Programming Project: Building a Web Server

Submission Instruction:

- Group project
- Submit your code through Bb by Dec. 8, 2014 11:59 PM.
- You need to generate a simple index.html page displaying all your group members' name.
- For a Webserver, a template is given. Your group needs to complete the code. Then, generate a multi-threaded webserver.

You will develop a Web server. You should be able to demonstrate that your Web server is capable of delivering your home page to a Web browser.

We are going to implement version 1.0 of HTTP, as defined in RFC 1945, where separate HTTP requests are sent for each component of the Web page. The server will be able to handle a service requests. The server listens to a fixed port. When it receives a TCP connection request, it sets up a TCP connection through another port and services the request in a separate thread. After this program is running properly, you will add the code required to generate an appropriate response.

As you are developing the code, you can test your server from a Web browser. But remember that you are not serving through the standard port 80, so you need to specify the port number within the URL that you give to your browser. For example, if your machine's name is host.someschool.edu, your server is listening to port 6789, and you want to retrieve the file index.html, then you would specify the following URL within the browser:

http://host.someschool.edu:6789/index.html

If you omit ":6789", the browser will assume port 80 which most likely will not have a server listening on it.

When the server encounters an error, it sends a response message with the appropriate HTML source so that the error information is displayed in the browser window.

Web Server in Java: Part A

In the following steps, we will go through the code for the first implementation of our Web Server. Wherever you see "?", you will need to supply a missing detail.

Our first implementation of the Web server will be multi-threaded, where the processing of each incoming request will take place inside a separate thread of execution. This allows the server to service multiple clients in parallel, or to perform multiple file transfers to a single client in parallel. When we create a new thread of execution, we need to pass to the Thread's constructor an instance of some class that implements the Runnable interface. This is the reason that we define a separate class called HttpRequest. The structure of the Web server is shown below:

```java
import java.io.*;
import java.net.*;
import java.util.*;

public final class WebServer
{
    public static void main(String argv[]) throws Exception
    {
```
final class HttpRequest implements Runnable
{
  ...  
}

Normally, Web servers process service requests that they receive through well-known port number 80. You can choose any port higher than 1024, but remember to use the same port number when making requests to your Web server from your browser.

public static void main(String argv[]) throws Exception
{
  // Set the port number.
  int port = 6789;
  ...
}

Next, we open a socket and wait for a TCP connection request. Because we will be servicing request messages indefinitely, we place the listen operation inside of an infinite loop. This means we will have to terminate the Web server by pressing ^C on the keyboard.

// Establish the listen socket.
?

// Process HTTP service requests in an infinite loop.
while (true) {
  // Listen for a TCP connection request.
  ?
  ...  
}

When a connection request is received, we create an HttpRequest object, passing to its constructor a reference to the Socket object that represents our established connection with the client.

// Construct an object to process the HTTP request message.
HttpRequest request = new HttpRequest( ? );

// Create a new thread to process the request.
Thread thread = new Thread(request);

// Start the thread.
thread.start();

In order to have the HttpRequest object handle the incoming HTTP service request in a separate thread, we first create a new Thread object, passing to its constructor a reference to the HttpRequest object, and then call the thread's start() method.

After the new thread has been created and started, execution in the main thread returns to the top of the message processing loop. The main thread will then block, waiting for another TCP connection request,
while the new thread continues running. When another TCP connection request is received, the main
thread goes through the same process of thread creation regardless of whether the previous thread has
finished execution or is still running.

This completes the code in main(). For the remainder of the lab, it remains to develop
the HttpRequest class.

We declare two variables for the HttpRequest class: CRLF and socket. According to the HTTP
specification, we need to terminate each line of the server’s response message with a carriage return
(CR) and a line feed (LF), so we have defined CRLF as a convenience. The variable socket will be used
to store a reference to the connection socket, which is passed to the constructor of this class. The
structure of the HttpRequest class is shown below:

```java
final class HttpRequest implements Runnable
{
    final static String CRLF = "\r\n";
    Socket socket;

    // Constructor
    public HttpRequest(Socket socket) throws Exception
    {
        this.socket = socket;
    }

    // Implement the run() method of the Runnable interface.
    public void run()
    {
        . . .
    }

    private void processRequest() throws Exception
    {
        . . .
    }
}
```

In order to pass an instance of the HttpRequest class to the Thread’s constructor, HttpRequest must
implement the Runnable interface, which simply means that we must define a public method
called run() that returns void. Most of the processing will take place within processRequest(),
which is called from within run().

Up until this point, we have been throwing exceptions, rather than catching them. However, we can not
throw exceptions from run(), because we must strictly adhere to the declaration of run() in
the Runnable interface, which does not throw any exceptions. We will place all the processing code
in processRequest(), and from there, throw exceptions to run(). Within run(), we explicitly catch
and handle exceptions with a try/catch block.

```java
// Implement the run() method of the Runnable interface.
public void run()
{
    try {
        processRequest();
    } catch (Exception e) {
        System.out.println(e);
```
private void processRequest() throws Exception
{
    // Get a reference to the socket's input and output streams.
    InputStream is = ?;
    DataOutputStream os = ?;

    // Set up input stream filters.
    ?

    BufferedReader br = ?;
    . . .

    Now we are prepared to get the client's request message, which we do by reading from the socket's input stream. The readLine() method of the BufferedReader class will extract characters from the input stream until it reaches an end-of-line character, or in our case, the end-of-line character sequence CRLF.

    The first item available in the input stream will be the HTTP request line. (See Section 2.2 of the textbook for a description of this and the following fields.)

    // Get the request line of the HTTP request message.
    String requestLine = ?;

    // Display the request line.
    System.out.println();
    System.out.println(requestLine);

    After obtaining the request line of the message header, we obtain the header lines. Since we don't know ahead of time how many header lines the client will send, we must get these lines within a looping operation.

    // Get and display the header lines.
    String headerLine = null;
    while ((headerLine = br.readLine()).length() != 0) {
        System.out.println(headerLine);
    }

    We don't need the header lines, other than to print them to the screen, so we use a temporary String variable, headerLine, to hold a reference to their values. The loop terminates when the expression

    (headerLine = br.readLine()).length()

    evaluates to zero, which will occur when headerLine has zero length. This will happen when the empty line terminating the header lines is read. (See the HTTP Request Message diagram in Section 2.2 of the textbook)
In the next step of this lab, we will add code to analyze the client's request message and send a response. But before we do this, let's try compiling our program and testing it with a browser. Add the following lines of code to close the streams and socket connection.

```java
// Close streams and socket.
os.close();
br.close();
socket.close();
```

After your program successfully compiles, run it with an available port number, and try contacting it from a browser. To do this, you should enter into the browser's address text box the IP address of your running server. For example, if your machine name is `host.someschool.edu`, and you ran the server with port number 6789, then you would specify the following URL:

```
http://host.someschool.edu:6789/
```

The server should display the contents of the HTTP request message. Check that it matches the message format shown in the HTTP Request Message diagram in Section 2.2 of the textbook.