CIS 457- Lab04
Building a Multi-Threaded Web Server
Due by: 10/4/2016
Total Points: 20 Points
Submission format: hardcopy

Lab Objectives
The purpose of this lab is to:

- Develop a Multi-Threaded Web server in two steps

Part A **(10 Points)**

In this section, you will have built a multi-threaded Web server that is capable of processing simultaneous service requests in parallel. You should be able to demonstrate that your Web server is capable of delivering the attached index.html file to a Web browser. You may read about multi-threading here: https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

We are going to implement a multi-threaded web server, as defined in RFC 1945. In the main thread, 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. To simplify this programming task, we will develop the code **in two stages**:

- In *the first stage*, you will write a multi-threaded server that simply displays the contents of the HTTP request message that it receives.
- *After the first stage is done*, 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 IP address:6789/index.html or http://127.0.0.1:6789/index.html

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

Stage 1: 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. So, the code for the application in this stage will have 2 files (1) WebServer.java and (2) HttpRequest.java For your own convenience, these files are attached with the lab assignment and explained in the following section:

➔ The structure of the <u>WebServer.java</u> is shown below:

```
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;
        int port = (new Integer(argv[0])).intValue();
        . . .
}
```

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.

          Socket connection =?;
. . .
}
```

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(?);

// Start the thread.
?;

}}}
```

************************************ **End of WebServer.java Code***

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 WebServer's 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 such as follows:

➔ The structure of the HttpRequest.java is shown below:

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. So, the structure of the *HttpRequest class* is shown below:

```
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()
        {
                . . . //Provide implementation such as given below
        }

        private void processRequest() throws Exception
        {
                . . . //Provide implementation such as given below

        }
}
```

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.

```
// Implement the run() method of the Runnable interface.
public void run()
{
        try {
                processRequest();
        } catch (Exception e) {
                System.out.println(e);
        }
}
```

Now, let's develop the code within processRequest(). We first obtain references to the socket's input and output streams. Then we wrap InputStreamReader and BufferedReader filters around the input stream. However, we won't wrap any filters around the output stream, because we will be writing bytes directly into the output stream.

```
private void processRequest() throws Exception
{

        // Set up an output stream.
        DataOutputStream os = new DataOutputStream(?);
        // Set up input stream filters.
        BufferedReader br = new BufferedReader(new InputStreamReader(?));
        . . .
}
```

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.

```
// Read 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 print the header lines.
String headerLine = null;
while ((headerLine = br.readLine()).length() ?) {
        ?
}
```

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. Add the following lines of code to close the streams and socket connection.

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

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 to direct a web request to our server.***

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 has an IP address 148.61.112.30, and you ran the server with port number 6789, then you would specify the following URL: <u>http://148.61.112.30:6789/ or http://127.0.0.1:6789/</u> The server should display the contents of the HTTP request message at the **<u>command prompt window</u>**. Then answer the following question:

Question 1 **(10 Points)**

a) Does the message match the format shown in the HTTP Request Message diagram, which is described in chapter 02 materials?
b) Provide a screen capture and the program codes used in the testing process.

Part B: **(10 Points)**

Stage 2: Web Server in Java returning a response back to client - Part B

Instead of simply terminating the thread after displaying the browser's HTTP request message, we will analyze the request and send an appropriate response. We are going to ignore the information in the header lines, and use only the file name contained in the request line. In fact, we are going to assume that the request line always specifies the GET method, and ignore the fact that the client may be sending some other type of request, such as HEAD or POST.

We extract the file name from the request line with the aid of the StringTokenizer class. First, we create a StringTokenizer object that contains the string of characters from the request line. Second, we skip over the method specification, which we have assumed to be "GET". Third, we extract the file name. You may use the power point slides (attached) for building a web server as a guide in this process. This process would require making a change in the file HttpRequest.java

```
// Extract the filename from the request line.
? tokens = new StringTokenizer(?);

tokens.nextToken();  // skip over the method, which should be "GET"
String fileName = tokens.nextToken();

// Prepend a "." so that file request is within the current directory.
?
```

Because the browser precedes the filename with a slash, we prefix a dot so that the resulting pathname starts within the current directory.

Now that we have the file name, we can open the file as the first step in sending it to the client. If the file does not exist, the FileInputStream() constructor will throw the FileNotFoundException. Instead of throwing this possible exception and terminating the thread, we will use a try/catch construction to set the boolean variable fileExists to false. Later in the code, we will use this flag to construct an error response message, rather than try to send a nonexistent file.

```
// Open the requested file.
FileInputStream fis = null;
boolean fileExists = true;
try {
        ?;

} catch (FileNotFoundException e) {
        fileExists = false;
}
```

There are three parts to the response message: *the status line, the response headers, and the entity body*. The status line and response headers are terminated by the character sequence CRLF. We are going to respond with a *status line*, which we store in the variable statusLine, and a *single response header*, which we store in the variable contentTypeLine. In the case of a request for a nonexistent file, we return *404 Not Found* in the status line of the response message, and include an error message in the form of an HTML document in the entity body.

```
// Construct the response message.
String statusLine = null;
String contentTypeLine = null;
String entityBody = null;
if (fileExists) {
        statusLine = ?;
        contentTypeLine = "Content-type: " +
                contentType( fileName ) + CRLF;
} else {
        statusLine = ?;
        contentTypeLine = ?;
```

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

When the file exists, we need to determine the file's MIME type and send the appropriate MIME-type specifier. We make this determination in a separate private method called contentType(), which returns a string that we can include in the content type line that we are constructing.

Now we can send the status line and our single header line to the browser by writing into the socket's output stream.

```
// Send the status line.
os.writeBytes(?);

// Send the content type line.
os.writeBytes(?);

// Send a blank line to indicate the end of the header lines.
os.writeBytes(?);
```

Now that the status line and header line with delimiting CRLF have been placed into the output stream on their way to the browser, it is time to do the same with the entity body. If the requested file exists, we call a separate method to send the file. If the requested file does not exist, we send the HTML-encoded error message that we have prepared.

```
// Send the entity body.
if (fileExists)		{
	sendBytes(fis, os);
	fis.close();
} else {
	os.writeBytes(?);
```

After sending the entity body, the work in this thread has finished, so we close the streams and socket before terminating.

We still need to code the two methods that we have referenced in the above code, namely, the method that determines the MIME type, contentType(), and the method that writes the requested file onto the socket's output stream. Let's first take a look at the code for sending the file to the client.

```
private static void sendBytes(FileInputStream fis, OutputStream os)
throws Exception
{
  // Construct a 1K buffer to hold bytes on their way to the socket.
  byte[] buffer = new byte[1024];
  int bytes = 0;

  // Copy requested file into the socket's output stream.
  while((bytes = fis.read(buffer)) != -1 ) {
    os.write(?, ?, ?);

  }
}
```

Both read() and write() throw exceptions. Instead of catching these exceptions and handling them in our code, we throw them to be handled by the calling method.

The variable, buffer, is our intermediate storage space for bytes on their way from the file to the output stream. When we read the bytes from the FileInputStream, we check to see if read() returns minus one, indicating that the end of the file has been reached. If the end of the file has not been reached, read() returns the number of bytes that have been placed into buffer. We use the write()

method of the OutputStream class to place these bytes into the output stream, passing to it the name of the byte array, buffer, the starting point in the array, 0, and the number of bytes in the array to write, bytes.

The final piece of code needed to complete the Web server is a method that will examine the extension of a file name and return a string that represents it's MIME type. If the file extension is unknown, we return the type application/octet-stream.

```
private static String contentType(String fileName)
{
        if(fileName.endsWith(".htm") || fileName.endsWith(".html")) {
                return "text/html";
        }

        if(fileName.endsWith(".gif")) {
           return "image/gif";
        }

        if(?) {
                ?;
        }


        return "application/octet-stream";
}
```

There is a lot missing from this method. For instance, nothing is returned for GIF or JPEG files. You may want to add the missing file types yourself, so that the components of your home page are sent with the content type correctly specified in the content type header line. For GIFs the MIME type is image/gif and for JPEGs it is image/jpeg.

This completes the code for the second phase of development of your Web server. Now, run the server. And open a browser to direct a web request to the server to service this web request http://127.0.0.1:6789/index.html (Assuming that your server will still listen on a port number 6789. When you connect to the running web server, examine the GET message requests that the web server receives from the browser.

Question 2 **(10 Points)**

a) What are the benefits from writing data to a buffer instead of directly to a stream?
b) Provide a screen capture and the program codes for the testing process.

Acknowledgements

The ideas here in this lab are partially taking from Computer Networking: A Top-Down Approach Featuring the Internet, by Kurose & Ross Addison Wesley.