Proxy Server

Computer Network – Project #3

2018008904 이성진

2019-11-30

Proxy Server: Table of Contents

1. Overview

Project # 3 aims to implement a small proxy server that can cache web pages. This proxy server can only recognize GET requests, but can handle objects such as images as well as HTML pages. In addition, the proxy server must be able to receive requests, send them to the server, then receive the response and send it back to the client.

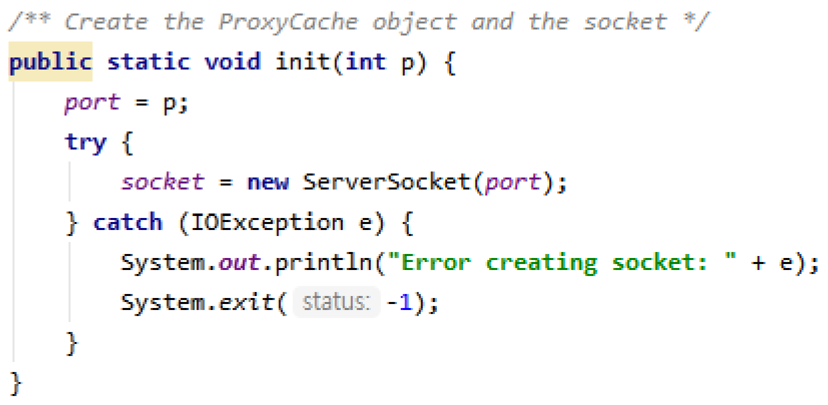
For this project, I implemented four Java class files, ProxyCache, ThreadHandler, HttpRequest, and HttpResponse. The roles and operation of these classes are as follows. Each of the classes also contains some implementations of the optional objectives specified in the assignment description (Better error handling and caching), which will be explained further at the end.

2. Source Review

2.1. ProxyCache.java

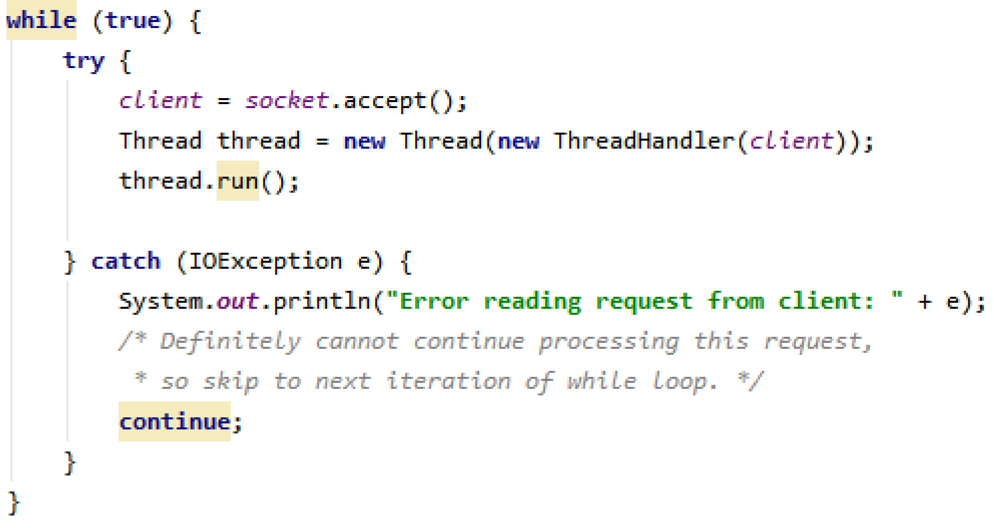
ProxyCache.java is the main class that contains the main method. This class owns the main and init methods. The main method receives the port number of the proxy server as a command argument. After that, it is passed as a parameter of init and init starts the socket connection by using the input port number. If the port number is entered correctly and the socket connection is successful, create a new thread to process the requests coming in through the socket and hand it over to the client using the socket.accept() method. This thread is an instance of a class named ThreadHandler that implements the Runnable interface, and handles requests from clients connected to sockets on the proxy server.

I implemented by calling an instance of the ThreadHanlder class instead of implementing the handle() method. This allows for more efficient code management.



```java
/** Create the ProxyCache object and the socket */
public static void init(int p) {
    port = p;
    try {
        socket = new ServerSocket(port);
    } catch (IOException e) {
        System.out.println("Error creating socket: " + e);
        System.exit( status: -1);
    }
}
```

public static void init(**int** p) from **ProxyCache.java**

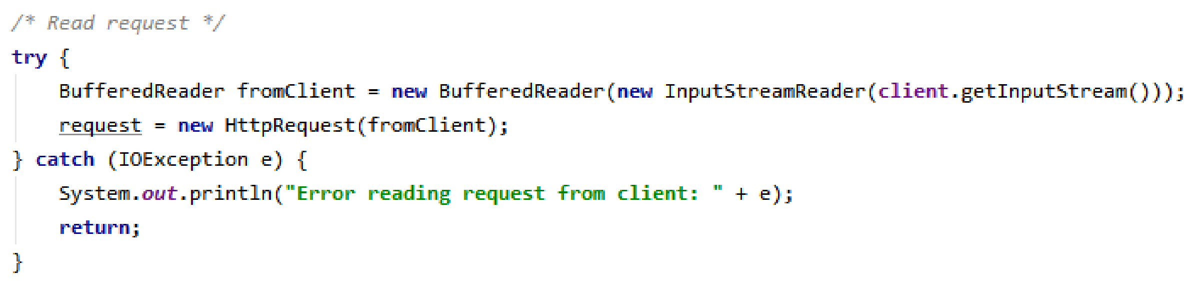


```java
while (true) {
    try {
        client = socket.accept();
        Thread thread = new Thread(new ThreadHandler(client));
        thread.run();

    } catch (IOException e) {
        System.out.println("Error reading request from client: " + e);
        /* Definitely cannot continue processing this request,
         * so skip to next iteration of while loop. */
        continue;
    }
}
```

while loop which calls an instance of class **ThreadHandler** from **ProxyCache.java**

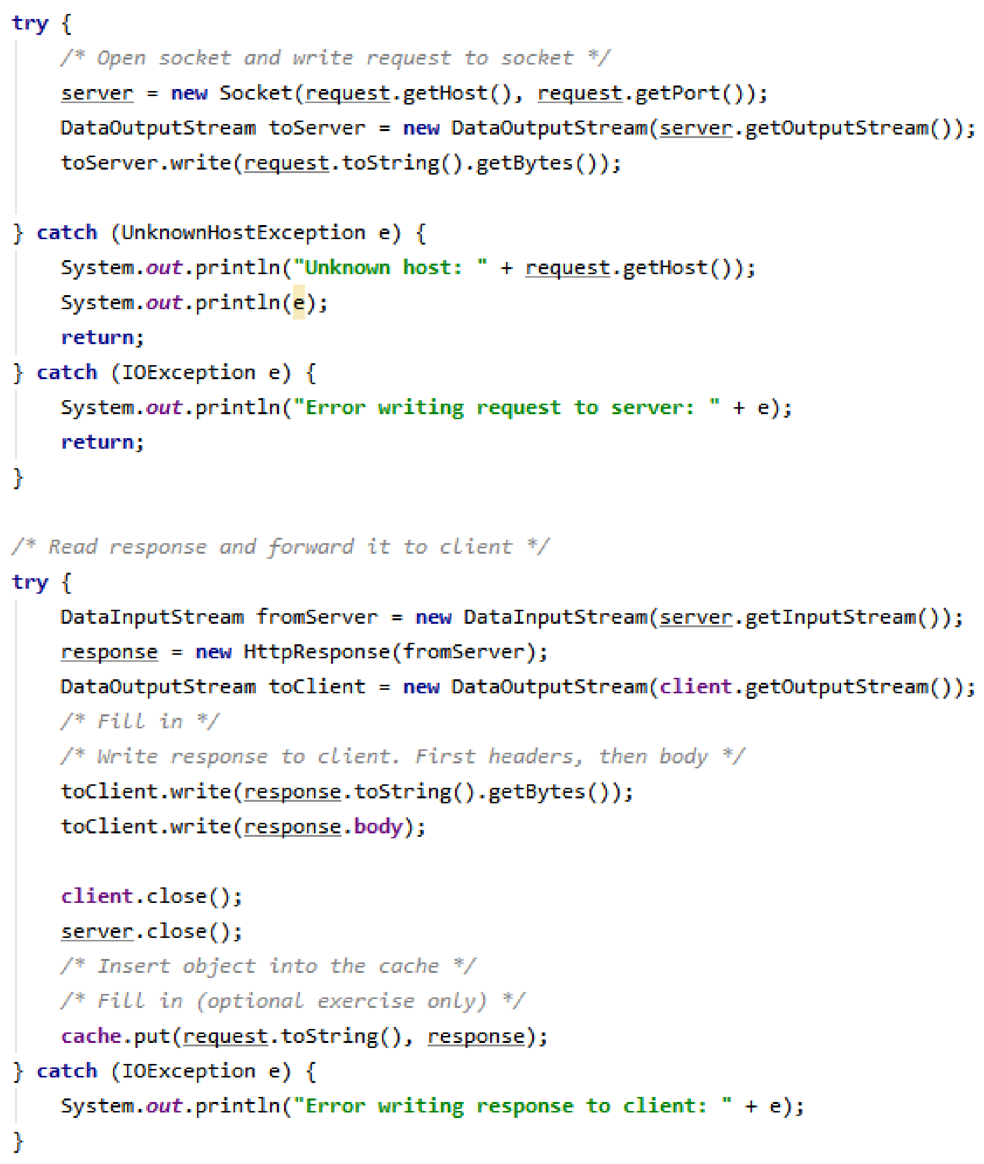
2.2. ThreadHandler.java

The ThreadHandler class replaces the Handle() method of the existing example code. In order to manage various exception handling more easily, new class file is created. Also, as mentioned earlier, the Runnable interface is implemented so that execution is possible by simply calling the run() method from the main method. First, the constructor requires a Socket as a parameter. The input socket is stored in the client which is a member variable. When the run() method is executed, first create a BufferedReader instance to read the Http Request information from the client, and then create an HttpRequest object using the instance. After that, it sends the request to the server, receives the response of the request for the server (or finds ones already cached) and sends it back to the client. The class used for inputting and outputting data in this process is DataOutputStream, because it is useful because it can handle data in byte unit and string type data at once.



```
/* Read request */
try {
    BufferedReader fromClient = new BufferedReader(new InputStreamReader(client.getInputStream()));
    request = new HttpRequest(fromClient);
} catch (IOException e) {
    System.out.println("Error reading request from client: " + e);
    return;
}
```

try-catch statement which is used to read request information from the client from **ThreadHandler.java**

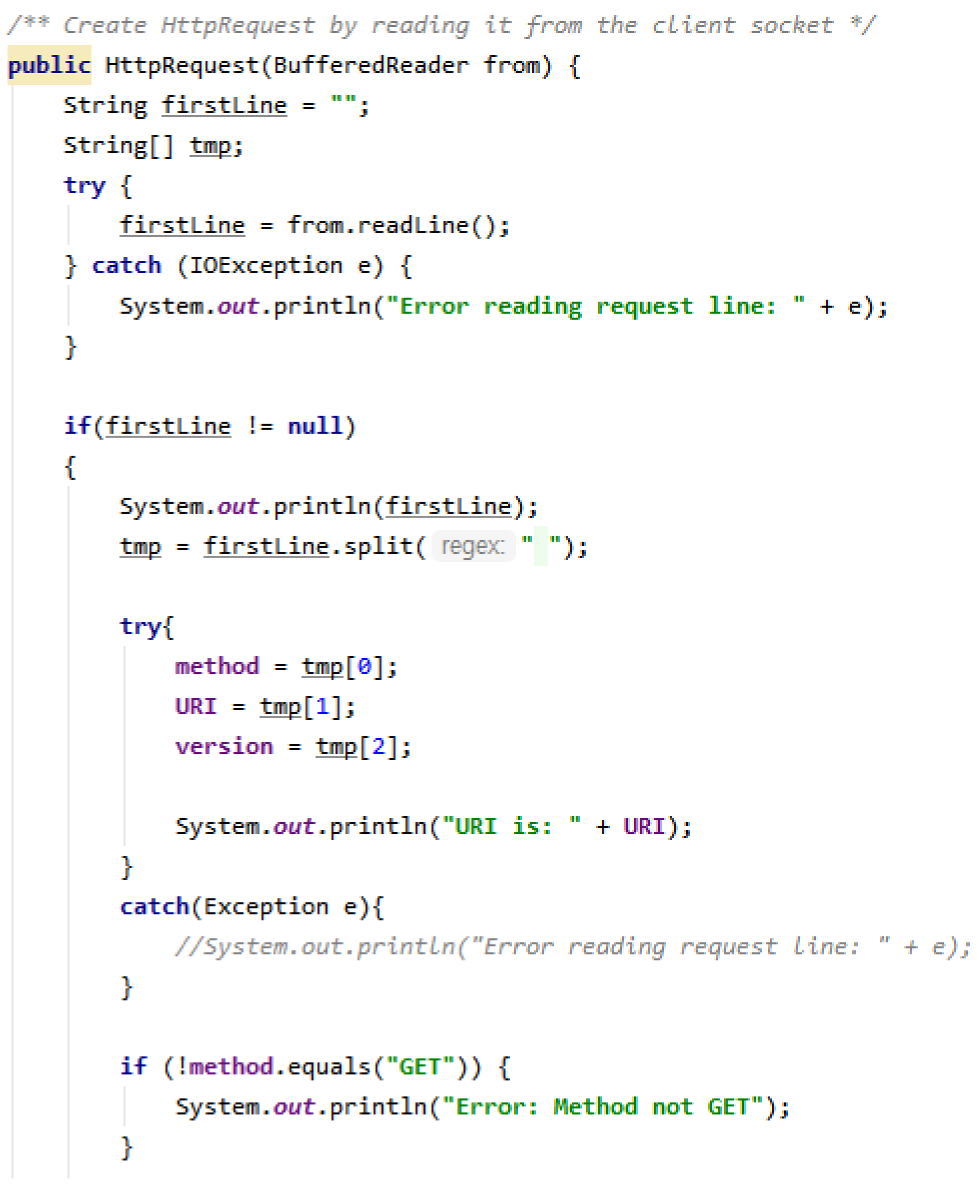


```
try {
    /* Open socket and write request to socket */
    server = new Socket(request.getHost(), request.getPort());
    DataOutputStream toServer = new DataOutputStream(server.getOutputStream());
    toServer.write(request.toString().getBytes());

} catch (UnknownHostException e) {
    System.out.println("Unknown host: " + request.getHost());
    System.out.println(e);
    return;
} catch (IOException e) {
    System.out.println("Error writing request to server: " + e);
    return;
}

/* Read response and forward it to client */
try {
    DataInputStream fromServer = new DataInputStream(server.getInputStream());
    response = new HttpResponse(fromServer);
    DataOutputStream toClient = new DataOutputStream(client.getOutputStream());
    /* Fill in */
    /* Write response to client. First headers, then body */
    toClient.write(response.toString().getBytes());
    toClient.write(response.body);

    client.close();
    server.close();
    /* Insert object into the cache */
    /* Fill in (optional exercise only) */
    cache.put(request.toString(), response);
} catch (IOException e) {
    System.out.println("Error writing response to client: " + e);
}
```

try-catch statements which are used to send request to the server and get response for the request in order to send it back to the client from **ThreadHandler.java**

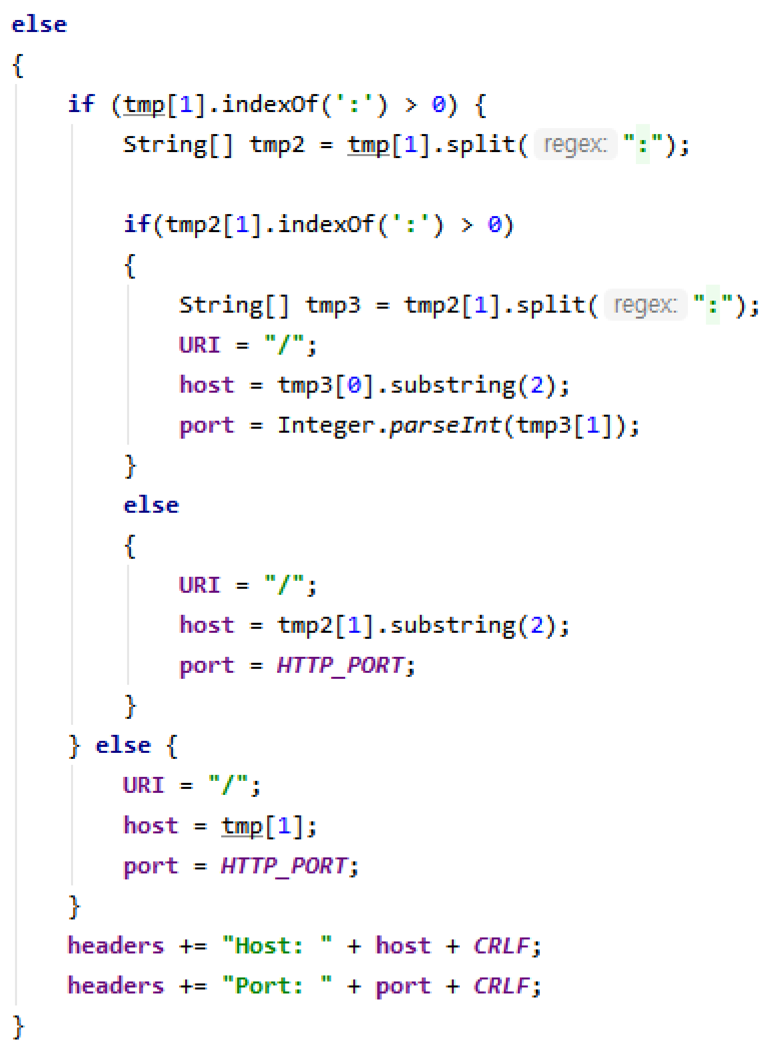
2.3. HttpRequest.java

The HttpRequest class consists of code that is very similar to the implementation of the existing Http Client. Basically, the code was composed by filling in the blanks using the skeleton code given in the assignment manual. Through this, the Http Request message received by using the buffered reader of the socket connected to the client can be analyzed and converted into a String. However, other requests such as POST and PUT are not implemented except GET request. After that, I found the additional parts to be managed by try catch statement and added the code that handles exception handling as follows.



```java
/** Create HttpRequest by reading it from the client socket */
public HttpRequest(BufferedReader from) {
    String firstLine = "";
    String[] tmp;
    try {
        firstLine = from.readLine();
    } catch (IOException e) {
        System.out.println("Error reading request line: " + e);
    }

    if(firstLine != null)
    {
        System.out.println(firstLine);
        tmp = firstLine.split( regex: " ");

        try{
            method = tmp[0];
            URI = tmp[1];
            version = tmp[2];

            System.out.println("URI is: " + URI);
        }
        catch(Exception e){
            //System.out.println("Error reading request line: " + e);
        }

        if (!method.equals("GET")) {
            System.out.println("Error: Method not GET");
        }
```

The first part of the constructor of the **HttpRequest** class which added some exception handling from **HttpRequest.java.** This code allows the class instance to parse the Http request message and check whether its method is GET or not.



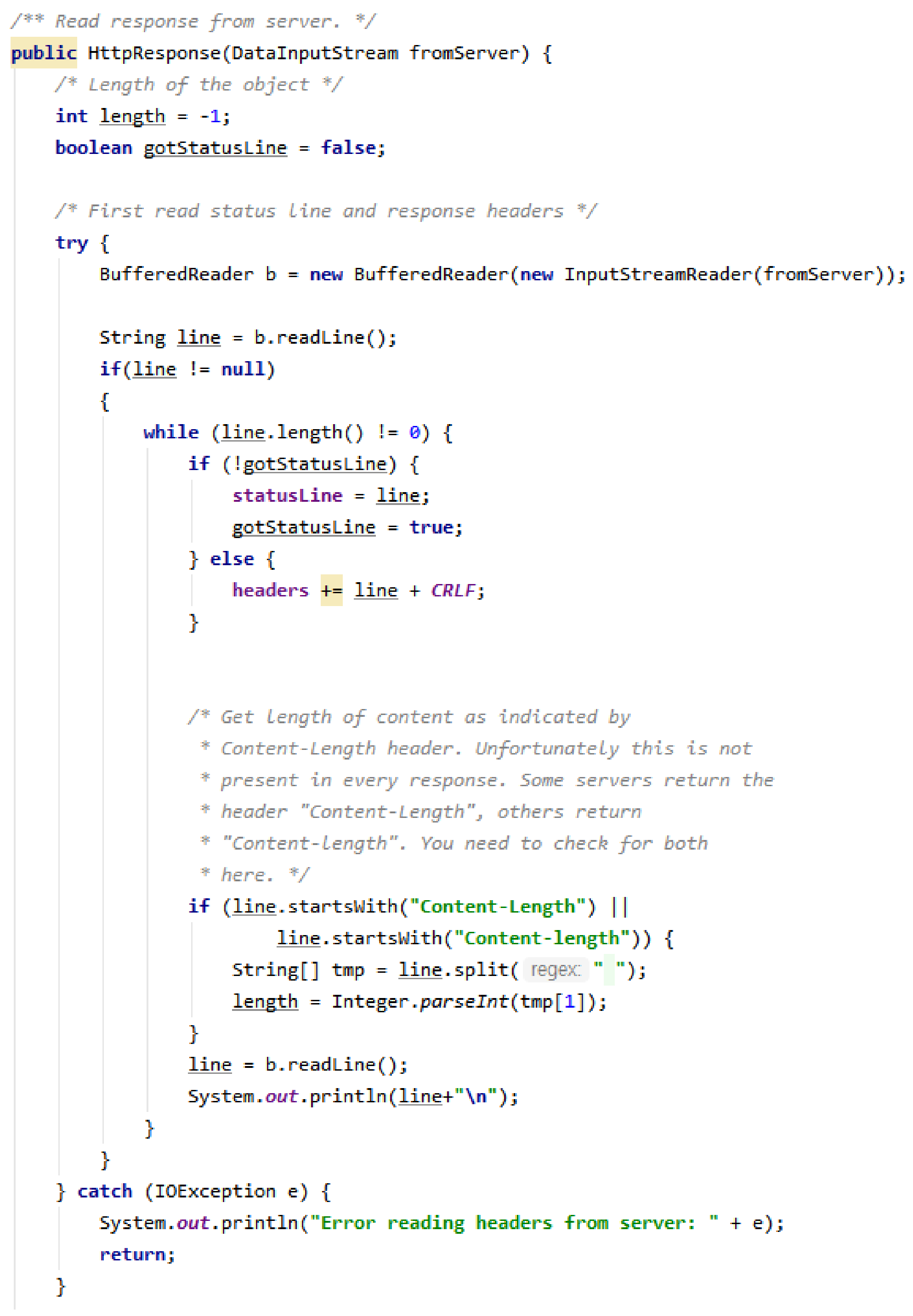
```
else
{
    if (tmp[1].indexOf(':') > 0) {
        String[] tmp2 = tmp[1].split( regex: ":");

        if(tmp2[1].indexOf(':') > 0)
        {
            String[] tmp3 = tmp2[1].split( regex: ":");
            URI = "/";
            host = tmp3[0].substring(2);
            port = Integer.parseInt(tmp3[1]);
        }
        else
        {
            URI = "/";
            host = tmp2[1].substring(2);
            port = HTTP_PORT;
        }
    } else {
        URI = "/";
        host = tmp[1];
        port = HTTP_PORT;
    }
    headers += "Host: " + host + CRLF;
    headers += "Port: " + port + CRLF;
}
```

This is newly added part for **HttpRequest.java** that can allow **HttpRequest** class's instance that if first line is **null** then it finds the requested page from the **root directory(/)**.

2.4. HttpResponse.java

HttpResponse.java is also designed to fill in the blanks based on the skeleton code in the assignment manual. It starts by connecting to the host that can be known from the analyzed Http request and obtaining DataInputStream that outputs the data transmitted by the server and using it as a parameter of the constructor. First, we use the BufferedReader class to read the data sent by the server. After that, the

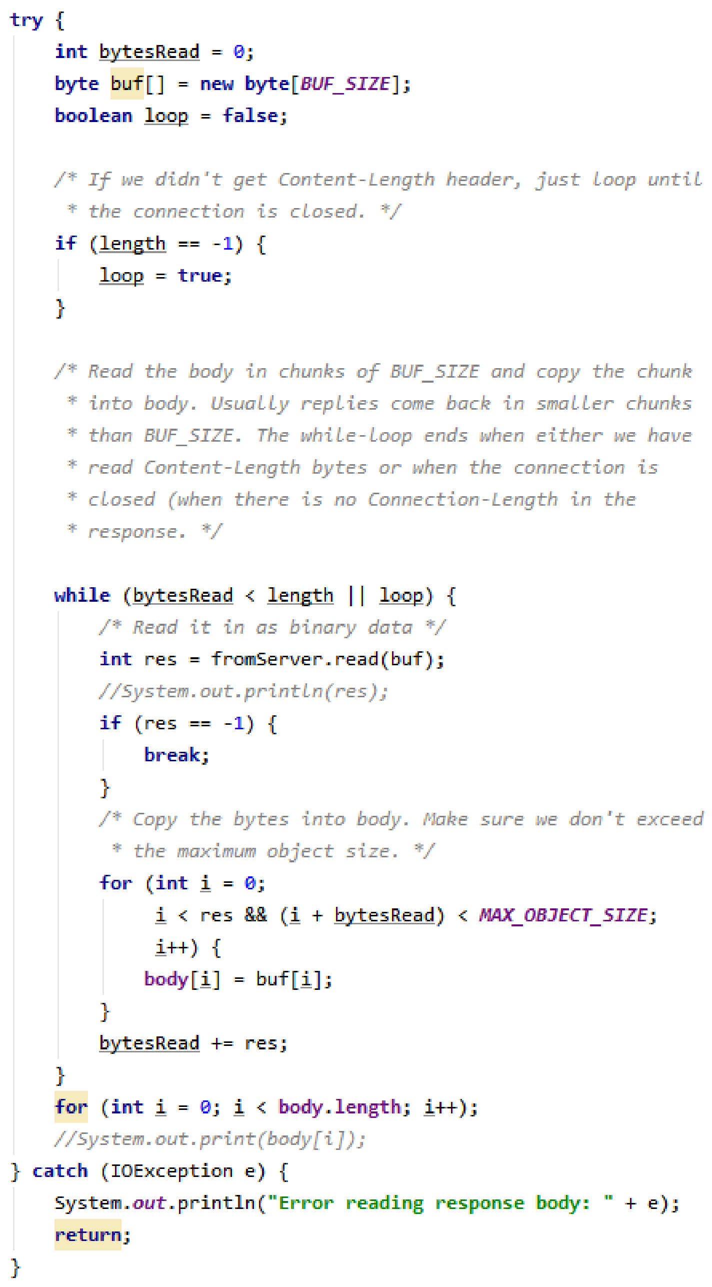
Status Line is obtained through the BufferedReader.readLine() method, and the headers are analyzed to obtain the length of the content and store the header lines if the Content-Length header exists.



```java
/** Read response from server. */
public HttpResponse(DataInputStream fromServer) {
    /* Length of the object */
    int length = -1;
    boolean gotStatusLine = false;

    /* First read status line and response headers */
    try {
        BufferedReader b = new BufferedReader(new InputStreamReader(fromServer));

        String line = b.readLine();
        if(line != null)
        {
            while (line.length() != 0) {
                if (!gotStatusLine) {
                    statusLine = line;
                    gotStatusLine = true;
                } else {
                    headers += line + CRLF;
                }


                /* Get length of content as indicated by
                 * Content-Length header. Unfortunately this is not
                 * present in every response. Some servers return the
                 * header "Content-Length", others return
                 * "Content-length". You need to check for both
                 * here. */
                if (line.startsWith("Content-Length") ||
                        line.startsWith("Content-length")) {
                    String[] tmp = line.split( regex: " ");
                    length = Integer.parseInt(tmp[1]);
                }
                line = b.readLine();
                System.out.println(line+"\n");
            }
        }
    } catch (IOException e) {
        System.out.println("Error reading headers from server: " + e);
        return;
    }
```

The first part of the constructor of **HttpResponse** class. This code allows the instance of **HttpResponse** in order to read the status line and header lines from the Http response from server and maintain the **Content-length** if it exists.

After checking the Content-Length, the body contents are read. Here, a code that reads only the data indicated by the Content-Length header has been added. If there is no Content-Length (such as 404 Not found), the byte data is read repeatedly until no new data can be read. The data read through buf is stored in a member variable called body.



```
try {
    int bytesRead = 0;
    byte buf[] = new byte[BUF_SIZE];
    boolean loop = false;

    /* If we didn't get Content-Length header, just loop until
     * the connection is closed. */
    if (length == -1) {
        loop = true;
    }

    /* Read the body in chunks of BUF_SIZE and copy the chunk
     * into body. Usually replies come back in smaller chunks
     * than BUF_SIZE. The while-loop ends when either we have
     * read Content-Length bytes or when the connection is
     * closed (when there is no Connection-Length in the
     * response. */

    while (bytesRead < length || loop) {
        /* Read it in as binary data */
        int res = fromServer.read(buf);
        //System.out.println(res);
        if (res == -1) {
            break;
        }
        /* Copy the bytes into body. Make sure we don't exceed
         * the maximum object size. */
        for (int i = 0;
             i < res && (i + bytesRead) < MAX_OBJECT_SIZE;
             i++) {
            body[i] = buf[i];
        }
        bytesRead += res;
    }
    for (int i = 0; i < body.length; i++);
    //System.out.print(body[i]);
} catch (IOException e) {
    System.out.println("Error reading response body: " + e);
    return;
}
```

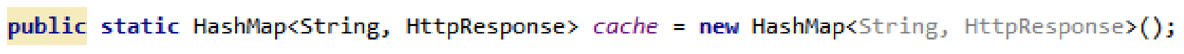
The second part of the constructor of **HttpResponse** class. This code check whether the **Content-Length** was stored or not and read the body in chunks of **BUF_SIZE** and copy the chunk into the body.

3. Implemented Optional Objectives

Two of the optional objectives specified in the task description were implemented: **Better exception handling** and **caching**.

First, Better exception handling checks the value of Content-Length and reads the body contents as much as that value, as described in the above HttpResponse.java code description. If the Content-Length header line does not exist, new data is no longer available. This is done by only reading until it can't read it. This ensures that when the server sends a status code such as 404 Not found, it no longer has any data to read and can safely stop reading the body.

Caching could be implemented by adding the following code to ProxyCache.java and ThreadHandler.java. First, add a HashMap member variable that stores <String, HttpResponse> pairs in the ProxyCache class as shown below. After processing the client's request sent to the socket from the ThreadHandler, the request is not always sent to the server. Instead, the code checks if there is a cached response and if so, configures the code to send the response directly to the client. If there is no cached response, send the request to the server to receive the response and save the received response in the caching hash map as shown below. Based on this code, we were able to implement Better exception handling and caching.



```java
public static HashMap<String, HttpResponse> cache = new HashMap<String, HttpResponse>();
```

The **static HashMap** member **cache** which is included in class **ProxyCache** in order to cache response messages for the previous Http requests.

```java
public class ThreadHandler implements Runnable
{
    HashMap<String, HttpResponse> cache = ProxyCache.cache;
```

The **HashMap** member **cache** which is included in class **ThreadHandler** in order to cache response messages for the previous Http requests.

```java
if(cache.get(request.toString()) != null)
{
    try{
        response = cache.get(request.toString());

        DataOutputStream toClient = new DataOutputStream(client.getOutputStream());
        /* Fill in */
        /* Write response to client. First headers, then body */
        toClient.write(response.toString().getBytes());
        toClient.write(response.body);

        client.close();
    } catch (IOException e) {
        System.out.println("Error writing response to client: " + e);
    }
}
```

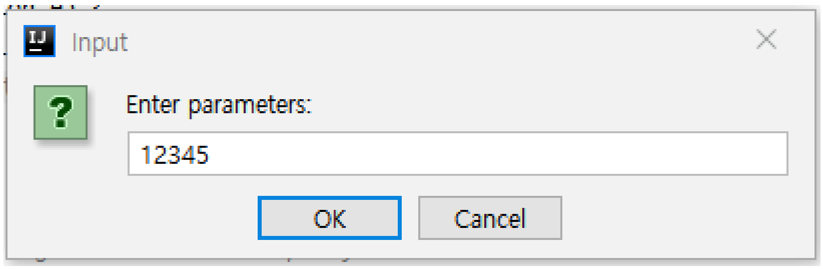
The code included in **ThreadHandler** in order to check whether the current Http request has been send through this proxy server that it maintains previous response for the same request. If it exists, then send it right away to the client, not requesting to the host server for response.

```java
/* Insert object into the cache */
/* Fill in (optional exercise only) */
cache.put(request.toString(), response);
```

This is the code in **ThreadHandler** in order to cache the response from the server that it could be used for later Http responses.

4. Results

Since the Web proxy server implemented in this project is designed to receive port number using command argument, I was able to execute as below screens by utilizing the function of InteliJ IDEA which I am currently using.

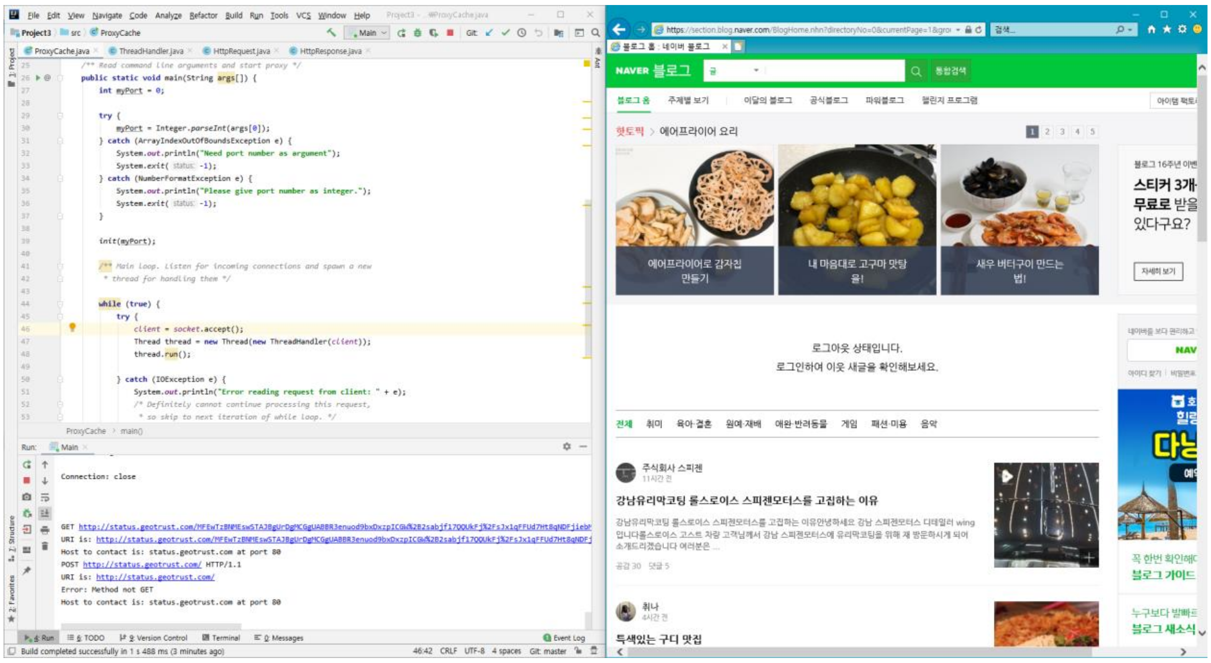


Input

Enter parameters:

12345

If I input the same port number as the proxy setting of the internet browser, I can see the normal web page and the web page caching process is displayed on the proxy server program console as shown below.



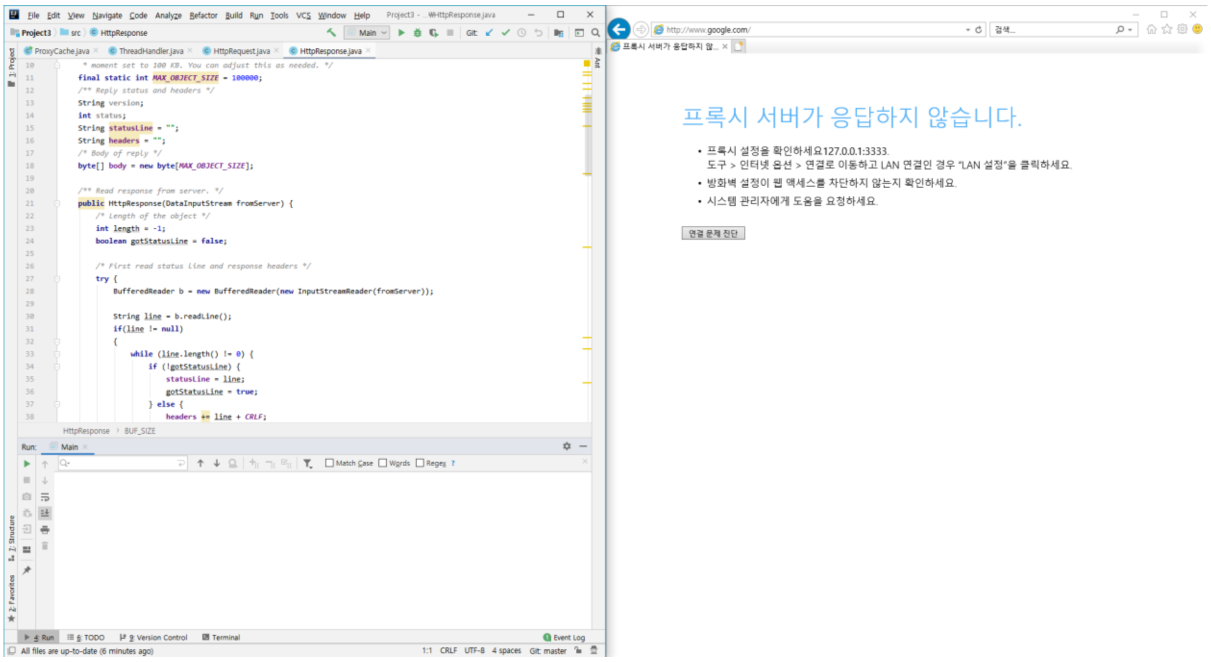
```java
    /** Read command line arguments and start proxy */
    public static void main(String args[]) {
        int myPort = 0;

        try {
            myPort = Integer.parseInt(args[0]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Need port number as argument");
            System.exit( status: -1);
        } catch (NumberFormatException e) {
            System.out.println("Please give port number as integer.");
            System.exit( status: -1);
        }

        init(myPort);

        /** Main loop. Listen for incoming connections and spawn a new
         * thread for handling them */

        while (true) {
            try {
                client = socket.accept();
                Thread thread = new Thread(new ThreadHandler(client));
                thread.run();

            } catch (IOException e) {
                System.out.println("Error reading request from client: " + e);
                /* Definitely cannot continue processing this request,
                 * so skip to next iteration of while loop. */
```

```
Connection: close



GET http://status.geotrust.com/MFEwTzBNMEswSTAJBgUrDgMCGgUABBR3enuod9bxDxzpICGW%2B2sabjf17QQUkFj%2FsJx1qFFUd7Ht8qNDFjieb
URI is: http://status.geotrust.com/MFEwTzBNMEswSTAJBgUrDgMCGgUABBR3enuod9bxDxzpICGW%2B2sabjf17QQUkFj%2FsJx1qFFUd7Ht8qNDFj
Host to contact is: status.geotrust.com at port 80
POST http://status.geotrust.com/ HTTP/1.1
URI is: http://status.geotrust.com/
Error: Method not GET
Host to contact is: status.geotrust.com at port 80
```

Build completed successfully in 1 s 488 ms (3 minutes ago)

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Project3 src ProxyCache
ProxyCache.java ThreadHandler.java HttpRequest.java HttpResponse.java
ProxyCache > main()
Run: Main
4: Run 6: TODO 9: Version Control Terminal 0: Messages Event Log
46:42 CRLF UTF-8 4 spaces Git: master
NAVER 블로그
로그아웃 상태입니다.
로그인하여 이웃 새글을 확인해보세요.

However, if I enter the wrong port number or the proxy server is not working, it is not available as shown below.



```java
     * moment set to 100 KB. You can adjust this as needed. */
    final static int MAX_OBJECT_SIZE = 100000;
    /** Reply status and headers */
    String version;
    int status;
    String statusLine = "";
    String headers = "";
    /* Body of reply */
    byte[] body = new byte[MAX_OBJECT_SIZE];

    /** Read response from server. */
    public HttpResponse(DataInputStream fromServer) {
        /* Length of the object */
        int length = -1;
        boolean gotStatusLine = false;

        /* First read status line and response headers */
        try {
            BufferedReader b = new BufferedReader(new InputStreamReader(fromServer));

            String line = b.readLine();
            if(line != null)
            {
                while (line.length() != 0) {
                    if (!gotStatusLine) {
                        statusLine = line;
                        gotStatusLine = true;
                    } else {
                        headers += line + CRLF;
```

http://www.google.com/

프록시 서버가 응답하지 않습니다.

- 프록시 설정을 확인하세요127.0.0.1:3333.
  도구 > 인터넷 옵션 > 연결로 이동하고 LAN 연결인 경우 "LAN 설정"을 클릭하세요.
- 방화벽 설정이 웹 액세스를 차단하지 않는지 확인하세요.
- 시스템 관리자에게 도움을 요청하세요.

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Project3 › src › HttpResponse
ProxyCache.java
ThreadHandler.java
HttpRequest.java
HttpResponse.java
HttpResponse › BUF_SIZE
Run: Main
Match Case
Words
Regex
Run
6: TODO
9: Version Control
Terminal
Event Log
All files are up-to-date (6 minutes ago)
1:1 CRLF UTF-8 4 spaces Git: master
검색...
프록시 서버가 응답하지 않...
연결 문제 진단