Program 2 – A Multi-Threaded Web Server (25 points)

Due Date: Nov. 1 (Friday) by 11:59pm

Program Submission: submit a runnable .jar file and the source code (*.java) to Canvas.

Program Description

You will develop a **multi-threaded Web server** that can serve multiple clients. The Web server should be able to deliver a web page to the client (a browser). The web page may include other objects such as pictures. The server runs a simplified **HyperText Transfer Protocol (HTTP)** that responds to the **GET** request message only. Port number **5520** will be used for connection requests. Each request should be handled with a separate thread.

You can test your server with a browser. But remember the server is 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, let's assume the host name is **host.someschool.edu**, the server is listening on port **5520**, and the web page to be retrieved is **index.html**. Then you must type the following URL in the browser: **http://host.someschool.edu:5520/index.html.** If you are running the server and the browser on the same machine, you would type: **http://localhost:5520/index.html.** If you omit **":5520"**, 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.

Program Requirement

1. System output must show your server is currently running. You must try-catch everything and print out appropriate error messages, to the console, identifying the specific exception. The server must not crash under any situations. The server console must display all the client activities.
2. You MUST have 3 classes implement the WebServer (the Server class), WebRequest (the ServerThread class) and the HyperText Transfer Protocol (the HTTP class.) The class implementing the HTTP is described as follows.

❖ HTTP class

This class implements a simplified version of the HyperText Transfer Protocol that handles the **GET request message** and composes the **response message** to be sent to the browser. For the request message from the Client (browser), you only need to parse the first line, and the method is always **GET**. The format of the first line of the HTTP request message is as follows.

Method	SPACE	URL	SPACE	HTTP Version	\r	\n

- **Parsing the request message**

You will read the request message from the socket's input stream and parse the request message. The **readLine()** method of the **BufferedReader** class will extract characters from the input stream until it reaches an end-of-line character, or in this case, the end-of-line character sequence (\r\n).

Extract the file name (URL) only from the request line with the **StringTokenizer** class. Because the browser precedes the filename with a slash, prefix a dot to the file name so that the resulting pathname starts within the current folder. For example, **fileName = "." + fileName;.** Next, create an instance of the **FileInputStream** as the first step of sending the file (might be a web page or other objects such as a gif file) to the client. If the file does not exist, the constructor will throw the **FileNotFoundException**. Instead of throwing this possible exception and terminating the thread, you will use a try/catch construction to set the boolean variable fileExists to false. Later in your code, you will use this flag to construct an error response message, rather than try to send a nonexistent file.

- **Composing the response message.**

There are **three parts** to the response message: the **status line**, the response **headers**, and the **entity body**. According to the HTTP specification, you need to terminate each line of the server's response message with a carriage return (**\r**) and a line feed (**\n**), so you might want to define a constant **CRLF** as a convenience. Send the response message in sequence by writing to the socket's output stream. The status line and response headers are terminated by the character sequence (\r\n), and the entity body (the HTML file) is preceded by a blank line (\r\n). The format of the status line is as follows.

Version	SPACE	Status Code	SPACE	Phrase	\r	\n

Always use "**HTTP/1.0**" as the version. The status code and the phrase are "**200 OK**" if the file is found, or "**404 Not Found**" if the requested object does not exist. If you are using **DataOutputStream,** there is a **writeBytes()** method that writes a String to the output. The format for the header line is as follows.

Header Field Name	SPACE	Value	\r	\n

Always use "**Content-type:** " as the header field name. The Value is the MIME type of the object requested. For example: **text/html**, **image/gif**, **image/bmp**, or **image/jpeg.** You could write a **contentType()** method that determines the MIME type. This method will examine the extension of a file name and return a string that represents its MIME type. If the file extension is unknown, we return the type **application/octet-stream**.

```
private String contentType(String fileName)
{
   if(fileName.endsWith(".htm") || fileName.endsWith(".html"))
      contentTypeLine = contentTypeLine + "text/html\r\n";
   if(?) ... ;
```

```
        if(?) ... ;
        return "application/octet-stream";
    } //end of contentType()
```

You must have a method that writes the requested file (entity body) onto the socket's output stream. This method may throw an IOException. You MUST write the file with a **1024-byte chunk**, meaning you must use the **read(byte[] b)** method in the **FileInputStream** class to read a 1024-byte chunk, and use the **write(byte[] b, int offset, int len)** method in the **OutputStream** class to write the chunk to the socket's output stream. NOTE: the last chunk of the file is normally less than 1024 bytes. The **read()** method returns an integer indicating the number of bytes read. To construct a 1024-byte buffer to hold bytes:

```
byte[] buffer = new byte[CHUNK_SIZE];
```

Have a while loop to keep reading/writing from/to the socket until the end of file. The **read()** method returns **-1** indicating an end of file. If the file is not found, your entity body for the response message should be:

```
entityBody = "<HTML>" + "<HEAD><TITLE>Not Found</TITLE></HEAD>"
                      + "<BODY>Not Found</BODY></HTML>";
```

NOTE: you should generate your own "404 Not Found" webpage using the above entity body in the response message.

3. Grading

Exceptions/Violations	Each Offense	Max Off
Program not running	25	25
Cannot handle multiple connections	10	10
Missing any pieces of the files displayed on the browsers	2	4
Client activities were not shown properly on the server console	0.5	3
404 not found generated by the browsers	2	2
Improper handling try/catch exceptions	0.5	2