In this part, you will implement another web server – HTTPAsk. This is a web server that uses the client you did in Task 1 and 2. When you send an HTTP request to HTTPAsk, you provide the parameters for TCPClient as parameters in the HTTP request.

When HTTPAsk receives the HTTP request, it will call the method TCPClient.askServer(), and return the output as an HTTP response. This may seem confusing, but it is really very simple: instead of running TCPAsk from the command line, you will build a web server that runs TCPAsk for you, and presents the result as a web page (in an HTTP response).

What you will learn here is to:

- Read and analyse an HTTP GET request, and extract a query string from it.
- Compose and return a valid HTTP response.

Task

Your job is to implement a class called HTTPAsk. It's "main" method implements the server. It takes one argument: the port number. So if you want your server to run on port 8888, you would start it in the following way:

```
$ java HTTPAsk 8888
```

You will be provided with a file with a template for the HTTPAsk class. There is not much content in it though, but your task is to fill in the missing Java code *in this file*.

Since HTTPAsk is a TCP server, you should implement it in the same way as other servers: it should run in an infinite loop, and when one client has been served, the server should be prepared to take care of the next. Your server does not need to serve more than one client at a time, though. It does not need to be multi-threaded, in other words (that's for later).

Instructions and Tips

- Use the ServerSocketclass to create the socket for the server.
- Keep in mind that you are required to use byte streams I/O with explicit encoding/decoding. No wrapper classes!
- It is tricky to get everything right at once. Follow a sound software engineering approach and develop your code step by step. After each step, design and run one or more tests to verify that your code works.

Start with the HTTP Server

A suitable first step can be to create an HTTP server that generates a static HTTP response. In other words, when a client connects to the server, ignore the data that the client is sending. Instead, just return an HTTP response that always has the same data (like "hello world"). Then use a browser to connect to the server, and check that the response is shown correctly in the browser. In this way, you can verify that you can generate a correct HTTP response that the browser recognises.

Some things to consider when generating HTTP messages.

- Line breaks are significant in HTTP: Each field in the header (header line) is terminated with a line break. The header is separated from the body by an empty line (= two consecutive line breaks.)
- A line break in HTTP consists of two characters: carriage returned followed by line feed. This is probably not what your operating system uses to mark end of line (it probably uses only a line feed). In Java, the syntax for carriage return followed by line feed in a string is "\r\n". So make sure to write this sequence to mark the end of each line.
- A web browser understands and can present a plethora of different data formats (the browser often tells the server what formats it recognises through an "Accept" header line in the HTTP request). So what format should you use? Luckily, the default format for HTTP is plain text, "text/plain". As you can tell by the name, this format is just plain text, without any formatting. So unless your server specifies something else in its HTTP response, the web browser will assume your response is in plain text, and present it as such in the browser window. (If you want, you can format the response more nicely with HTML code, but that's not a requirement. But if you do that, you should specify that the content format is HTML – with the header line "Content-Type: text/html".)
- An HTTP response should also specify the character encoding that is used. The default is UTF-8, which is a superset of ASCII, so here you can just use the default as well. In other words, you don't need to specify this either.

Parsing the HTTP Request

What you need to work on in this task is mainly to process the HTTP request. The HTTPAsk server recognises one "resource" in the URL, called "ask". So the general format of an HTTP request for HTTPAsk is as follows:

```
http://hostname.domain/ask?<parameters>
```

Here, <parameters> represents the arguments for the TCPClient instance.

You should extract the hostname, port, and string parameters from the request, as well as the optional arguments for askServer (ie., shutdown flag, timeout, and limit). Your HTTP server only needs to recognise GET requests. In a GET request, parameters are sent in the URI component (the second component) of the first line of the request. The URI component comes from the URL that was used to navigate to the server.

So, for instance, this is the URL you would type into the navigation field of your browser to tell the server to access the "daytime" service at "time.nist.gov":

```
http://hostname.domain/ask?hostname=time.nist.gov&limit=1200&port=13
```

The browser will then open a TCP connection to "time.nist.gov" at port 13, and send the following GET request over the connection:

```
GET /ask?hostname=time.nist.gov&limit=1200&port=13 HTTP/1.1
Host: hostname.domain

```

Note that there is an empty line (two consecutive line breaks) that marks the end of the GET request header. last The "/ask" part of the URI tells the HTTPAsk server to set up a TCP connection to a remote server (specified by the "hostname" parameter) and a port ("port" parameter), with a data limit of 1200 bytes, and return the result. Think of this as a command to the server!

You need to write the code that validates the request, "picks apart" the URI component of the request and extracts the parameters and their values. This involves a certain amount of string processing. There are (at least) two ways you could do this. One way is to use the URL class, which has many different methods for processing URL strings. Another way is to take care of the string processing yourself – you will probably find that the String "split" method is very handy for this. It is up to you!

The query parameters your HTTPAsk server should recognize are shown in the following table:

Parameter	Meaning	Example
hostname	Domain name for host	hostname=whois.iis.se
port	Port number	port=43
string	Data to send to server Optional **Note**: before passing the string to askServer (as a byte array), make sure to terminate the string with a line break "\n".	string=kth.se
shutdown	Shutdown flag Optional, default is false	shutdown=true
limit	Data limit in bytes Optional, default is no limit	limit=512
timeout	Maximum with time in milliseconds Optional, default is no timeout	timeout=2000

If a parameter is optional, it means that it may or may not be present in the GET request. The non-optional arguments are always present.

Resources

For this task, you will be provided with the following:

- HTTPAsk.java – Skeleton declaration for the HTTPAsk class.

The file are available in a zip archive called "task3.zip". In this zip archive, there is a directory called "task3" with the file in it. So, when you unzip the archive, the "task3" directory will be created, and in this directory you will find the Java file. *This is important, because you are expected to submit your solutions in a zip archive with exactly the same structure!*

You can find the zip archive here .

Submission

Your submission should be a zip file named "task3.zip", containing the same files and directory structure as in the template. In other words, you should return a zip archive like the one you received with templates, with the same file names and the same directory.

You may want to add more files in addition to the files in the template, since your submission should be *self-contained*. This means that if you reuse code from your solutions to previous tasks, that code should be included in your submission. In particular, we expect many of you to reuse the TCPClient class from task 1. In that case, make sure to include it as well. Put the additional files in the directory together with your other files so that they are included in the ZIP archive.

We use automatic tools for the grading. If your submission does not follow this format, the tools will not recognise your submission, and it cannot be graded. Before you upload your submission, make sure that it follows the format requirements. You can do that by repeating what the submissions tools do, by running the following commands *at the command line*:

```
$ unzip task3.zip
$ cd task3
$ javac HTTPAsk.java
$ java HTTPAsk <port number>
```

where <port number> represents a port number (and not the string "<port number>"!).

Note that the tools will not look for, or compile separately, any additional classes you have created. Instead, your submission needs to be organised in such a way that all required files are compiled by the above command. Before you submit, test it yourself at the command line to make sure that it works.

Submit by uploading your "task3.zip" file below. You can submit as many times as you like before the due date.

Evaluation

The first part of the evaluation is to verify that your submission has the correct format. If it does not have the correct format, no further evaluations are made, and the result will be "fail".

As part of the evaluation of your submission, a number of tests are conducted to check the functionality of your submission. The following tests are included in the evaluation:

- Send a request to connect to a daytime server, and verify that the output is correct
- Send a request to send one line to an echo server, and verify that the output is correct
- Send a bad request, and verify that the server returns a proper HTTP response

There are some error scenarios here that your server needs to deal with. For instance, what happens if the host does not exist, or does not respond? What happens if the parameters in the HTTP request aren't valid (for example, the URI in the request does not start with "/ask")? What else could go wrong? Since you are implementing a proper HTTP server, you should make sure that your server returns suitable HTTP error codes for the different error situations that could occur. There are many different error codes; it is sufficient that you implement two: 400 and 404. Look them up and see what they mean!

It is a bit a of a philosophical question what should happen when the user asks HTTPAsk to connect to a service that does not exist (a non-existing host for instance). The grading tool does not consider this to be an error situation, so it is not something that should return an HTTP error code. A more appropriate response would be to return an HTTP page that reports the error that occurred when HTTPAsk tried to contact the host.