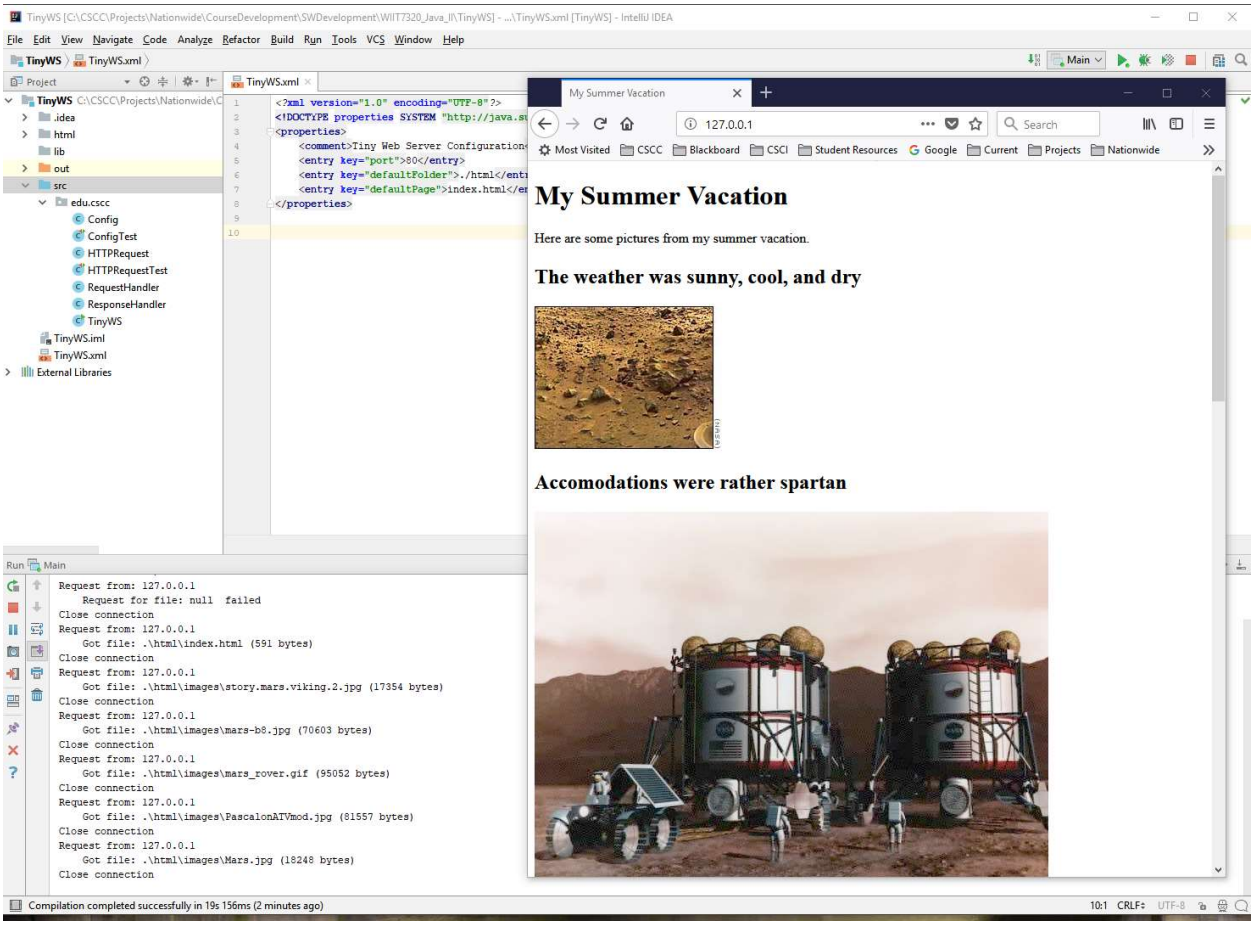
WIIT-7320 Java II Group Project



Synopsis

Working in groups of two people you will design and implement a tiny web server. A web server is system software that is used to implement a web site. The web server responds to requests from web browsers and retrieves and returns requested web files to the browsers for display to end users. Browsers may request a variety of different files including web pages (HTML files), images, styling information (CSS files), JavaScript, audio, video, and many other file types. Your web server will only handle simple web pages (HTML files) and several common types of image files.

The web server and the browser communicate using the Hypertext Transfer Protocol (HTTP). HTTP has a variety of request methods, but your tiny web server will only implement a simplified version of the GET method. The GET request consists of the word GET followed by a space followed a filepath. A delimiter (either a space or a question mark) and then additional information follows the filepath.

Your web browser will handle requests in the form

GET /path/filename <additional request info>

Examples

GET /index.html

GET /images/somepix.jpg

GET / *(note: if no filename is present, web server will look for a default filename)*

GET /index.html?name1=value1&name2=value2 *(note: ignore the '?' and all that follows)*

There may be additional information after the filepath, but your web server will ignore it. The web server will extract the path and filename and retrieve the corresponding local file. It will then encode the file as an HTTP response and send it back to the web browser. An appropriate error will be sent if the file cannot be found.

Your web server will contain five java source file and some additional files containing configuration, the sample web site, and unit tests. The five java files are:

- TinyWS.java – the tiny web server (main)
- Config.java – reads configuration information for the web server at start-up
- HTTPRequest.java – a request from a web browser
- RequestHandler.java – handles a request from a web browser
- ResponseHandler.java – returns a file or an error response to a web browser

You can read more about web servers here: https://en.wikipedia.org/wiki/Web_server

Part 1: Create TinyWS Project and Config class

Please do the following to begin implementation of your tiny web server.

1. Start a new IntelliJ Java project called TinyWS
2. Download the GroupProject_Starter.zip file from Blackboard
3. Unzip the file
4. Open the project in IntelliJ and review the includes files and folders.

We will start by implementing a Config Java classes that will be used in our web server.

The Config class will be run at start-up and will get web server information from a configuration file. The configuration file (called TinyWS.xml) is written in XML and looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>Tiny Web Server Configuration</comment>
    <entry key="port">80</entry>
    <entry key="defaultFolder">./html</entry>
    <entry key="defaultPage">index.html</entry>
</properties>
```

Your tiny web server will use three configuration values:

- port – the default HTTP port is 80. It is possible that your PC may already be using port 80 for another program. In this case you can instruct your tiny web server to use a different port.
- defaultFolder – this is the path to the web site that is served up by the tiny web server.

- defaultPage – if the requesting browser asks for a folder rather than a page (for instance '/'), this is the default web page file to use.

The Config class will use the Java Property class to process configuration information. The Java Properties class is derived from the Java Hashtable class. Please review the following:

- Tutorial: How do I read properties from an XML file - http://www.avajava.com/tutorials/lessons/how-do-i-read-properties-from-an-xml-file.html
- Properties class - https://docs.oracle.com/javase/9/docs/api/java/util/Properties.html
- Hashtable class - https://docs.oracle.com/javase/9/docs/api/java/util/Hashtable.html
- Properties topic in Schildt book (10th ed.) – Chapter 19, pages 603-608

Your class should use the following constants. The first three are property key values and the last is the location of the XML configuration file.

```
public static final String PORT = "port";
public static final String DEFAULTPAGE = "defaultPage";
public static final String DEFAULTFOLDER = "defaultFolder";

private static final String CONFIG_FILE = "./TinyWS.xml";
```

In addition to the constructor, your config class should have the following methods:

```
public void readProperties() – read the XML properties file

public String getProperty(String key) – return a value associated with a property key
public void dumpProperties() – print all property values to console (for debugging)
```

Write the code to fully implement the constructor and the above methods.

Create a unit test class (using JUnit 4) to test your new class. To create a test class in IntelliJ, highlight the class name in the source folder and type control-shift-t. Use JUnit4 as the testing library (you may need to add JUnit4 to your external libraries) and use the class name ConfigTest. Use assertions for testing. See the documentation here - https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

Document your source code using JavaDoc comments. In IntelliJ add a comment before the class definition and prior to each public and protected method. Start a comment with '/**' and then hit enter. IntelliJ will start a JavaDoc comment. Be sure and include an @author tag in the initial class description comment. A complete list of JavaDoc tags can be found in Appendix A of your textbook.

Once the class is documented, generate JavaDoc pages in IntelliJ. Go to the 'Tools' tab and select "Generate JavaDoc …". Review your generated documentation, make any required adjustments to the JavaDoc comments, and re-generate the documentation.

Part 2: HTTPRequest class

Implement the HTTPRequest Java class. This class will accept an HTTP request from a web browser and extract the relevant path of the file to be returned to the web browser.

You'll need to impelement the following methods:

- **public** HTTPRequest(String r) – (constructor) parse request, get path, determine if request is valid. The parameter r is the HTTP request string from the web browser.
- **public** String getPath() – return the path to the requested file
- **public boolean** isValidRequest() – true if the request is valid, false otherwise
- **private boolean** parse(String r) – parse the HTTP request. Break up the request into string tokens using the String split() method. Use the argument "[\t\n?]" to split the string if a space, tab, newline, or question mark are encountered. Verify there are at least two tokens. Verify the first token is 'GET'. The second token should be the file path. Make sure it is not null or empty. Return true if the request is syntactically valid, false otherwise.

Create a JUnit test file to test the class. Add JavaDoc comments and verify JavaDoc generates properly.

Part 3: TinyWS class and RequestHandler stub

Now we're ready to implement our main TinyWS class.

1. Write a main class that instantiates a TinyWS object.
2. Review the fields for port, defaultFolder, and defaultPage configuration values. Add getters for each of these fields
3. Review the following error handling methods:

```
public static void log(String s) {
    System.out.println(s);
}


public static void fatalError(String s) {
    handleError(s, null, true);
}


  public static void fatalError(Exception e) {
      handleError(null, e, true);
  }

  public static void handleError(String s, Exception e, boolean isFatal) {
      if (s != null) {
          System.out.println(s);
      }
      if (e != null) {
          e.printStackTrace();
      }
      if (isFatal) System.exit(-1);
  }
```

4. Refactor your Config and HTTPRequest code to use the above error handlers. Use them to handle error conditions in subsequent code you write.
5. Write a constructor that uses the Config class to initialize the configuration fields you just defined. Use the Config dumpProperties() method to display the configuration values.
6. Test the code you've just completed to verify that configuration works.
7. Create a listen() method that does the following. Add exception handling where required.

a. Instantiate a ServerSocket on the web server's port. See https://docs.oracle.com/javase/9/docs/api/java/net/ServerSocket.html Also, see the Schildt book (10th ed.) pages 763-764 for information of ServerSockets.
b. Use the setSoTimeOut() method with an argument of zero to allow an indefinite time-out while listening on the port.
c. Create an infinite loop that does the following:
    i. Call accept() on server socket – which waits for a request and then returns a Socket *connection* object. See https://docs.oracle.com/javase/9/docs/api/java/net/Socket.html
    ii. Log *connection*.getInetAddress().getCanonicalHostName() – which outputs the source of the HTTP request.
    iii. Instantiate a RequestHandler object that takes the Socket connection as a parameter. (This is a class you will create momentarily)
    iv. Call the processRequest() method of the RequestHandler object you just instantiated.
    v. Close the Socket connection.

8. Add a call to the listen() method in main.
9. Implement a dummy RequestHandler class. Write a constructor and a processRequest() method that prints "Got a request".
10. Try running your code. Using a browser – connect to your web server using the URL: http://127.0.0.1/
11. Debug your code as required.

Part 4: RequestHandler

Now you will fully implement your RequestHandler class. Do the following:

1. Create a field for the connection:

```
private Socket connection;
```

2. Modify the constructor to save the connection parameter in the above field.
3. Implement a private method called readRequest() that reads the HTTP request from the socket and returns a string containing the request. Use the following code:

```
connection.setSoTimeout(500);
int recbufsize = connection.getReceiveBufferSize();
InputStream in = connection.getInputStream();
```

   to set the socket timeout to 500 milliseconds, get the buffer size (in bytes) for the connection, and get the InputStream for the connection. Implement code to read the request using a BufferedReader and use a StringBuilder to build the request string.
4. Modify the processRequest() method to call readRequest() and get the HTTP request as a String.
5. Create a HTTPRequest object from the request String.
6. Create a ResponseHandler object passing the HTTPRequest object as a parameter
7. Call the ResponseHandler method sendResponse passing the Socket connection as a parameter.
8. Use the *finally* clause to always close the Socket connection.
9. Include exception and error handling as appropriate.

10. Create a ResponseHandler class stub with a stub sendResponse method. The method should log the path of the HTTPRequest.
11. Run your code and debug as appropriate.
12. Make sure you include JavaDoc comments

Part 5: ResponseHandler

Now you'll fully implement the final piece of the web server – the ResponseHandler class.

1. Review the following String constants to the class:

```
private static final String NOT_FOUND_RESPONSE =
        "HTTP/1.0 404 NotFound\n" +
                "Server: TinyWS\n" +
                "Content-type: text/plain\n\n" +
                "Requested file not found.";

private static final String FORBIDDEN_RESPONSE =
        "HTTP/1.0 403 Forbidden\n" +
                "Server: TinyWS\n" +
                "Content-type: text/plain\n\n" +
                "Requested action is forbidden.  So there!";

private static final String HTTP_OK_HEADER =
        "HTTP/1.0 200 OK\n" +
                "Server: TinyWS\n" +
                "Content-type: ";
```

2. Create an HTTPRequest object field to store the request and an int responseCode field to store the HTTP response code. Your web server will use one of the following response codes:

Code	Meaning
200	OK – request was valid, returning result
403	Forbidden – request was disallowed
404	Not Found – could not find requested file

   A complete list of HTTP status (response) codes can be found here - https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
3. Modify the constructor to save the HTTPRequest passed in and to initialize the responseCode to 404.
4. Create the following private methods:
   a. **private** String getMimeType(String path) – reads the filename suffix and returns a String containing the MIME type. Use the table below to determine the type:

File extension	MIME Type
.html	text/html
.txt	text/plain
.gif	image/gif
.jpg	image/jpeg

A more detailed list of MIME types can be found here - https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

b. **private byte**[] readFile(File f) – read a file and return a byte array. Return null if method fails. (Be sure to always close the file).

c. **private byte**[] getFile(String path) – read a file using the String path and return a byte array. This method calls readFile method above. The method also sets the appropriate response code. Be sure and handle the following:
   i. Make sure the path is not null (response code: 404)
   ii. Make sure the path does not contain ".." (security violation – response code: 403)
   iii. If the path starts with a '/' than prefix the path with the default folder.
   iv. If the path does not start with a '/' than prefix the path with the default folder and a '/'.
   v. Read about the Java File class - https://docs.oracle.com/javase/9/docs/api/java/io/File.html
   vi. Using the File class, you can check if the path is a directory, if it is not a file, if it is not readable, and if it's zero length. If it's a directory, add the default page to the end of it (you may need to add a '/' as well). The other cases are errors and you'll need to detect and handle those, as well.
   vii. Finally, if the path is valid, call readFile, and get the byte array. If the byte array is null, set the response code to 404.
   viii. If a valid byte array is returned, concatenate HTTP_OK_HEADER, the MIME type, two newlines ("\n\n"), and the retrieved file into a byte array. Set the response code to 200. You may need to use calls to System.arraycopy() to build up the final byte array.
   ix. If the response code is not valid return null.

d. Fully implement the sendResponse method. It should begin like this:

```
public void sendResponse(Socket connection) throws
IOException {
    byte[] response = null;
    int sendbufsize = connection.getSendBufferSize();
    BufferedOutputStream out = new BufferedOutputStream(
            connection.getOutputStream(), sendbufsize);
```

   i. If the HTTPRequest is a valid request call the private getFile() method you've implemented to get the byte array.
   ii. If the response is null (either it returned null or getFile() was never called because the request is invalid), test the return code. If it's 403 set the response to FORBIDDEN_RESPONSE, otherwise set it to NOT_FOUND_RESPONSE. You'll need to use the built-in String getBytes() method to convert a String to a byte array.
   iii. Output the response, flush, and close the connection.

5. Run the code, use your web browser (set to http://127.0.0.1/) to send requests to the web server. It is unlikely your web server will work the first time. Use debugging statements (add TinyWS.log() calls) and IntelliJ debugging (breakpoints) to detect and fix problems in the code.
6. Add JavaDoc comments. Generate JavaDocs for your finished working program

When you have finished your web server, demonstrate it to the instructor. Zip your IntelliJ project and submit it via Blackboard.

Extra Credit: Multithreaded Web Server

Read Chapter 11 in the text on Multithreaded Programming. Pay attention to the section **Creating Multiple Threads** on pages 246-247.

- What would you need to change in TinyWS.java to multithread your web server?
- Which methods need to be synchronized?

Modify your web server to make it multithreaded as follows:

1. Create a new class called NewRequestThread using what you learned in Chapter 11.
2. Move the request processing code in the listen() method of TinyWS to the new class. Instantiate the new class in listen() to create a new thread and process a request.
3. Synchronize any shared resources (such as writing to the console).
4. Run your threaded server.
5. Questions:
    - Does your web page render faster?
    - How would does this effect web server performance? What effect would a processor with 16 or 32 cores have on web server performance now?
    - What effect does the threaded server have on logging output? What would you do to address this?