The coursework is in two parts. Part 1 requires you to solve a problem involving calculation of Effective Access Times. Part 2 requires you to implement a simple HTTP server in Java. This document describes Part 2.

Part 2: HTTP Web Server

For this part of the coursework you are required to implement a very simple HTTP web server. A web server waits for incoming client connections on a specified port. Once a client connects to the server, the client can issue HTTP requests. Web servers are typically able to service multiple client requests concurrently using threads.

In what follows we discuss the requirements in more detail and outline the necessary implementation steps. In a companion document (**Testing and Tips**) we discuss useful tools that you can use while coding and debugging your web server.

We have provided a NetBeans project as your starting point. This contains a skeleton implementation of the web server itself and is bundled with libraries which provide classes for parsing, constructing and querying HTTP messages. You don't have to develop in NetBeans if you prefer to use a different IDE but **you must use the code and libraries provided.** You are **not permitted** to use any other libraries except for the standard Java libraries. Implementations which violate these constraints will be awarded zero marks.

1. Core Functionality [40 marks]

1.1. Overview

Study the provided Java source file for the WebServer class. You are required to implement your solution by adding code to this class. Your final submission must consist of the **single** file WebServer.java (it is not necessary to define any additional classes for this project but, if you feel the need, you must define them as nested classes *inside* WebServer).

The WebServer constructor accepts two arguments:

1. `port`: an integer; this is the port number on which your server will listen for incoming connections
2. `rootDir`: a string; this is the path to a folder on the local file system from which your server code must fetch files in response to client requests. This path may either be a relative path, interpreted relative to the Java working directory, or an absolute path. In either case, the URI in an HTTP request will be interpreted relative to this path (see Legitimate URI in the *HTTP protocol specifications* section later in this document).

Running the server

Your server application must accept two input parameters at invocation, as follows:

```
java webserver.WebServer port-number root-dir
```

The *port-number* and *root-dir* arguments correspond directly to `port` and `rootDir` as described above. Note that command line argument processing has already been implemented for you. For this to work from the command line, you must set the Java classpath appropriately (else the JVM won't be able to find your WebServer class and/or the supporting library classes). If you develop in NetBeans you can, of course just run your server in NetBeans (to change the port number and/or the root directory you will have to edit the project's Run configuration). If you *do* want to run it at the command line, first find out where NetBeans has created the distribution Jar for your project; it will be something like

```
C:\Users\seb\WebServer\dist\WebServer.jar
```

Then if you cd into the `dist` directory

```
cd C:\Users\seb\WebServer\dist
```

you could run your server like so:

```
java -cp WebServer.jar webserver.WebServer 1200 /srv/data
```

(the command line option `-cp WebServer.jar` tells the JVM where to look for class files).

Implementing the `start()` method

Your code should use the `java.net` library classes `ServerSocket` and `Socket`. The port on which the server listens is specified when the ServerSocket is created:

```
ServerSocket serverSocket = new ServerSocket(port);
```

The `ServerSocket` method `accept()` is then used to listen for incoming connection requests; when a connection is accepted, a new `Socket` object is created which encapsulates input and output streams for communicating with the remote client:

```
Socket connection = myServerSocket.accept();
```

Your `start()` method will have this basic form:

```
1: create a server socket;
2: while (true) {
3:    listen for a new connection on the server socket;
4:    process an HTTP request over the new connection
5: }
```

Consider how your server will behave if it is single-threaded. It will only be able to process one HTTP request at a time, because it will not return to the start of the while loop (and so be able to accept a new connection) until it has finished processing the request at step 4. For higher marks, you should make your server multi-threaded, by spawning a new thread at step 4. We recommend that you start by implementing and thoroughly testing a single-threaded server, then make it multi-threaded when you are confident that it is working correctly.

Reading and Writing HTTP messages

Use the library class `http.RequestMessage` to read and parse request messages from the input stream provided by the client connection socket. Use the library class `http.ResponseMessage` to build response messages and write them back to the client on the output stream provided by the client connection socket. Consult the provided API documentation for details.

Robustness and Error Reporting

Your web server should run forever and never crash. This is not 100% achievable in practice, since unrecoverable errors may arise in the host OS (eg file system failure). However, you should aim to make your server as robust as possible. In particular, even if a client behaves badly (e.g. by issuing a malformed HTTP request or by prematurely closing a connection) the server should not crash. If coded correctly, a multi-threaded server will tend to be more robust than a single-threaded one, since the main server loop will survive even if one of the child threads crashes; even so, your server should strive to detect these kinds of problems and (where possible) send an appropriate HTTP response to the client, rather than simply terminating the connection.

One aspect of this concerns exception handling. You should consider carefully what exceptions might be thrown by your code (your compiler or IDE will help identify those places where exceptions might be thrown) and how to catch and handle each one. In particular, you should consider in each case whether it is possible to send an appropriate HTTP response to the client.

HTTP[1]

The Hypertext Transfer Prototol (HTTP) protocol is used in the World Wide Web for clients/users to communicate with servers. The protocol defines communication in the form of client requests to servers and server responses to clients. For every client *request message*, the server responds with a *response message*. There are nine methods that an http client can request a server to carry out. In this coursework you are asked to implement a server that can process just GET. Before discussing the behavioural specifications, we first describe the format of the relevant HTTP messages. You can find a comprehensive description of the formats of HTTP messages in HTTP/1.1 RFC 2616. Here we provide a shorter description that is probably sufficient for this coursework.

[1] The current document uses RFC 2616 as the HTTP/1.1 spec, rather than the more recent RFC's 7230-7235.

1.2 HTTP Message Format

An HTTP (request or response) message has the following format:

message-header

empty line

message-body (optional)

The ***message-body*** is an optional part of a message containing data associated with a request or response. For example, when a client wishes to upload a file to the server, the message-body contains the actual contents of the file to be uploaded (not needed for the coursework).

The ***message-header*** has the format:

start-line

header-line(s) (optional)

where:

start-line specifies a specific type of request or response (see examples below)

header-line(s) (also known as *header-fields* or *message-headers*) carry additional information required by a request or response. This part of an HTTP message may contain zero or more lines and each line refers to a different type of information. Each header line has the following format:

field-name : *field-value*

where:

field-name is the name of the type of information associated with the current message; and ***field-value*** is the value of this header field. For example, consider the following header-lines:

```
Host: www.google.com      ➔ the domain name of the server
From: eva@example.com     ➔ email address of the user sending the request
```

Note that there are many different header fields. However for the core functionality we only require your code to consume header lines without acting upon them. Header lines become more significant for the conditional GET request Advanced feature (see below).

Let's consider a simple example without any header-lines.

Example A client wishes to download the file `test` from the server. The request message starts with the keyword GET, followed by the URI for the requested file, followed by an http version string. Suppose the file exists and contains the text "This is a test". Then the following exchange of two messages should occur:

Request message, client → server:

```
GET /public/test HTTP/1.1
```

Response message, server → client:

```
HTTP/1.1 200 OK

This is a test
```

(Note: there are empty lines after the headers of both messages.)

As mentioned in Section 1, library classes RequestMessage and ResponseMessage have been provided to allow you easily to read, construct and write messages on the input and output streams of a client connection socket.

1.3 The GET method

The GET method is used by clients to download content (for example an html file) from the server's root directory. It has the following syntax:

`GET` *URI* `HTTP/1.1`

where *URI* is a path that points to a file on the server's file system (see below for the URI format).

Processing a GET request is in essence a two-stage process along the following lines:

1. Read the message header:

```
InputStream inStream = connection.getInputStream();

RequestMessage msg = RequestMessage.parse(inStream);
```

2. Send back the data:

```
OutputStream outStream = connection.getOutputStream();
```

then open the file identified by `msg.getURI()` and copy its contents back to the client on `outStream`. If your Java file-handling skills are rusty, consult the accompanying **JavaNIO** document for help.

Uniform Resource Identifiers (URI)

Before describing the protocol specifications to handle a GET request we discuss the URI format. As specified in HTTP 1.1 RFC 2616: *"As far as HTTP is concerned, Uniform Resource Identifiers are simply formatted strings which identify -- via name, location, or any other characteristic -- a resource."* The complete URI specification can be found in RFC 2396. However, in the current coursework you need only use the following specification.

Your code needs only to support URIs which conform to a hierarchical (tree-like) naming scheme where the different directories in the tree are separated by the delimiter "/". You may assume (but need not enforce) that directory and file names are composed only of ASCII letters and digits plus dot ("`.`"), hyphen ("`-`") and underscore ("_"). For example:

```
/clients/test_file-1.jpeg
```

Note that "`.`" and "`..`" should be interpreted as the *current* and the *parent* directories, respectively. For example the URI `/clients/child/../test.jpeg` is equivalent to `/clients/test.jpg`. and should be treated as such *even if the* `child` *directory does not exist* (the Java NIO libraries support these conventions; see the *Implementation Tip* below).

To be considered ***legitimate***, a URI must point to a *legitimate location* in the server's file system space, where a location is considered legitimate only if it lies within the designated root directory. Your implementation should interpret the URI as a path *relative* to the root and only accept paths that point to files that actually lie within the root or one of its sub-directories.

For example: if the absolute path for `rootDir` is `/srv/data` and the GET request URI is

```
/clients/index.html
```

Then the absolute path of the file identified by the URI comes from appending the URI *at the end* of the `rootDir` path:

```
/srv/data + /clients/index.html =
/srv/data/clients/index.html
```

This is legitimate. However, consider instead a GET request with URI

```
/clients/../../staff/salaries.html
```

Then the absolute path of the specified file would be:

```
/srv/data + /clients/../../staff/salaries.html =

/srv/data/clients/../../staff/salaries.html =

/srv/data/../staff/salaries.html =

/srv/staff/salaries.html
```

This is *not* legitimate, because it points to a file which does not lie within `/srv/data` or one of its sub-directories. Rewriting the path to eliminate `.` and `..` as above is known as *normalising* the path. The Java NIO classes can do this for you, as explained in the following tip.

Implementation Tip: Like Unix file systems, HTTP URI syntax uses / as the path delimiter. But Windows file systems use \ instead. So you might expect some compatibility issues in servers running on Windows platforms. Luckily, between them, the Java and Windows API's manage to do the right thing, so you shouldn't have to worry about this. For example, suppose that `myURI` is the string "`/clients/index.html`". You can safely use java.nio.Path to construct a normalised absolute path like this:

```
String ps = rootDir + myURI;

Path p = Paths.get(ps).toAbsolutePath().normalize();
```

Assuming that `rootDir` is appropriately defined, this should work fine, whether on Windows or Unix. However, you should *not* hardcode any absolute file paths (either Windows-style or Unix-style). Remember, the root-directory path is provided *on the command line* when your server is started (see Section 1.1) so there is no need to hardwire it in your code.

Responding to a GET request

Your server should check that (a) the request URI is legitimate and (b) that it identifies an existing regular file (not a directory) and (c) that the file is readable by the web server. If these conditions are met then the server should form an appropriate HTTP response and return it to the client. If any of these conditions fail then the server should reject the request, responding to the client with an appropriate HTTP status code.

1.4 Response Messages

As described above, your server must send a response message back to the client in response to each request message received. In common with request messages (as described in Section 1.1), response messages contain a start-line and a (possibly empty) message-body. Note that, for the core functionality, response messages need not include any header-lines. For a successful GET request, the message-body in the response contains the content of the file requested. The syntax of the start-line for a response message is:

`HTTP/1.1` *status-code reason-phrase*

where status-code is a three-digit integer and reason-phrase can be any string.

There are many different status codes defined in the HTTP/1.1 RFC Section 10. For the coursework core functionality you will probably need only the following codes; you should study the definitions in the RFC document carefully and ensure that your server uses them appropriately in response messages:

200	OK
304	Not Modified
400	Bad Request
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
505	HTTP Version Not Supported

For example, according to the RFC 2616 description for code 200:

> "The request has succeeded. The information returned with the response is dependent on the method used in the request, for example: GET an entity corresponding to the requested resource is sent in the response"

So if the client issues a GET request and the server finds the requested file and is prepared to return its contents, the start-line in the server's response message should use code 200:

```
HTTP/1.1 200 Ok
```

This tells the client to get ready for the data which will follow.

2. Advanced feature [10 marks]

Warning: make a safe copy of your working core implementation before attempting an advanced feature. If you break core functionality by making mistakes in the advanced feature you should submit the original version instead, or you will lose marks. We will only assess one submission per group.

Implement *one* of the following features. **Add a comment** at the top of your WebServer.java file to say which feature you have attempted. If there is no explanatory comment at the top of your submission we will not assess an advanced feature, even if the code is present (even if you have attempted both we will only assess one, so say clearly which one you want to be assessed).

2.1. Support for client-side caching (conditional GET requests)

HTTP clients (such as web browsers) may cache copies of files which they have previously downloaded from a web server. If a user issues a request for the same file again, the client may send a conditional GET request to the server. For our purposes, a conditional GET request is distinguished only by the inclusion of an `If-Modified-Since` header line in the request message. On receipt of such a message, a server may compare the field value in the header with the last-modified time of the requested file; if the last-modified time of the file is later than the If-Modified-Since field value, the response is as usual, but otherwise the server is permitted to respond with status 304 and an empty message body. Note that clients typically will only issue a conditional GET request if the response to an earlier request included a `Last-Modified` header line.

Modify your server so that it includes `Last-Modified` headers in its responses to all successful GET requests and so that it responds as above to conditional GET requests.

Note that `org.apache.http.client.utils.DateUtils` can be used to convert between the `java.Date` and HTTP date formats. The relevant part of this library is already included in the NetBeans skeleton project we provided.

2.2. Directory GET requests

For the core functionality as described above you are asked to reject GET requests where the requested file exists but is a directory rather than a regular file. Modify your server so that it allows such requests as follows: if the specified directory contains a file called `index.html` then respond with the contents of that file; otherwise, build and return an html response which displays the directory contents, with one hyperlink per file. Test your server in a web-browser; a correct implementation will result in the browser displaying a web page which allows access to the directory contents by clicking on the links displayed. (Making this work correctly even in the case that the browser is pointed at `/folder` rather than `/folder/` can be a bit tricky; try to find out what real web servers do in these cases.)

3. Submission

For Part 2 of the coursework, submit **one Java source file** named `WebServer.java`.