# Java Advanced

Part II



# Generate the initial project

We will use maven to generate our initial project.

Run this command on a single line:

```
mvn archetype:generate -DgroupId=isen.[youralias]
-DartifactId=httpserver
-DarchetypeArtifactId=maven-archetype-quickstart
```

Define value for property 'version':  1.0-SNAPSHOT: :
[leave the default value by pressing the Enter key]

Then confirm. The maven source files are now generated.

# Execute the build and run the app

**Challenge #1**: Compile the project and execute the main class using java command line.
**Challenge #2**: Import your project in your IDE as a maven project.

# Code a basic HTTP Server

## Create the TCP server

Edit App.java class that was generated in the package `isen.[youralias]`

In the main method, create the Socket server and wait for a client to connect.

```
serverSocket = new ServerSocket(8999);
Socket socket = serverSocket.accept();
```

Handle the exceptions, compile and start your server in order to see if there is no firewall restrictions. Using any TCP client (telnet, curl, wget, firefox…), try to connect to your server that is now listening on 8999 on your machine (localhost).

If something goes wrong, in order to understand the behavior of your server, set breakpoints and start your server in debug mode, watch the variables.

### Try / catch / finally

The server socket opens a socket between you and your client. This socket must be closed in order to free the TCP connections in your system.

In your main method, use a finally block to close the server socket. Remember, a variable that is declared inside a code block {} is only visible from inside the code block, if you want to access to a variable outside of a code block, declare it outside of it.

```
try {
    // Create your socket server...
} finally {
                // call the close method of your server socket
}
```

## Input / Output Streams

In order to establish a discussion with the client, we need:

- An InputStream to read the message from our client.
- An OutputStream to write a message to our client.

You can find these streams available in the `socket` instance you've received when a client connected. Using the javadoc, on a single line of code, turn the `input` and output `streams` of the socket instance respectively into a `BufferedReader` and a `PrintWriter` instance:

```
BufferedReader in = new BufferedReader(//use the socket
```

```
instance);
PrintWriter out = new PrintWriter(// use the socket instance);
```

Now, read and display in the output console what comes from the input Stream.

In order to do so we need to read the inputstream until there is no more content, ie until we read an empty line.

In a while loop, read a line coming from `in` and display it in the console:

```
String line = null;
while ((line = in.readLine()) != null) {
     // display the content of the line in the console
     // don't forget to break the loop when you read an empty line
}
```

Open your web browser and go to the URL of your server http://localhost:8999, if everything went fine, you should see the HTTP protocol sent by your browser displaying in your console.

Now, we are going to reply to the client using HTTP with some HTML content

In your class, not in the main method, declare a static String that will be the body of your HTML response. For now, this will be the response you will send to any client connecting to your server.

```
private static String HTML_BODY = "<html><body><h1>Hello World!</body></html>";
```

Obviously, the output stream we've received from the socket instance will be used to write some content to our client. We need to prepare our response following the HTTP protocol.

Using the `print` method of your `PrintWriter` instance, write the following HTTP content:

| Content to write | Definition |
|---|---|
| `"HTTP/1.1 200 \r\n"` | The version of HTTP protocol and the status Code. 200 means OK. |
| `"Content-Type: text/plain\r\n"` | The content that the client receives shall be interpreted as a basic plain text |
| `"Connection: close\r\n"` | After reading the body of this content, the client can close his connection |
| `"\r\n"` | The header section is finished. Next will be the body. |

After you've set the headers, put the `HTML_BODY` defined previously in the output stream.

```
out.print(HTML_BODY + "\r\n");
```

Now, for everything that's in the output to be actually sent to the client, we need to flush it. Call the flush method on your output stream:

```
out.flush();
```

Then you can close the socket input and outputs so that the browser does not expect anymore content from the server.

```
out.close();
in.close();
```

For testing, open your web browser at the page: http://localhost:8999 and see your html body displayed.

Change the appropriate response header so that the HTML page displays properly and not the HTML code in plain text.

Use different http clients/browsers in order to see in the output of your server the differences between all the HTTP headers depending on the client (like the User Agent for example).

# Infinite Loop

As you've noticed, every request you send to your server is now working, but you have to start your server every time you want to accept a request. This is clearly not a server "by the book". We need the server to be alive and ready to "serve" all the time.

Put an infinite loop so that when a client request was performed, the server goes back to waiting for another client request.

```
while (true) {
    // put your code here
}
```

## Arrays + Command line arguments + Exceptions

You can put arguments on the java command line, after the class name you are starting. These arguments are passed to the main method in the `args` array, like shown below:

```
public static void main(String[] args) { //...
```

Modify your code so that the following command starts your server on the specified port:

```
java -cp target/httpserver-1.0-SNAPSHOT.jar isen.[youralias].App <port>
```

If the port number is not specified on the command line, use a default one. Make a try/catch block and use the exceptions `ArrayIndexOutOfBoundsException` or `NumberFormatException` to determine if the port was passed as a parameter or is a valid integer. Use the javadoc in order to convert a String into an Integer.

# Administrate the Server with JMX

The idea here is to manage remotely the server.

As you've noticed after you've set the infinite loop, now the only way to stop your server is to kill the java process. This is not appropriate as it doesn't release properly the TCP connections.

We will use JMX to write a method to check the server status and shut it down properly.

## Check server status

Create a new java Class named `ServerAdmin` implementing an interface Named `ServerAdminMBean` and a new enum named `ServerStatus`.

The ServerStatus enum will help us to define the state of the server lifecycle. Please define the following status: STARTING, RUNNING, SHUTTING_DOWN, ERROR.

Here are the specifications of the `ServerAdminMBean` interface:

```
public interface ServerAdminMBean {
    public String getStatus();
}
```

In the ServerAdmin class, implement the `ServerAdminMBean` interface methods plus the following method (you must add an attribute named "status" in your class):

```
public void setStatus(ServerStatus status) {
        // your code here
}
```

Now we will register our JMX MBean instance to the MBean Server of our JVM.
Open your server class and add the following code in the main method (don't forget to replace your alias):

```
// Get the MBean server
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
// register the MBean
ServerAdmin serverAdmin = new ServerAdmin();
ObjectName name = new ObjectName("com.[alias]:type=ServerAdmin");
mbs.registerMBean(serverAdmin, name);
```

In your code, set the status `RUNNING` of your server calling the appropriate method on `serverAdmin` at the proper place (ie, after you've created the `SocketServer`)

Now start your server and launch the Java JConsole (located in your JAVA_HOME/bin). Connect to the VM of your server (Insecure Connection is fine) and click on the MBean tab. You should see your MBean object. Reach for the attributes and check the status of your server. It should be in `RUNNING` state.

## Shut down the server properly

Now we want to be able to shutdown the server properly. Create a new method in the `ServerAdminMBean` interface named `shutdown()`.

```
public void shutdown();
```

Now in order to shutdown the server, we will need to call the `close()` method on our `ServerSocket` instance when this `shutdown()` method is called remotely from the JConsole. To do so, you will need:

- To create a method in `ServerAdmin` class to pass a `ServerSocket` instance as a parameter (call it `set ServerSocket`)
- To store the `SocketServer` you've passed as a member variable of `ServerAdmin class`
- To call the `close()` method on this `ServerSocket` instance in the `shutdown` method. This should make the `accept()` method raise an `IOException` and therefore stop the server.

Once you've written the code, start your server and your JConsole, navigate to the MBean panel, select your AdminServer MBean, call the shutdown method and see what happens.

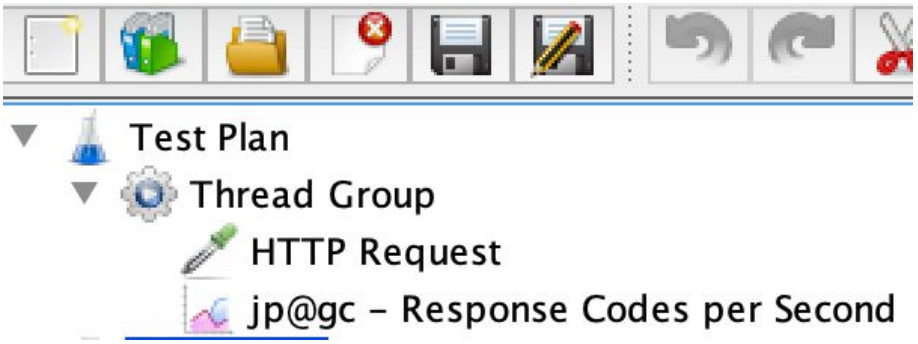
Make the appropriate changes in your code to not display any exception in the console that happens normally when you shutdown the server. Just output a message *"Shutting down the server…"*

# HTTP Testing with JMeter

Download Apache JMeter. Download additional graphs plugin and unzip it in the JMeter folder you've just installed.

Add the Following elements in your Test Plan (right click each time you want to add an element):

- A Thread Group: Add > Threads > Thread Group
- An HTTP Request: Right Click on Thread Group > Add > Sampler > HTTP Request
- A result Tree: Right Click on Thread Group > Add > Listener > jp@gc - Response Codes per Second




Save your test plan in your project folder: src / test / resources.

On the HTTP Request element, set the Server Name as localhost and Port as 8999 (or the port you've started you server with). Edit the Thread Group and in the Loop Count element, put the value 100. This means it JMeter will send 100 requests to your server.

Launch your server if not already running, then start the JMeter test (use the Green Play button). Then click on the "Hits per Second" elements in order to see the result.

You should see that your server is very fast, it can handle your requests with no time. Now we are going to simulate that your server is actually working and that every request takes 1 second for your server to perform.

Place the following code before you flush the output to your client (don't forget to handle any exception that might occur):

```
Thread.sleep(1000);
```

Restart your server, clear the results of JMeter (right-click on Hits per Second > clear).
Re-run the tests and watch the results  on the Hits per Seconds graph.
You should see that now your server can handle only one request per second.

Increase the number of Threads in your JMeter test plan: Click on Thead Group > Number of Threads > 10. Add the jp@gc Response codes per second.
Run your test, watch the new graph data. You should observe that despite the number of clients sending requests to your server simultaneously, your server still cannot handle more than one request per second.

# Multi-Threaded Server

We are going to modify our server so it can handle simultaneously multiple client requests.

Place the java code dealing with the `Socket` instance into a method having one parameter: the `Socket` instance (*tip*: you can do this easily with the Refactoring menu of your IDE).

```
public void handleRequest()
```

Create a new class named `Client` (inside the same java file, after your server class), and place the `handleRequest` method inside.

```
class Client implements Runnable {
    // place your code here
}
```

In this new `Client` class, do the following:

- Create a constructor with one parameter which is a `Socket` instance
- Save this Socket instance into a member variable named `socket` of your `Client` class
- Implement the `run()` method inherited from the `Runnable` class by calling the `handleRequest(socket)` method.
- Handle the exceptions by catching them and throw a new `RuntimeException(e)` when they occur

Back to your server code, after the server accepts a new connection, create a new instance of your `Client` class and pass it the `Socket` instance returned by the `accept()` method. Create a new `Thread` for your client instance and run it, here is the code to do that:

```
new Thread(new Client(socket)).start();
```

Start your server, go to your JMeter and run the test plan. Observe what's happening in the Responses code per second results. Now your server can handle multiple client requests simultaneously.

Run the JConsole, connect to your server's MBean server and watch the memory and thread usage.

Play with the number of Threads of your Thread Group in JMeter, increase to 100 or 500, relaunch the test plan and see what happens.

# Thread Pools

It's not safe to let the number of Threads created by your server to be controlled by the number of requests from the clients. Your java application could easily run out of resources.

We will implement a Thread Pool that will handle our client requests. This thread pool will have a maximum fixed value so that the clients requesting a connection will have to wait until a Thread is ready to handle their request if the pool is already full.

First, we need a Java ExecutorService, which is basically a thread pool manager. Declare the following in your HTTP server class.

```
private static ExecutorService executor = Executors.newFixedThreadPool(20);
```

Then instead of manually starting a client thread like this:

```
new Thread(new Client(socket)).start();
```

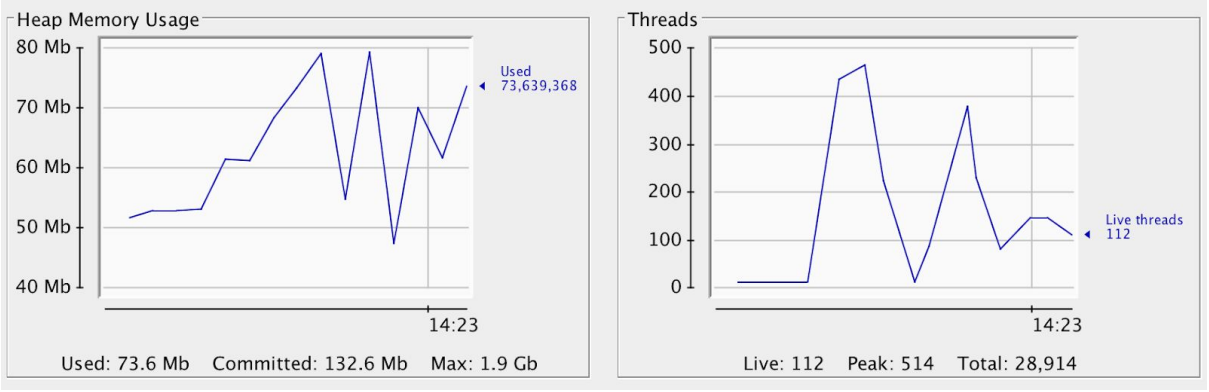
We will request the Java executor service to handle the request like that:

```
executor.execute(new Client(socket));
```

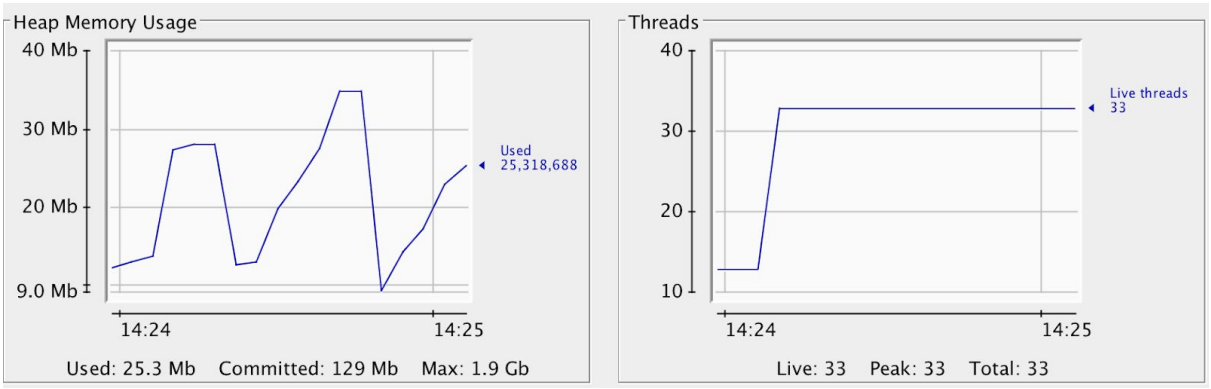
We also need to properly close the executor service as soon as the server shuts down. Place this code in your server's finally block:

```
executor.shutdown();
```

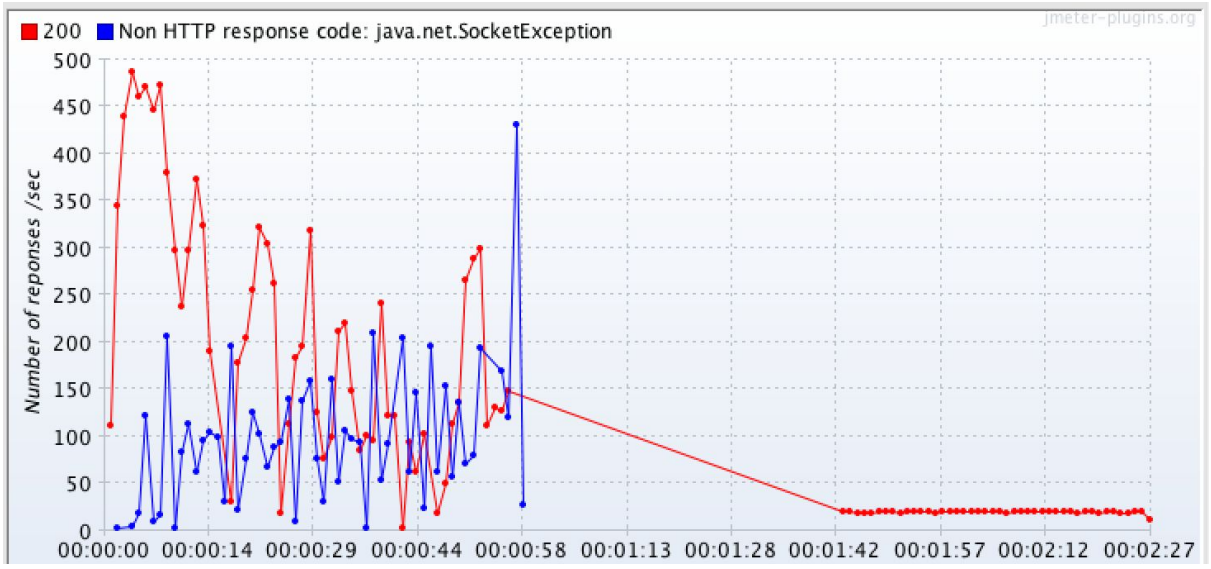
Restart you server and launch the JMeter plan. Watch the response codes and observe the JVM number of threads through the JConsole. You should see something like the following images.




*Before defining the Thread Pool*




*After defining the Thread Pool*




*Before and After defining the Thread Pool*

# Scaling & Load Balancing

With our current implementation, you've noticed that the number of requests your server can handle per second is equals to the number of threads your server can manage simultaneously. This is because we have hard coded that each request will wait 1 second until it sends its response back.

Now we will put a load balancer in front of our server, and run multiple times our server, we will use nginx in order to do that.

As stated on the dedicated nginx load balancer page: "Load balancing across multiple application instances is a commonly used technique for optimizing resource utilization, maximizing throughput, reducing latency, and ensuring fault-tolerant configurations.
It is possible to use nginx as a very efficient HTTP load balancer to distribute traffic to several application servers and to improve performance, scalability and reliability of web applications with nginx."

If you're using Windows, download nginx and install if you don't already have it. Pay attention to the location of the files after the install. On macos X, install it using brew, the important files should be located in:

```
Docroot is: /usr/local/var/www

The default port has been set in /usr/local/etc/nginx/nginx.conf to 8080 so that
nginx can run without sudo.

nginx will load all files in /usr/local/etc/nginx/servers/.
```

Edit nginx configuration file (`nginx.conf`) and put the following in the `http` section:

```
http {
    upstream myapp {
        server localhost:8998;
        server localhost:8999;
    }

    server {
        listen 8080;
        location / {
            proxy_pass http://myapp;
        }
    }
}
```

Start nginx.

Configure your JMeter test plan to target `localhost:8080` server (this parameter is in the HTTP Request element). Set the number of threads to 40 and the loop count to 1000. Clear

the Response code per Result and launch your test. Watch the output, it should still be 20 requests per second.

Now we are going to scale horizontally. Launch a new http server, specifying the port to 8998. Once the server has started, go back to JMeter and look at the number of requests per seconds, it should have doubled.

Congratulations, you've just scaled your web server horizontally. Now stop one of the two http servers you've started and watch on JMeter the request per seconds go back to 20.

# Integrate with your Library

Now, we are going to modify our code so that our web server can display the list of books of our Library. Here is the request that your server should be able to handle:

```
http://localhost:8999/books
```

The code to replace is that one:

```
out.print(App.HTML_BODY + "\r\n");
Thread.sleep(1000);
```

On one hand, you'll use the BookMain class in order to create a set of Books that are persisted in a file on the hard drive (use the proper BookDAO implementation in order to do so).

On the other hand, upon HTTP request from a client, you'll generate HTML to display the list of books that are available in the library.

You'll need:

- Regular expressions to analyze the HTTP request coming from your client.
- Maven dependencies to integrate the classes from the Library project

# Logging

Use Log message for the debug messages you display in the console using `System.out.println()`

Documentation can be found here:
https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html