# Lab 1: Denial of Service (DoS) Attack

**Objective and Set Up:**

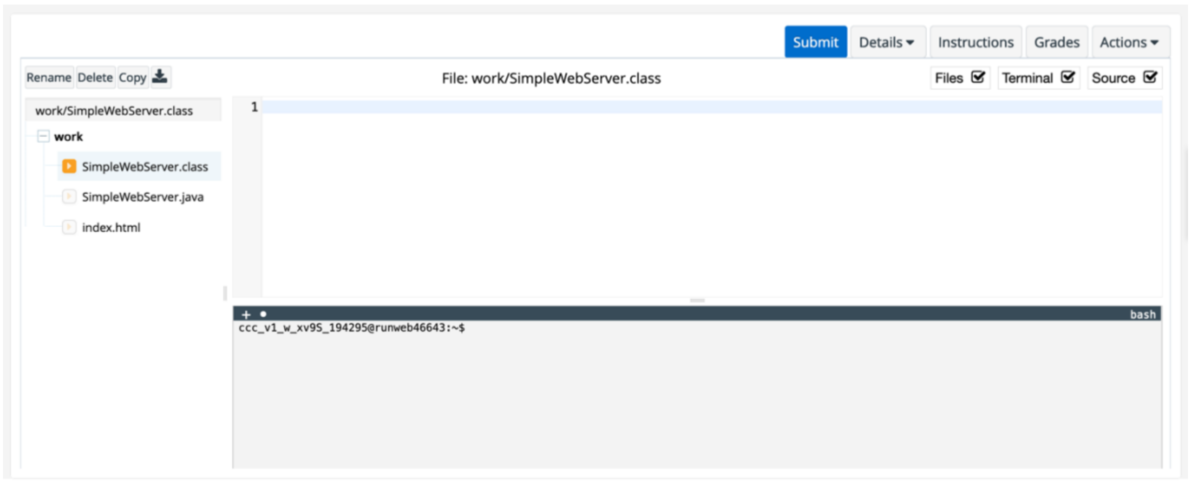
The goal of this assignment is to help you understand how a simple Denial of Service (DoS) attack brings down a server. You will carry out a DoS attack against "SimpleWebServer" that you will run inside the Vocareum Lab environment.

*NOTES:*

- *This assignment is optional and is not graded, but it is highly recommended you complete it.*
- *In this exercise you don't need to change nor download any code. The exercise is completed by sending HTTP requests to the web server.*

**Files:**

There should be 3 files in your work area on the left panel. You will see a preview of the files on the top right, and the terminal window on the bottom right – this is the area where you will work.



```
ccc_v1_w_xv9S_194295@runweb46643:~$
```

This is a description of each file in your work area:

**SimpleWebServer.java:** The simple java file that implements the web server functionalities.

**SimpleWebServer.class:** This is the class file compiled from the SimpleWebServer.java. This class file will be executed to bring up the web server.

**index.html:** A simple html file that displays a "Hello World" message.

# Denial of Service (DoS) Attack Lab

**Objective:**

Your goal in this exercise is to send a request to the web server in order to bring it down. You will not change any code in this exercise.

A successful attack means you won't get a response from the server. That is, instead of seeing a response message from SimpleWebServer:

> "Hello world"

You will see an error page

> "404: Not Found"

You can find a few hints below, but we encourage you to try attacking the running web server yourself in any way that brings it down.

**Environment:**

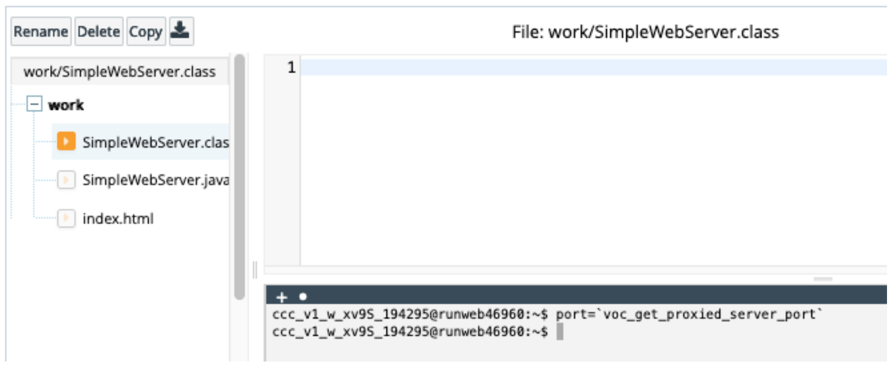
You don't need to compile the java file. The SimpleWebServer.java is already compiled to SimpleWebServer.class. You can use any web browser to send requests to the web server.

**Set Up Help and Hints:**

**1.** For security reasons, all web server applications must be run on a dynamic port through proxy. Therefore, before starting SimpleWebServer you must first get the port set up.

In your terminal, type the following command (note that the character is ` , not ‘):

```
port=`voc_get_proxied_server_port`
```



```
ccc_v1_w_xv9S_194295@runweb46960:~$ port=`voc_get_proxied_server_port`
ccc_v1_w_xv9S_194295@runweb46960:~$
```

To view the dynamic port number, type the following command (you might need the information about the port number later):

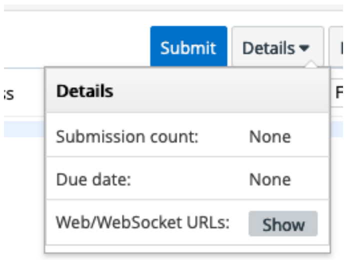
```
echo $port
```

**2.** Once you have the port, you can start the web server SimpleWebServer by typing:

```
java SimpleWebServer $port
```

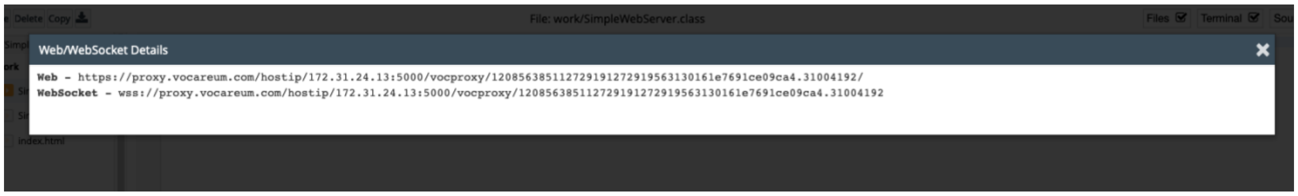
You will see a blinking cursor identifying that your server is running.

**3.** Find “Details” on the upper navigation pane of the Lab Environment, and click on “Show” next to “Web/WebSocket URLs”



| Details | |
|---|---|
| Submission count: | None |
| Due date: | None |
| Web/WebSocket URLs: | Show |

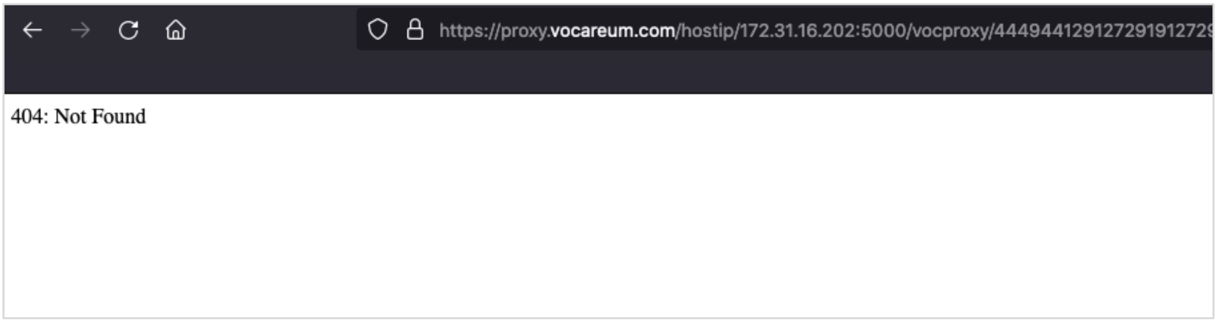
**4.** This will open a new window. Copy the Web URL link (not the WebSocket) and paste it in a browser window on your computer. If your server is running, you should see: "**Hello World**".



```
Web - https://proxy.vocareum.com/hostip/172.31.24.13:5000/vocproxy/120856385112729191272919563130161e7691ce09ca4.31004192/
WebSocket - wss://proxy.vocareum.com/hostip/172.31.24.13:5000/vocproxy/120856385112729191272919563130161e7691ce09ca4.31004192
```

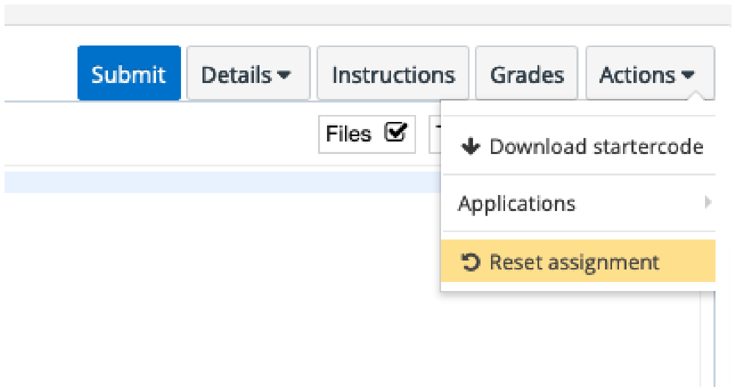
**5.** With the SimpleWebServer running (you can always check in your browser, if it's still showing "**Hello World**"), click on "+" in the upper left corner of the terminal to open a separate terminal window. What type of request can you send to the local running web server and which $port number? Perhaps, you could use *telnet*?

**6.** After your attack crashes the running web server, refresh your web browser. Instead of "Hello World", you should now get a "**404: Not Found**". Success!

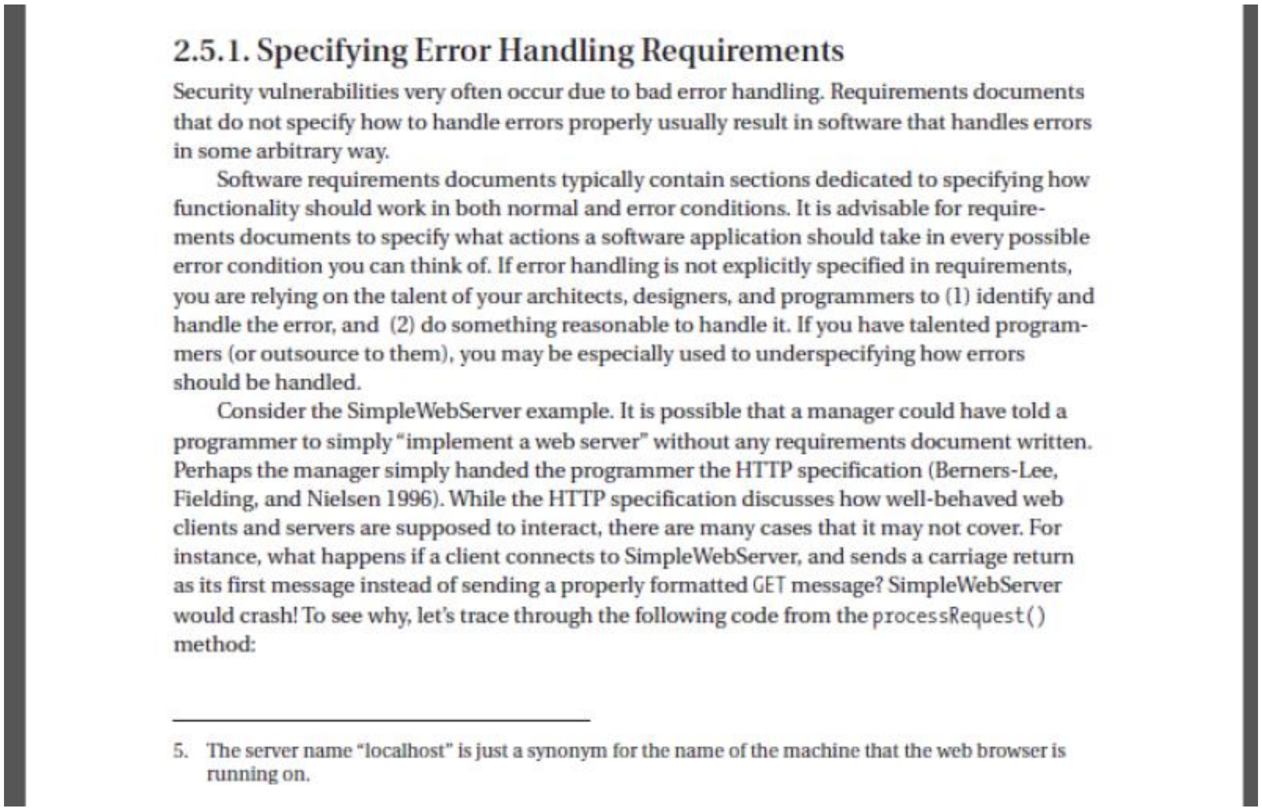

https://proxy.vocareum.com/hostip/172.31.16.202:5000/vocproxy/44494412912729191272

404: Not Found

*At any point, if you would like to reset the assignment and start over, find "Actions" and "Reset assignment" on the upper navigation pane.*



If you would like to read more about this example, you can review the following excerpt from the book Foundations of Security by Daswani, Kern, and Kesavan (images pasted below).



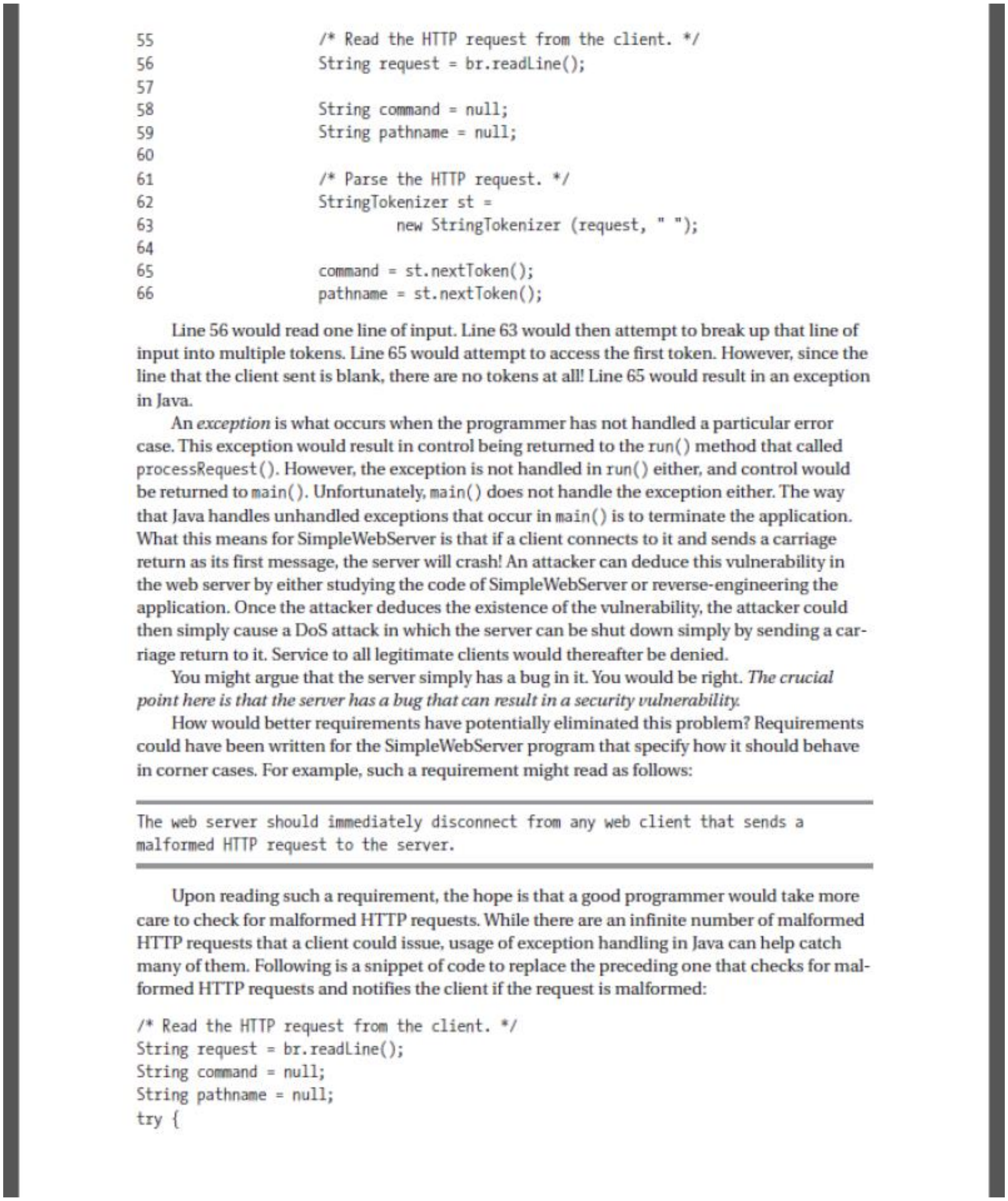
### 2.5.1. Specifying Error Handling Requirements

Security vulnerabilities very often occur due to bad error handling. Requirements documents that do not specify how to handle errors properly usually result in software that handles errors in some arbitrary way.

Software requirements documents typically contain sections dedicated to specifying how functionality should work in both normal and error conditions. It is advisable for requirements documents to specify what actions a software application should take in every possible error condition you can think of. If error handling is not explicitly specified in requirements, you are relying on the talent of your architects, designers, and programmers to (1) identify and handle the error, and (2) do something reasonable to handle it. If you have talented programmers (or outsource to them), you may be especially used to underspecifying how errors should be handled.

Consider the SimpleWebServer example. It is possible that a manager could have told a programmer to simply "implement a web server" without any requirements document written. Perhaps the manager simply handed the programmer the HTTP specification (Berners-Lee, Fielding, and Nielsen 1996). While the HTTP specification discusses how well-behaved web clients and servers are supposed to interact, there are many cases that it may not cover. For instance, what happens if a client connects to SimpleWebServer, and sends a carriage return as its first message instead of sending a properly formatted `GET` message? SimpleWebServer would crash! To see why, let's trace through the following code from the `processRequest()` method:

5. The server name "localhost" is just a synonym for the name of the machine that the web browser is running on.



```
55                  /* Read the HTTP request from the client. */
56                  String request = br.readLine();
57
58                  String command = null;
59                  String pathname = null;
60
61                  /* Parse the HTTP request. */
62                  StringTokenizer st =
63                          new StringTokenizer (request, " ");
64
65                  command = st.nextToken();
66                  pathname = st.nextToken();
```

Line 56 would read one line of input. Line 63 would then attempt to break up that line of input into multiple tokens. Line 65 would attempt to access the first token. However, since the line that the client sent is blank, there are no tokens at all! Line 65 would result in an exception in Java.

An *exception* is what occurs when the programmer has not handled a particular error case. This exception would result in control being returned to the `run()` method that called `processRequest()`. However, the exception is not handled in `run()` either, and control would be returned to `main()`. Unfortunately, `main()` does not handle the exception either. The way that Java handles unhandled exceptions that occur in `main()` is to terminate the application. What this means for SimpleWebServer is that if a client connects to it and sends a carriage return as its first message, the server will crash! An attacker can deduce this vulnerability in the web server by either studying the code of SimpleWebServer or reverse-engineering the application. Once the attacker deduces the existence of the vulnerability, the attacker could then simply cause a DoS attack in which the server can be shut down simply by sending a carriage return to it. Service to all legitimate clients would thereafter be denied.

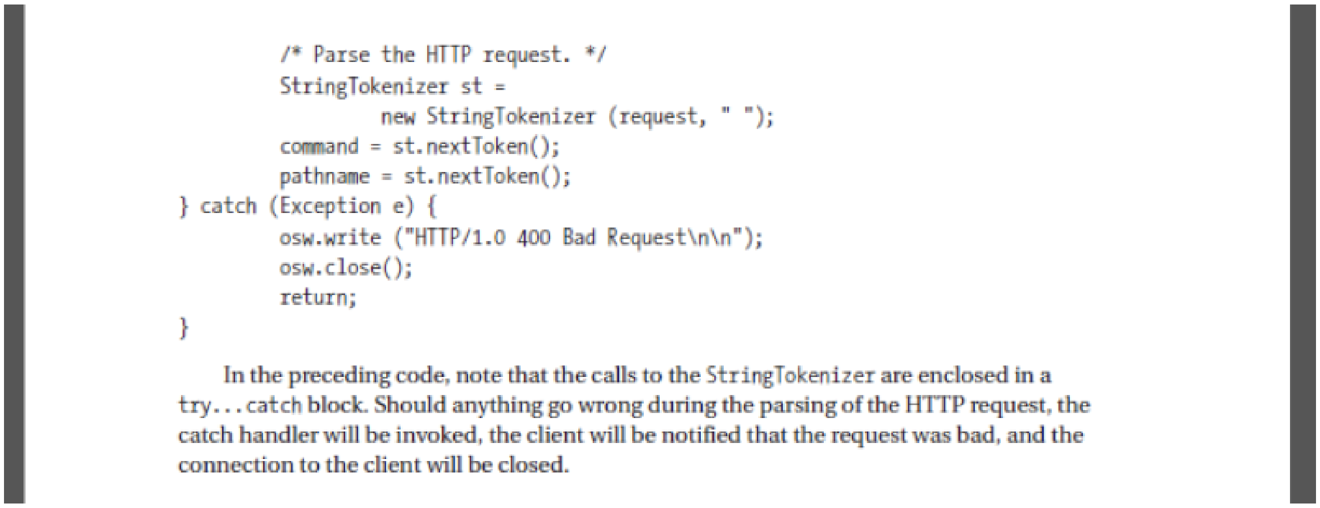
You might argue that the server simply has a bug in it. You would be right. *The crucial point here is that the server has a bug that can result in a security vulnerability.*

How would better requirements have potentially eliminated this problem? Requirements could have been written for the SimpleWebServer program that specify how it should behave in corner cases. For example, such a requirement might read as follows:

```
The web server should immediately disconnect from any web client that sends a
malformed HTTP request to the server.
```

Upon reading such a requirement, the hope is that a good programmer would take more care to check for malformed HTTP requests. While there are an infinite number of malformed HTTP requests that a client could issue, usage of exception handling in Java can help catch many of them. Following is a snippet of code to replace the preceding one that checks for malformed HTTP requests and notifies the client if the request is malformed:

```
/* Read the HTTP request from the client. */
String request = br.readLine();
String command = null;
String pathname = null;
try {
```



```
        /* Parse the HTTP request. */
        StringTokenizer st =
                new StringTokenizer (request, " ");
        command = st.nextToken();
        pathname = st.nextToken();
} catch (Exception e) {
        osw.write ("HTTP/1.0 400 Bad Request\n\n");
        osw.close();
        return;
}
```

In the preceding code, note that the calls to the `StringTokenizer` are enclosed in a `try...catch` block. Should anything go wrong during the parsing of the HTTP request, the catch handler will be invoked, the client will be notified that the request was bad, and the connection to the client will be closed.