress (**onProgressUpdate**)
  - To return data from the thread (**onPostExecute**)
- These methods are required by the abstract class (interface)

The **`AsyncTask`** Class (Implementing – Step 1)

- **`onPreExecute`** is invoked on the UI thread before the task begins
  - It is here that you setup the task
  - Initialize a progress meter, for example
- Implementing this step is optional

The **AsyncTask** Class (Implementing – Step 2)

- **doInBackground** is the worker
- It executes immediately after **onPreExecute** finishes
- It is here that you perform the background computation
  - Or in our case network access
- The parameters are passed to this procedure
- The results are returned in this step
- Call **publishProgress(Progress…)** to publish progress results (optional)

The **AsyncTask** Class (Implementing – Step 2)

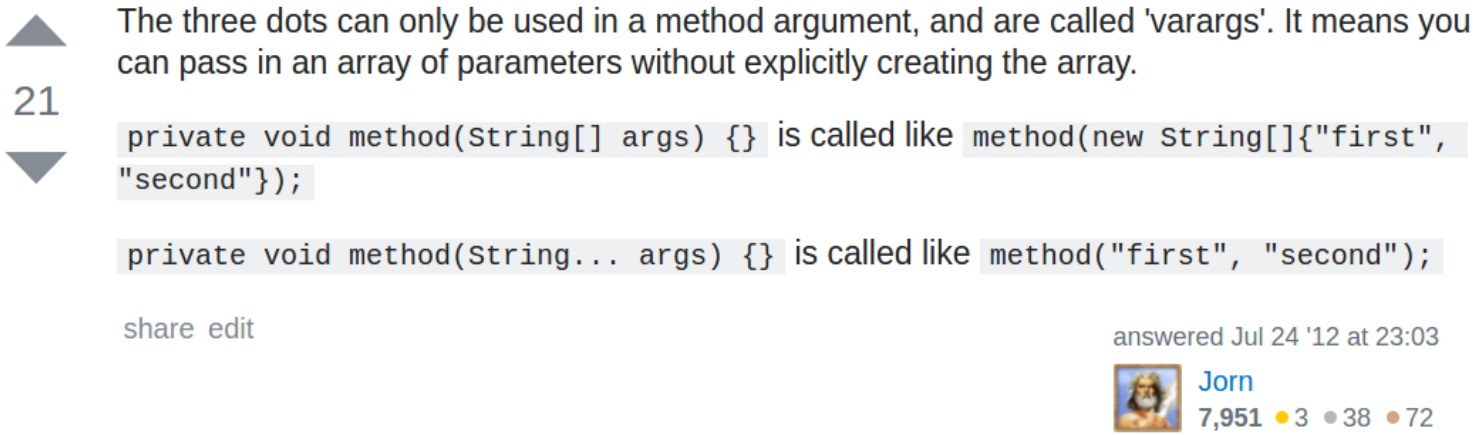
```
private class DownloadWebpageTask extends AsyncTask<String, Void, String> {

    private static final String DEBUG_TAG = null;


    @Override
    protected String doInBackground(String... urls) {

       // params comes from the execute() call: urls[0] is the url.
       try {
           return downloadUrl(urls[0]);
       } catch (IOException e) {
           return "Unable to retrieve web page. URL may be invalid.";
       }
   }
```

The **AsyncTask** Class (Implementing – Step 2)

```
private class DownloadWebpageTask extends AsyncTask<String, Void, String> {

    private static final String DEBUG_TAG = null;

                                                    ???

    @Override
    protected String doInBackground(String... urls) {

        // params comes from the execute() call: urls[0] is the url.
        try {
            return downloadUrl(urls[0]);
        } catch (IOException e) {
            return "Unable to retrieve web page. URL may be invalid.";
        }
    }
```

Language note: the ... (three dots) notation

- "Three dots" notation, also called **varargs**, is common in modern programming languages to support an open-ended number of arguments



21

The three dots can only be used in a method argument, and are called 'varargs'. It means you can pass in an array of parameters without explicitly creating the array.

`private void method(String[] args) {}` is called like `method(new String[]{"first", "second"});`

`private void method(String... args) {}` is called like `method("first", "second");`

share edit

answered Jul 24 '12 at 23:03

Jorn

7,951 ● 3 ● 38 ● 72

Varargs in Python

```
def test_var_args(farg, *args):
    print "formal arg:", farg
    for arg in args:
        print "another arg:", arg

test_var_args(1, "two", 3)
```

Results:

```
formal arg: 1
another arg: two
another arg: 3
```

The **`AsyncTask`** Class (Implementing – Step 3)

- **`onProgressUpdate`** is invoked on the UI thread
  - It gets called as a result of the **`publishProgress`** call
  - Use it to update a progress meter or log
  - Implementing this method is optional
    - It's really only useful when we can estimate progress
    - What if I don't know how big a page is when I request it

The **AsyncTask** Class (Implementing – Step 4)

- **onPostExecute** is invoked on the UI thread
- The parameter contains the result of the asynchronous method call

```
// onPostExecute displays the results of the AsyncTask.
@Override
protected void onPostExecute(String result) {
    //textView.setText(result);
}
```

STATUS

- You now know how to set up the asynchronous infrastructure
- Net we will see how to send a HTTP request with a URL and process the result

HTTP (Introduction)

- In our case the transfer protocol is HTTP
  - We connect the client device to a server and get data
  - We then process that data somehow
    - We might render a Web page
    - We might parse and process XML
    - Or any other message

HTTP (Introduction)

- Two HTTP clients
  - **`HttpClient`**
  - **`HttpURLConnection`**
- Both support HTTPS and IPV6
- Use **`HttpURLConnection`** for post Lollypop devices

Android URL and HTTP Related Classes

- Note that there are both Java and Android URL classes
- There are also URI classes

The **URL** Class

- The **`java.net.URL`** class represents a url
  - Convert strings to URLs
  - Convert URLs to strings
- http://developer.android.com/reference/java/net/URL.html

The **URL** Class

Protocol	http
Authority	username:password@host:8080
User Info	username:password
Host	host
Port	8080
File	/directory/file?query
Path	/directory/file
Query	query
Ref	ref

Opening a Connection (1)

- The **`URL.openConnectcion()`** method establishes a connection to a resource
- Over this connection, you make the request and get the response
- We will use HTTP here but other protocols are supported

Opening a Connection (2)

- The **`setReadTimeout()`** mutator defines the time to wait for data
- The **`setConnectTimeout()`** mutator the time to wait before establishing a connection
- The **`setRequestMethod()`** defines whether the request will be a GET or a POST
- The **`setDoInput()`** mutator, if true allows receiving of data

Opening a Connection (3)

- Calling the **`connect()`** method opens the connection
- **`getResponseCode()`** gets the HTTP response code from the server
  - -1 if there is no response code.
  - Such as 404 not found?
- **`getInputStream()`** gets the input stream from which you can read data
  - Works just like an open file

Opening A Connection (Example)

```
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);

    // Start the query.
    conn.connect();
    int response = conn.getResponseCode();
    Log.d(DEBUG_TAG, "The response is: " + response);
    is = conn.getInputStream();
```

Reading the Input

```
// 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);
   }
}
```

Parsing JSON

- **JSON**: **J**ava**S**cript **O**bject **M**odel
- Nowadays used as a lightweight replacement for XML (data transfer)

```
<Person>
    <FirstName>Homer</FirstName>
    <LastName>Simpsons</LastName>
    <Relatives>
        <Relative>Grandpa</Relative>
        <Relative>Marge</Relative>
        <Relative>The Boy</Relative>
        <Relative>Lisa</Relative>
        <Relative>I think that's all of them</Relative>
    </Relatives>
</Person>
```

```
{
    "firstName": "Homer",
    "lastName": "Simpson",
    "relatives": [ "Grandpa", "Marge", "The Boy", "Lisa", "I think that's all of them" ]
}
```

http://stackoverflow.com/questions/2620270/what-is-the-difference-between-json-and-xml

REST

- XML and JSON are merely *formats* (languages). We still need a **protocol** (rules of the game) to specify **clients** (like Android devices) will interact with **servers** (like weather updates).
- **REST**: **RE**presentational **S**tate **T**ransfer ("RESTful" web services)
- A **stateless** protocol: server doesn't keep track of the "conversation".

http://stackoverflow.com/questions/2620270/what-is-the-difference-between-json-and-xml