Flow-Controlled FTP Program Manual and Installation Procedure

Fall 2020

이름: 완

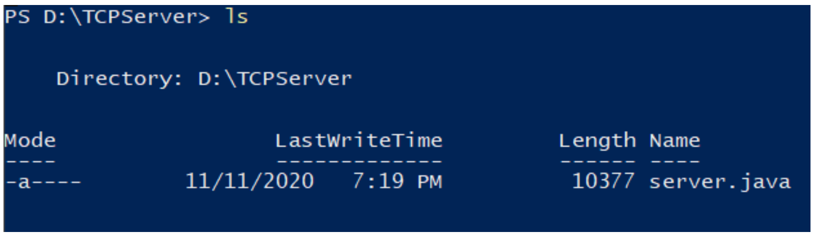
학생번호:2019007901

Environment: Windows 10

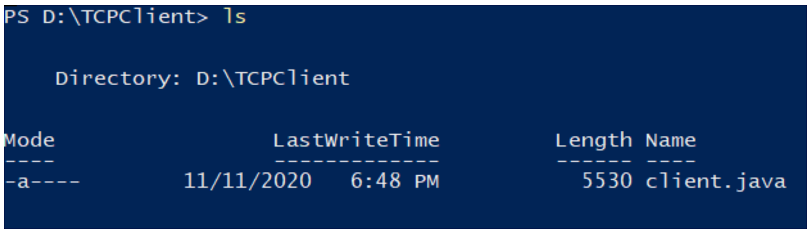
Requirement: Java v1.8.0 or later

- ## Compiling Procedure

Begin by saving the source files for both client application and server application (server.java and client.java respectively) to the desired location. That location will be the starting path for server application and default path for client application.



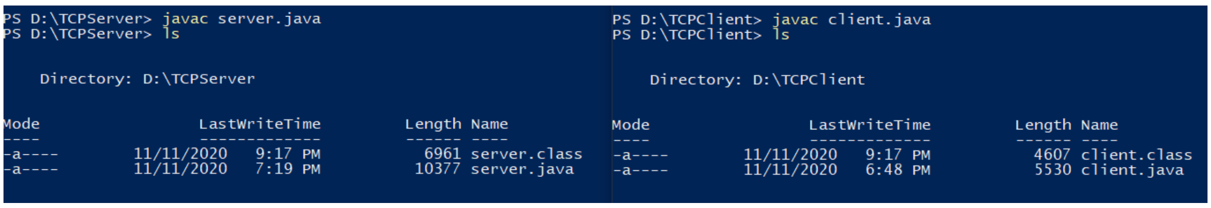
```
PS D:\TCPServer> ls


    Directory: D:\TCPServer


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        11/11/2020   7:19 PM          10377 server.java
```



```
PS D:\TCPClient> ls


    Directory: D:\TCPClient


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        11/11/2020   6:48 PM           5530 client.java
```

Example: Server.java in "D:\TCPServer" and client.java in "D:\TCPClient".

Start compiling the both java files in their respective directories by using the command "*javac <java filename>*" in the command prompt. If the compiling stage is succeeded, the CLASS file for both of java file will be created in the same directory.



```
PS D:\TCPServer> javac server.java
PS D:\TCPServer> ls


    Directory: D:\TCPServer


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        11/11/2020   9:17 PM           6961 server.class
-a----        11/11/2020   7:19 PM          10377 server.java
```

```
PS D:\TCPClient> javac client.java
PS D:\TCPClient> ls


    Directory: D:\TCPClient


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        11/11/2020   9:17 PM           4607 client.class
-a----        11/11/2020   6:48 PM           5530 client.java
```

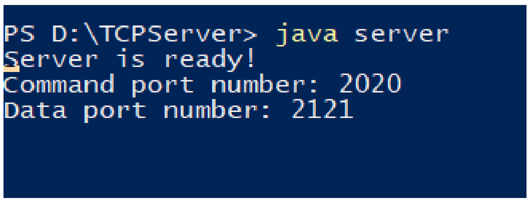
.CLASS files are created after compiling using javac.

After successfully compiling the java files. You can run both programs using "*java <filename>*" command in the command prompt. You will need to **run the server application first** before executing client application. Starting client application first will cause error as there is no server available at the moment.

- **Running the Program**

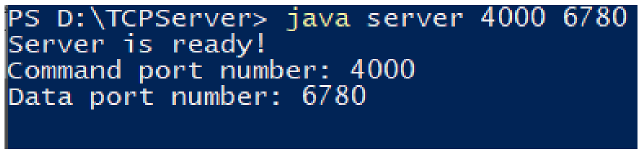
1. <u>Server Program</u>

To run the server application, simply type "*java server*" in the command prompt that contains the compiled *server.java* file. After running the server program, the program will tell the user that the server is on and displays the command port number and data port number.



```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121
```

By default, the server command port number is set to "2020" and data port number is "2121". You can provide your own designated port numbers by giving the command line arguments. Type "*java server <command port number> <data port number>*" when executing the server program.



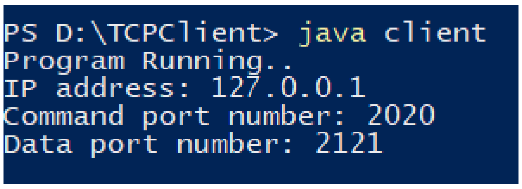
```
PS D:\TCPServer> java server 4000 6780
Server is ready!
Command port number: 4000
Data port number: 6780
```

Executing with command line argument to change the default port number

Now your server program is ready to receive any connection from the client!

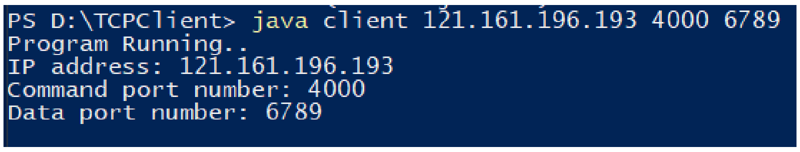
2. <u>Client Program</u>

You can run the client program after executing the server program. The method for running the client program is similar as before, by typing "*java client*" into the command prompt of client directory. If the execution successful, the program will display the IP Address of FTP server host, command line port number, and data port number onto the screen.



```
PS D:\TCPClient> java client
Program Running..
IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121
```

By default, the server FTP host will be designated to "127.0.0.1" which is the localhost, command port "2020" and data port "2121". To change these default arguments, type "*java client <server host> <command port number> <data port number>*" before executing the program.



```
PS D:\TCPClient> java client 121.161.196.193 4000 6789
Program Running..
IP address: 121.161.196.193
Command port number: 4000
Data port number: 6789
```

Executing with command line argument to change the default port number

The client's port numbers **must be similar to server program's port numbers** for the program to run successfully. Make sure the server IP address is valid to the server program too.

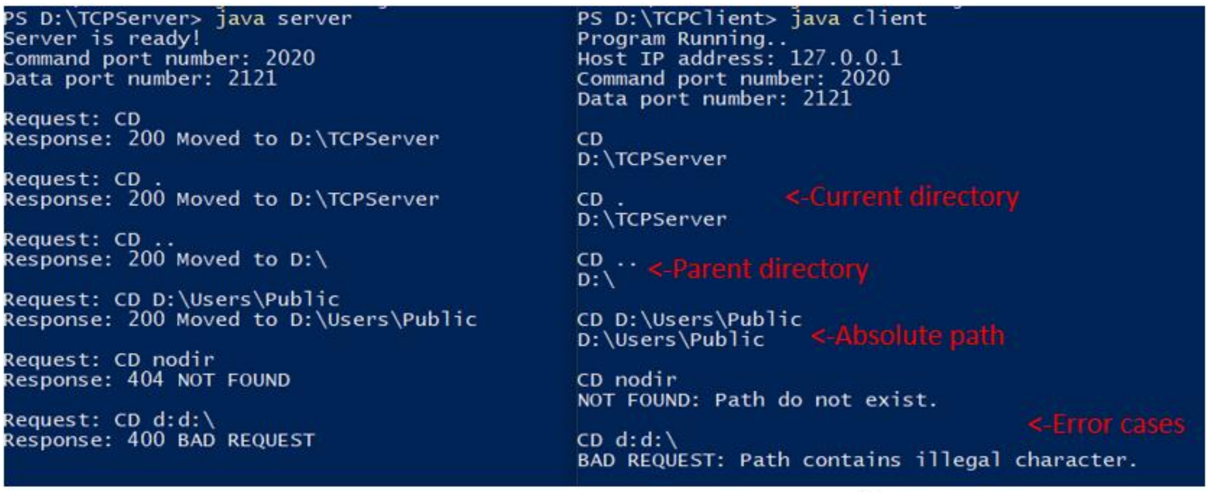
- **Command Lists for Client Program and Design for every Command**

1. CD <pathname>

This command is used to change the current directory of the server. In response, the server with the new directory path given by the client side. If successful, the server must return the full absolute path of server's current directory and print it on the client program screen.

From client side, the program receives a string of input by the user such as "*CD <pathname>*" and that string will be sent to server program. Then, the server program will parse the string and get the *<pathname>* parameter from the client. After that, a Path type variable *currentPath* is initialized and is used to represent the current path of the server throughout the whole execution process. *currentPath* variable will be changed accordingly to the parameter of <*pathname*> from client program. The <*pathname*> parameter is then checked by the server if it exist, a relative path, or an absolute path using several methods from PATH class such as *.isAbsolute()*, *.getParent()* and *.exist().* If <*pathname*> is "." or blank, server will not change its current directory. If <*pathname*> is "..", server will change the *currentPath* to the parent directory of the current server directory.

After changing the *currentPath* , the server will return the response message and status code to the client program and display the server current directory on the client program. If given <*pathname*> is either not exist, invalid path, or not a directory, server program will display an error code and the error statement is displayed onto client screen.



```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121

Request: CD
Response: 200 Moved to D:\TCPServer

Request: CD .
Response: 200 Moved to D:\TCPServer

Request: CD ..
Response: 200 Moved to D:\

Request: CD D:\Users\Public
Response: 200 Moved to D:\Users\Public

Request: CD nodir
Response: 404 NOT FOUND

Request: CD d:d:\
Response: 400 BAD REQUEST
```

```
PS D:\TCPClient> java client
Program Running..
Host IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121

CD
D:\TCPServer

CD .
D:\TCPServer

CD ..
D:\

CD D:\Users\Public
D:\Users\Public

CD nodir
NOT FOUND: Path do not exist.

CD d:d:\
BAD REQUEST: Path contains illegal character.
```

Server Client

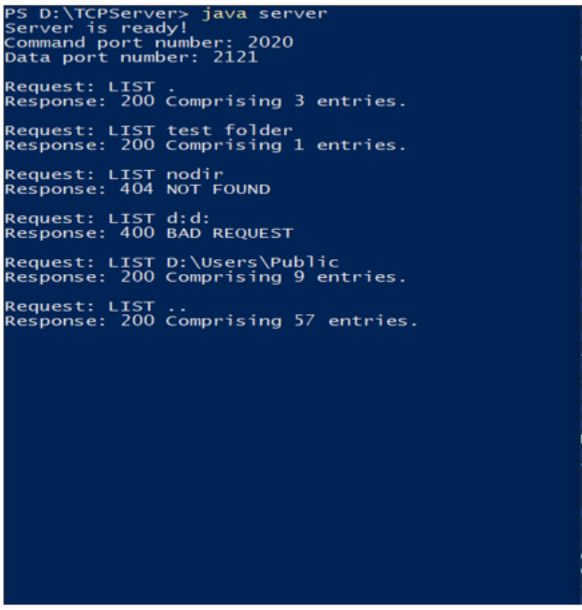
This program supports any type of pathname either if it is an absolute path or relative path. Using the method *Paths.get()*, we can parse the string <*pathname*> to the Path type variable. If there is an error occurred while parsing the <*pathname*>, the server will throw an error handling block returns BAD REQUEST status code to the client.

2. LIST <pathname>

The LIST command is used to list out all the files that are in the current directory of the server. The server will return a list of names of files with the size of it respectively to the client program. If the file is a directory, it size will be set to ”-“. Meanwhile, the size of files are represented in bytes and the client should receive a response with format of *<filename>,<size in bytes>\n*.

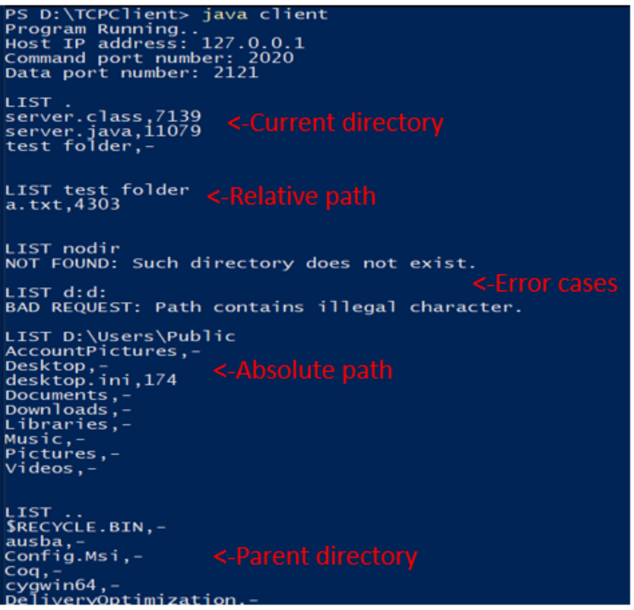
Similar to the previous command, the client will receive the command line from user and send it to the server. The server will parse that request string and jump to the correct command function which is LIST. Then, the <*pathname*> is parsed into Path type variable and server will check either if the path is a directory and is exist. Reminder that this LIST command will **NOT change** the current directory of the server. It is only used for checking the list of files that available in given pathname. Similar as before, the <*pathname*> parameter can be either absolute, relative, “.” for current directory or “..” for parent directory. To get the list of the filenames, the server uses *.toFile().list()* method to store all the filenames into an array. Server also creates another array to store the file sizes using *.length()* method from FILE class.

Finally, the server will return the contents of both filename and filesize arrays to the client in one long string while *<filename><size>* pair are separated using “|” character. When the client receives this response string. It will use the *.split()* method to split the response string by delimiter “|”. Finally, the client will display all the filenames and file sizes separated by newline character.



```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121

Request: LIST .
Response: 200 Comprising 3 entries.

Request: LIST test folder
Response: 200 Comprising 1 entries.

Request: LIST nodir
Response: 404 NOT FOUND

Request: LIST d:d:
Response: 400 BAD REQUEST

Request: LIST D:\Users\Public
Response: 200 Comprising 9 entries.

Request: LIST ..
Response: 200 Comprising 57 entries.
```

Server



```
PS D:\TCPClient> java client
Program Running..
Host IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121

LIST .
server.class,7139
server.java,11079
test folder,-

LIST test folder
a.txt,4303

LIST nodir
NOT FOUND: Such directory does not exist.

LIST d:d:
BAD REQUEST: Path contains illegal character.

LIST D:\Users\Public
AccountPictures,-
Desktop,-
desktop.ini,174
Documents,-
Downloads,-
Libraries,-
Music,-
Pictures,-
Videos,-

LIST ..
$RECYCLE.BIN,-
ausba,-
Config.Msi,-
Coq,-
cygwin64,-
DeliveryOptimization,-
```

Client

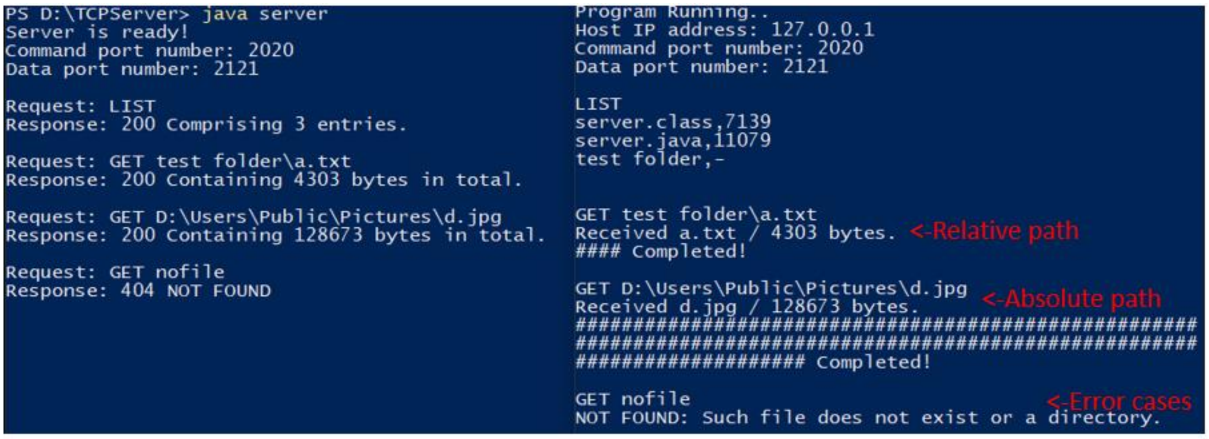
The server will return an error code if the pathname given is either invalid, not exist, or not a directory. If the client does not receive response code “200” from the server, error message is displayed to the client screen.

3. GET <filename>

This command is used to receive any specified file from the current server directory. If successful, the requested file is then saved to the current client directory. The argument <*filename*> can also be an absolute pathname or relative pathname. After requesting, the server then checks if the requested file exist in the current server directory. If exist, the server will return response code "200" along with the filename and size to the client. Then, the transfer progress bar will be displayed onto the client program. The file is transferred separately in several chunks though the data port.

First, the client program will send the command request that contains the path or the filename of the desired file. In the server program, the command request is parsed and the server will check the validity of the command argument. If the filename is valid and exist in the current server directory, the server will write the file using *FileInputStream* class. After that, a buffer ***b*** with size of 1000 is initialized and for every 1000 bytes, the contents of the file is read into the buffer and the server will send the buffer to the client through *write()* method from *OutputStream* class. This process is continued until the *read()* method of *FileInputStream* reaches end of file. Finally, server will close the data port after fully transferred all chunks.

Onto the client side, the data port is opened to receive the chunks of file from the server. Client will receive the file name and size from the server and read every chunks that arrives to the client using *read()* method from *InputStream* class. Then for every 1000 bytes received, the file is written to the client directory using *write()* method from *FileOutputStream* until the read method reaches the end of file. At the same time, a character "#" is printed to the client screen for every iteration of this process to indicate the transferring progress. Finally, after successfully write the full file to the client directory, the data port is closed to avoid resources leak.



```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121

Request: LIST
Response: 200 Comprising 3 entries.

Request: GET test folder\a.txt
Response: 200 Containing 4303 bytes in total.

Request: GET D:\Users\Public\Pictures\d.jpg
Response: 200 Containing 128673 bytes in total.

Request: GET nofile
Response: 404 NOT FOUND
```

```
Program Running..
Host IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121

LIST
server.class,7139
server.java,11079
test folder,-


GET test folder\a.txt
Received a.txt / 4303 bytes.
#### Completed!

GET D:\Users\Public\Pictures\d.jpg
Received d.jpg / 128673 bytes.
######################################################
######################################################
#################### Completed!

GET nofile
NOT FOUND: Such file does not exist or a directory.
```

Server Client

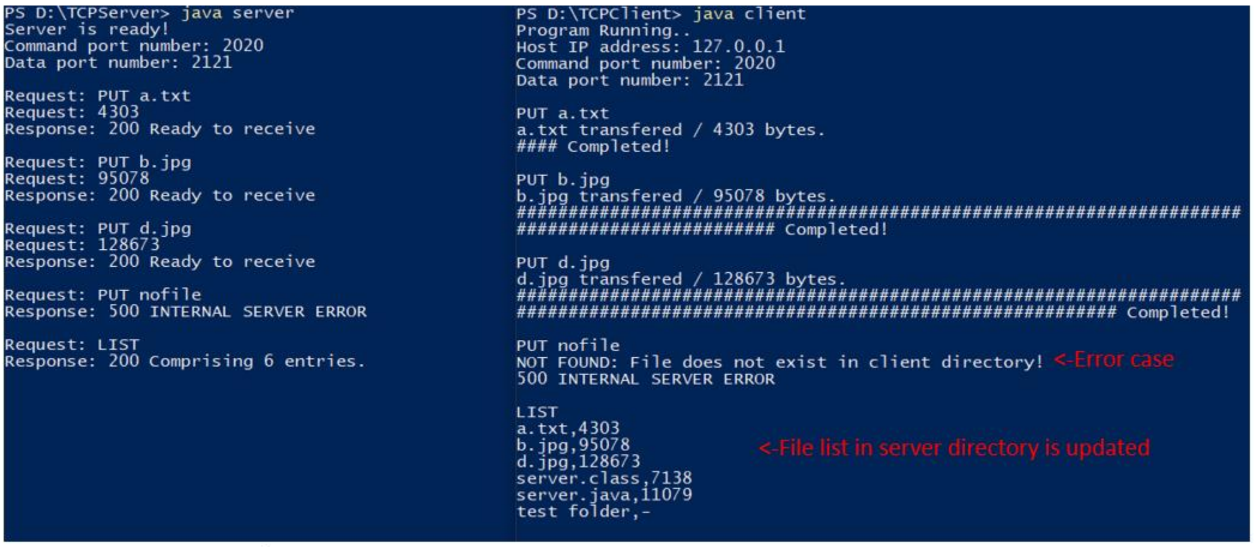
In this command, there is also an **error handling** provided in the server. This error handling is for the case when requested file from the client does not exist in current server directory. Server will return status code "404" if exception occurs and close the data port immediately. Error handling is not necessary on the client side because the client only request and write received files.

4. PUT <filename>

This command is used for the client to send a specific file to the current directory of the server. In real world situation, we call it as "uploading". Because this command only accepts filename as argument, we don't have wo worry about having to parse pathname and such. Only filename is required because this command only let the client to send files that exist **only in client directory**. Client is not able to send a file that located other than the client directory.

The way this command works is not so different from the *GET* command. It is just the other way round, in which the sender is client and the receiver is server. If we analyse the source code of both programs for this command, we can see it is really similar to *GET* command but swapped. Let's see how this command works. At the beginning, the client will open the data port for file transmission. Then, the client program will check the validity of the filename, if it exists and is not a directory. If both of the condition satisfied, client will send the file size to the server through the command port. If server responded with "200 Ready to receive", like *GET* command, a buffer ***b*** with size 1000 bytes is initialized and the file is written from client directory into the buffer for every 1000 bytes. After the buffer is full, it will get sent to the server as a chunk and every time a chunk is sent, the character "#" is displayed onto the client monitor. Finally, the *OutputStream* is closed together with the data port. If file is not found, exception occurs and the client will send "-1" as the file size and the data port is closed immediately.

From the server side, before receiving the chunk, the server will open the data port and receives the filename and the size from the client. If the server receives the file size as "-1", that means the filename is invalid and data port is closed while returning error response code to the client. Else, the server will respond with "200 Ready to receive" to the client and wait until the client sends the file. After receiving all the chunks, server will close its data port.



```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121

Request: PUT a.txt
Request: 4303
Response: 200 Ready to receive

Request: PUT b.jpg
Request: 95078
Response: 200 Ready to receive

Request: PUT d.jpg
Request: 128673
Response: 200 Ready to receive

Request: PUT nofile
Response: 500 INTERNAL SERVER ERROR

Request: LIST
Response: 200 Comprising 6 entries.
```

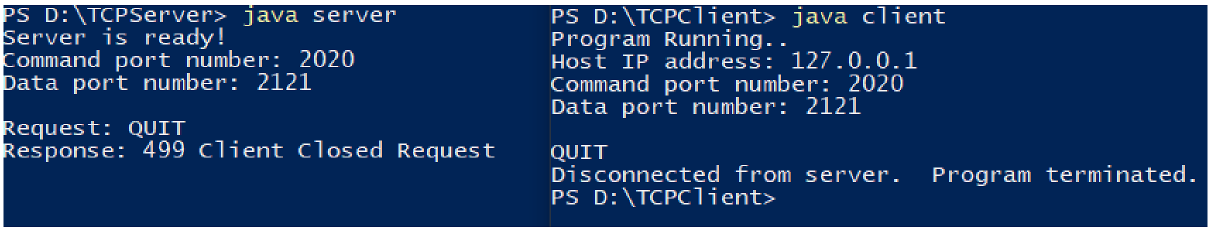
```
PS D:\TCPClient> java client
Program Running..
Host IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121

PUT a.txt
a.txt transfered / 4303 bytes.
#### Completed!

PUT b.jpg
b.jpg transfered / 95078 bytes.
#######################################################################
######################### Completed!

PUT d.jpg
d.jpg transfered / 128673 bytes.
#######################################################################
########################################################## Completed!

PUT nofile
NOT FOUND: File does not exist in client directory! <-Error case
500 INTERNAL SERVER ERROR

LIST
a.txt,4303
b.jpg,95078                 <-File list in server directory is updated
d.jpg,128673
server.class,7138
server.java,11079
test folder,-
```

Server   Client

5. QUIT

This command is used by the client to terminate the client program and disconnect from the server. When this command is entered, the client program will send the request the server to notify that the client will close the connection. The server will respond with status code. Finally, client will close the socket using the *.close()* method and stops the connection from the server before terminating the program.

Meanwhile, the server is in “always on” state (hence the “while(true)” statement) because the server program will always wait until another client program request to establish a connection.



```
PS D:\TCPServer> java server
Server is ready!
Command port number: 2020
Data port number: 2121

Request: QUIT
Response: 499 Client Closed Request
```

```
PS D:\TCPClient> java client
Program Running..
Host IP address: 127.0.0.1
Command port number: 2020
Data port number: 2121

QUIT
Disconnected from server.  Program terminated.
PS D:\TCPClient>
```

6. INVALID COMMAND

If the client enters any commands other that these 5 commands, server will return with status code “400 BAD REQUEST” and “Invalid command” will be printed to the client screen.

- **Force Terminate Program**

If you wanted to force terminate either the server or client program, simply press Ctrl + C at the screen of the command prompt and the program will be terminated. This is useful if you want to terminate the server program because as for now, the server does not provide any quit functions.