CSE 1710

Lecture 21

Net-Centric Programming, Part II

Learning Outcomes

- understand net-centric functionality in terms of client and server roles
- see how the Internet Protocol Suite is an example of layered abstraction
- distinguish between the WWW and the Internet
- Understand what the `URL` class encapsulates
- Understand what the `URLConnection` encapsulates
- programmatically get static content from a URL

Learning Outcomes

- Understand and describe the basics of the HTTP protocol
- Use the `URL` and `URLConnection` classes to:
  - to instantiate useful objects
  - to retrieve content from web servers
    - Use string processing to manipulate query strings
- Understand the concept of a class hierarchy
  - run-time checking using instanceof
- Use the `HttpURLConnection` class to examine the request and response messages

Setting the stage…

so you've sucessfully invoked **L19_App1** and **L19_App2**

What do all protocols have in common?

- All protocols provide a means to establish a connection to the remote object
- With Java SDK, the connection to a remote object *is abstracted away* from its URL
  - the connection is encapsulated by the service: `URLConnection`
  - in the **generic** case of any protocol, the connection is encapsulated by `URLConnection`
  - in the **particular** case of the HTTP protocol, the connection is encapsulated by `HttpURLConnection`
  - for any protocol, the particular method for establishing a connection is: `openConnection()`

Recap

- URLs – a specification that basically tells you:
  - "hey, here is this remote **resource** that you can access" and
  - "hey, here is the specific **protocol** that you can use to access it"
- What are some examples of remote resources?
- What are some examples of protocols?

What do <u>all</u> protocols have in common?

- All protocols provides a means to establish a connection to the remote object
- **All** protocols have a way to *"open the connection"* to *"get the remote object"*
  - Once a connection is open, we can obtain an input stream from it
    - The specifics of the "how" depends on the specific protocol
  - This is encapsulated as an InputStream object
  - Where else have we used an InputStream object?

Services we can use…

```
URL url = new URL(theURL);
```

- construct an object to encapsulate an URL

This accessor will return a reference to a `URLConnection` object:

```
url.openConnection();
```

Some URLConnections are specific to the HTTP protocol (depends on the URL!!!)

The Basics of HTTP

- HTTP stands for Hypertext Transfer Protocol
- HTTP is used to transmit resources, not just files.
- A resource is some chunk of information
  - Something that can be identified by a URL (it's the R in URL).
  - Examples of resources:
    - files
    - a dynamically-generated query result
    - the output of a CGI script
    - a document that is available in several languages.

The Basics of HTTP

- HTTP uses a model in which there is a client and a server role (the “client-server” model):
  - An HTTP **client** opens a connection and sends a request message to an HTTP **server**
  - A HTTP **server** then returns a response message to the **client**, usually containing the resource that was requested
  - After issuing the response, the **server** closes the connection.

The Basics of HTTP

- an **HTTP transaction** is defined as:
  - a single request from a **client** and the corresponding response from the **server**
- **Transaction sequence**: when a given pair of client and server has several transactions, one after another
- **Stateless:**
  - the transactions within a sequence are **independent** of one another
  - **no connection information** is maintained between transactions
  - there is no notion of “state” (e.g., the client doesn’t saves any info about how fast the server responds, etc)
  - thus, the transaction seqeunce is “stateless”
  - this is what is meant by **“http is a stateless protocol”**

The Basics of HTTP

- There are two types of messages:
  - request messages
  - response messages

- Both kinds of messages consist of:
  1. an initial line
  2. zero or more header lines
     - e.g., the “Date” field represents the date and time at which the message was originated
  3. a blank line
  4. (optionally, but not necessarily) a message body (aka “payload”)
     - e.g. a file, query data, or query output

Example – visit www.yorku.ca

here is the request

```
GET /web/index.htm HTTP/1.1                                   initial line
Host: www.yorku.ca
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:6.0.2) Gecko/20100101 Firefox/6.0.2
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5                               header lines
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Cookie: __utma=27695668.440770357.1291300324.1321558349.1322083439.188; __utmz=27695668.1321558349.187.129.
                                                              blank line
```

here is the response

```
HTTP/1.1 200 OK                                               initial line
Date: Wed, 23 Nov 2011 21:24:04 GMT
Server: Apache/1.3.34 (Ubuntu) FrontPage/4.0.4.3
Content-Type: text/html; charset=iso-8859-1                   header lines
Via: 1.1 optera.ccs.yorku.ca
Keep-Alive: timeout=15, max=93
Connection: Keep-Alive
Transfer-Encoding: chunked
                                                              blank line
the html content                                              the payload
```


OK, to illustrate…

Don't freak out…

We are going to look at the algorithm that a web server follows to serve up content

You don't have to memorize this – it's just to illustrate what is happening behind the scenes

Serving Content as per the HTTP Protocol

1. Listen on port 80
2. If and when an HTTP request arrives ("GET" or "POST"), start a process to handle it ("fork")
3. Extract the path/file from the URL
4. Check whether file exists
   If not, return status 404.
5. Check whether file is reachable & readable (file permissions?)
   If not, return status 403
6. Determine the content type (static vs dynamic)

Static Content

7. Return with status 200 (OK) and a type header.
8. Serve file as the payload.
9. Close HTTP session.
   Or, on keep-alive, wait brief time for another request.

Dynamic Content (CGI)

7. Masquerade as file owner.
8. Check that file is executable by owner.
   If not, return status 500.
9. Run the file and capture its output.
10. Check the validity of the output.
    Not valid? Return status 500.
    Valid? Return status 200 (OK), and the output as the payload.

Response Codes

100 series
  Sessional update from server.
200 series
  Success!
300 series
  Redirect.
400 series
  Client error.
500 series
  Server error.

For full detail, you can look at the full specification at:
`http://kb.globalscape.com/KnowledgebaseArticle10141.aspx`

Revisit `L19_App1`

Review the statements in light of the HTTP protocol

Discussion about `L19_App1`

We see that `L19_App1` effectively performed a single transaction

How and Where?

- there was a request message – the `openConnection()` method causes the instantiation of a `URLConnection` object
  - the `URLConnection` object, upon instantiation, attempts to establish contact to the server
  - things could go wrong, for instance an `java.net.UnknownHostException` may be thrown
  - if the connection is established, then the `URLConnection` object will issue the request message
- there was a response message
  - the server will issue a response, which the `URLConnection` object will capture
  - things could go wrong with the connection and an exception will be thrown

How can we examine the specifics of the request and response messages?

We can't!! Inadequate services - this is why we need `L19_App2`

We know the URL is HTTP, so…

we can infer that the URL connection is actually a connection established using HTTP

Approach #1 (used with `L19_App2`):

- cast the object at run-time
- **what could go wrong?**

Review: the `instanceof` operator

Recall sec 3.2.4 "Relational Operators" (p.110)

```
<     <=    >     >=    ==    !=
```

There was also the following operator:

```
instanceof
```

```
boolean test = x instanceof C;
```

the expression evaluates to true iff either:

- the object reference `x` references an instance of class `C` or
- the object reference `x` references an instance of a subclass of `C`

Another Approach

Approach #1:

- `L19_App2` is vulnerable because of **the manual cast**
- **manual casts** can potentially lead to exceptions
  - e.g. if the connection object is not actually an http connection
- this example points out the difference between early binding (p.103) and late binding (at run-time)

Approach #2:

- see `L21_App1`
- cast the object at run-time, but do so only conditionally

Another Approach

- Now that we have a `HttpURLConnection` we can access additional services
- The `HttpURLConnection` class offers the following services:

```
String : getRequestMethod()
int : getResponseCode()
String : getResponseMessage()
```