Today’s Topics

- Lab 6: Web Crawler!
- Java Sockets API
- String operations
This Week’s Assignment

- Build a simple web-crawler
  - Connect to a web server
  - Send an HTTP request to the server
  - Get the HTTP response from the server
  - Process it to find more URLs
  - Repeat!
Networking Protocols

- Two main Internet communication protocols
- TCP/IP (or just TCP)
  - Transmission Control Protocol/Internet Protocol
  - Stream-based, reliable, ordered communication
- UDP
  - User Datagram Protocol
  - Message ("datagram") based, unreliable, unordered communication
- Java supports both in `java.net` package
  - TCP: `java.net.Socket`
  - UDP: `java.net.DatagramSocket`
  - Others too... e.g. SSL (`javax.net.ssl` package)
Talking to Web Servers

- **HTTP**: Hypertext Transfer Protocol
  - Text-based protocol
  - Request/response interactions
  - Uses TCP/IP protocol

- **Connection parameters:**
  - IP address, or hostname (resolved to IP address)
  - Port (in range 1..65535; 1..1024 are reserved)

- **Different kinds of servers listen on specific ports**
  - E-mail servers typically listen to port 25
  - SSH servers typically listen to port 22
  - Web servers typically listen to port 80
Web-Page URLs

- URL = Uniform Resource Locator
- Specifies:
  - Communications protocol
  - Server’s hostname or IP address
  - Port (optional; each protocol has own default)
  - Path to document or resource (also optional)
- Example: http://www.cms.caltech.edu/people
  - Protocol is HTTP
  - Server’s hostname is www.cms.caltech.edu
  - Port defaults to 80 for HTTP servers
  - Resource on server is /people
Requesting a Web Page

- Connect to the specified host and port
  - Use `java.net.Socket` since it’s TCP
- Send an HTTP request for the desired page
- Receive HTTP response containing the page
  - …or a response saying there was an error!
- Close the socket used to connect
  - Don’t hold on to networking resources
- Do stuff with the retrieved document
  - In our case, process it to find more URLs
Connecting to the Server

- Create a new **Socket** for each connection
  - Specify hostname/IP address as a **String**
  - Specify port number
    ```java
    webServer = "www.cms.caltech.edu";
    webPort = 80;
    Socket sock = new Socket(webServer, webPort);
    ```

- **Problem:**
  - What if there’s no server by that name?
  - What if server isn’t listening on that port?

- **Socket** constructor reports connection errors by throwing exceptions
Interacting with Web Servers

- If socket can’t connect to remote server, an exception will be thrown
- Connection may fail during interaction, too
- Your web-crawler will need to catch the exceptions that could be thrown
  - Handling them can be simple – print a message indicating the error, then go on to next URL
- Use the Java API documentation to see what exceptions to handle in your program
Communicating Over the Socket

- Once socket is open, can get an `InputStream` and an `OutputStream` from it
  - `OutputStream` is for sending to remote host
  - `InputStream` is for receiving from remote host

- Problem:
  - `InputStream` and `OutputStream` not suited to text data!
  - Are designed for byte streams
  - “Read/write a byte,” or “read/write an array of bytes”
  - Won’t handle text character-sets
  - Converting byte arrays to/from `String` objects is a big pain
Readers and Writers

- **Reader, Writer** classes are for character streams
- Can wrap a **Reader** around an **InputStream**
  - **Reader** consumes bytes from **InputStream**; produces characters or strings
- Can wrap a **Writer** around an **OutputStream**
  - **Writer** takes characters; feeds bytes to **OutputStream**
- ...perfect for HTTP interactions!

- Several different subclasses of **Reader, Writer**
  - (Same with **InputStream** and **OutputStream**)
Sending HTTP Requests

- HTTP request must take form:
  ```plaintext
  GET /people HTTP/1.1
  Host: www.cms.caltech.edu
  Connection: close
  
  The blank line is required!!! 😊
  
  First line contains document/resource to fetch
  - For the root document of a website, must specify / as path
  
  Second line specifies web server hostname
  - (Multiple virtual hosts can be served from one physical server)
  
  Third line tells server to close connection when response is completely sent
Example Request-Sending Code

```java
Socket sock = new Socket(webHost, webPort);
sock.setSoTimeout(3000);  // Time-out after 3 seconds

OutputStream os = sock.getOutputStream();

// true tells PrintWriter to flush after every output
PrintWriter writer = new PrintWriter(os, true);

writer.println("GET " + docPath + " HTTP/1.1");
writer.println("Host: " + webHost);
writer.println("Connection: close");
writer.println();

// Request is sent!  Server will start responding now.
```
Receiving the HTTP Response

- Use **BufferedReader** to read lines of text from socket input
  - **BufferedReader** requires input from another **Reader**
  - Use **InputStreamReader** to convert socket’s input-stream into a reader
    ```java
    InputStream is = sock.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    ```
- Can call **br.readLine()** until it returns **null**
  - This is why we said “**Connection: close**” in the request
Example Response-Receiving Code

```java
InputStream is = sock.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);

while (true) {
    String line = br.readLine();
    if (line == null)
        break;  // Done reading document!

    // Do something with this line of text.
    System.out.println(line);
}
```
Exception Handling in the Web Crawler

- Make sure your exception handling has the right level of granularity.

- Operations for crawling a web page:
  1. Connect to remote server with a socket
  2. Send the HTTP request
  3. Read back the HTTP response
  4. Parse URLs from the response text

- All of these steps could conceivably throw an exception.
  - URL parsing may or may not, depending on your implementation
Exception Handling: A Simple Approach

- Operations for crawling a particular web page:
  1. Connect to remote server with a socket
  2. Send the HTTP request
  3. Read back the HTTP response
  4. Parse URLs from the response text

- A simple approach:
  - Wrap each step with its own try/catch block.

- Does this approach make sense?
  - If any step fails, cannot perform any subsequent steps!

- An exception from steps 1-3 should terminate the entire operation of crawling the web page
  - (If a URL doesn’t parse, just go on to next URL in page…)
Smarter Exception Handling

- Exceptions should be handled on a “per unit of work” basis

- Example:
  - A good “unit of work” for the web crawler is attempting to process a particular web page

- A better approach:
  - Put code for processing a single URL into a function
  - Within the function, operations might throw exceptions
    - The function just lets any exceptions propagate out
    - Any exception will terminate the entire unit of work
  - The function’s caller wraps the call with a try/catch block
Searching Strings

- **String** class provides many useful features
- Find the index of a character or string:
  - `int indexOf(int ch)`
  - `int indexOf(int ch, int fromIndex)`
  - `int indexOf(String str)`
  - `int indexOf(String str, int fromIndex)`
  - Also, `lastIndexOf(…)` for searching from end
- These functions return -1 if value is not found
  - Valid indexes are 0 to `length()` – 1
Manipulating Strings

- Get a substring of a String
  - String substring(int beginIndex)
  - String substring(int beginIndex, int endIndex)

- Change the case of a string:
  - String toLowerCase()
  - String toUpperCase()

- Trim whitespace off a string:
  - String trim()

- Note: Java strings are immutable
  - These operations return a new String object
Example: Searching for Words

// TODO: Get the word and line from somewhere...
String word = "after";
String line = ...;

// Search for our word in the current line.
int idx = 0;
while (true) {
    idx = line.indexOf(word, idx);
    if (idx == -1) // No more copies of word in this line
        break;

    // Record that we found another copy of the word.
    count++;

    // Skip past this copy of the word, so that next
    // iteration of the loop doesn't see it again!
    idx += word.length();
}
Searching for Links

- Links are trickier to find
  
  `<a href="http://www.caltech.edu">Caltech</a>`

  1) Search for: `a href="`
  2) Once you find that, look for the closing "`
  3) Text between the double-quotes is the URL

- Make sure to handle case where multiple URLs appear in the same line
  - After pulling out the current URL text, advance the index past it, and look for next URL.
  - Don’t need to handle links that wrap to next line
Create a simple `URLDepthPair` class to track the depth of each URL that is found.

First URL is at depth 0.

When processing a page, its URLs get created with that page’s depth + 1.
- Put new `URLDepthPair` objects into a list!
- After a page is processed, get the next URL to process from your list.

Take a second command-line argument specifying max depth to crawl a website to.

This strategy doesn’t handle cycles very cleverly…
Lists of URL-Depth Pairs

- A **LinkedList** is good for this task
  ```java
  LinkedList<URLDepthPair> pendingURLs =
      new LinkedList<URLDepthPair>();
  ```

- **When you find a new URL:**
  ```java
  pendingURLs.add(new URLDepthPair(linkText, childDepth));
  ```

- **When you need another URL to process:**
  ```java
  while (!pendingURLs.isEmpty()) {
      nextURLPair = pendingURLs.removeFirst();
      ...
      // Process this URL-depth pair
  }
  ```

- **When a URL is processed:**
  - Use another **LinkedList** to store processed URLs

- **At end of program, print out all processed URLs**
Plan for Reuse!

- Make URL-processing code reusable
  - Encapsulate it in a method or a few methods
  - This will help you with lab 6, and with lab 7!
- Next week’s lab is more powerful
  - A multithreaded version of the web-crawler
  - URLs will be processed concurrently
  - Minimize interactions with shared resources
Next Week

- All about the Java threading model
  - Can be very tricky! Make sure to attend lecture.