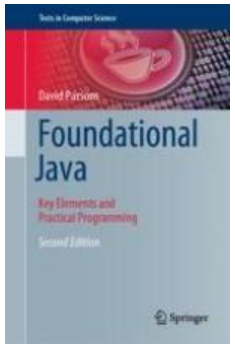


Chapter 20
Java Web Servers and the HttpClient

Foundational Java

Key Elements and Practical Programming

Web Start and Applets

- Java first came to prominence because of Java applets, running in web browsers, later followed by Java Web Start
  - These technologies were retired primarily due to security concerns, which led to increasingly limited support by web browsers
  - They were removed from the Java Standard Edition (Java SE) in 2018, along with components of Java Enterprise Edition (Java EE)
- Java SE can still be used in a Web related context
- The HttpClient class provides a mechanism for Java code to make an HTTP connection to a web server

Web Browsers, URLs and HTTP

- Web browser software retrieves information from the World Wide Web (WWW) using Uniform Resource Locators (URLs)
- URL
  - The Internet address of a resource on a particular server
  - Comprises:
    - Protocol
    - Server address
    - Name of the resource (including any path information).
- HTTP
  - The protocol prefix for web pages is ‘http://’ (hypertext transfer protocol)

Complete URLs

- The server address
  - Usually begins ‘www’ (World Wide Web), followed by the name of the site and its ‘domain’
  - Separated by periods, e.g.

```
http://www.foundjava.com
```

  - “foundjava” is the name of the server site and “com” means a company
  - Common alternatives to “com” are “edu” for academic institutions and “org” for organizations
  - Many URLs are within a country domain, such as “.co.uk” or “.ac.nz”
  - If you are running a test server on your local machine, the domain name becomes ‘localhost’.

The Path and Resource Name

- The final part of a URL can include the location (directory) and (optionally) name of the file at the site. For example (this is not a real web address):

```
http://www.mywebsite.com/docs/index.html
```

- This looks for the file “index.html” in the “docs” directory
- Increasingly, URLs that include the filename are being replaced by “clean URLs” or “slugs”, where the URL has no filename but just ends in what looks like a folder but is a reference to the page, e.g.

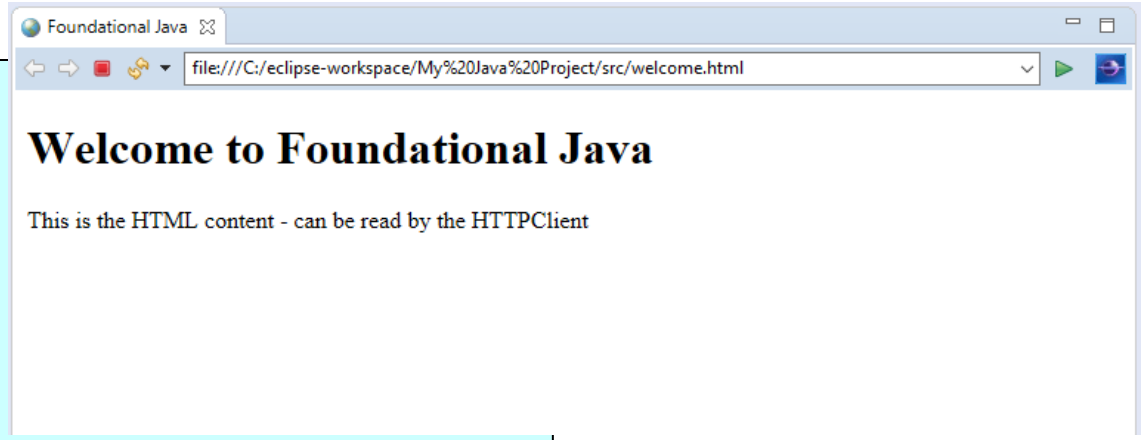
```
http://www.foundjava.com/java-code/
```

  – Note that the page uses the slug “java-code” but this is not an HTML filename.

HTML (HyperText Markup Language)

- Web browsers display screens of information written using HTML
- The browser formats the content using tags embedded into the text of the file
- All HTML files begin with an <html> tag and end with </html>
- You can view HTML files in Eclipse

```
<!DOCTYPE html>
<html>
  <head>
    <title>Foundational Java</title>
  </head>
<body>
  <h1>Welcome to Foundational Java</h1>
  <p>This is the HTML content - can be read by the HTTPClient</p>
  </body>
</html>
```



The Tomcat Server

- Tomcat is an open source Java application server that supports the Web application components of the Jakarta EE specification
- These components can generate dynamic content (i.e. web pages that are generated dynamically by code running on the server)
- Application servers like Tomcat can also serve static content, such as pre-written HTML pages, through a built-in HTTP server
- Tomcat can be downloaded as zipped archive

Installing Tomcat

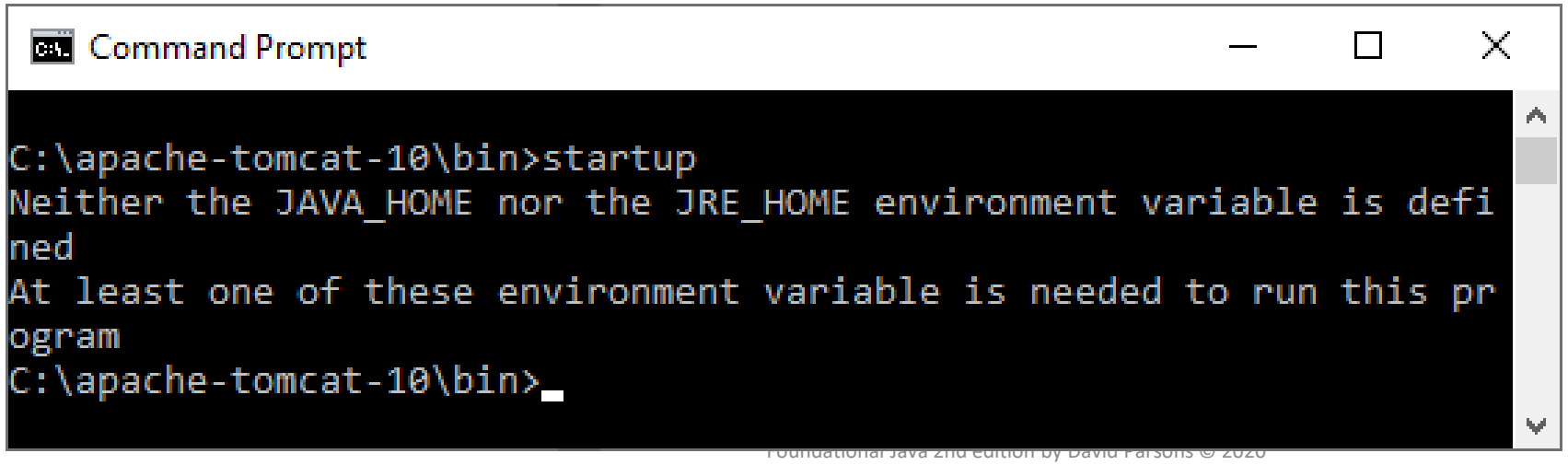
- Tomcat can be downloaded in multiple formats
- We will work through an example based on downloading a zipped archive, which can be extracted to a suitable location on your computer
- Using this type of installation helps you to get a clearer view of how Tomcat runs and how it can be configured.
- Tomcat can be downloaded from https://tomcat.apache.org/
- The following sections assume that the zipped archive version has been downloaded and extracted to a folder called “apache-tomcat-10” on the “C:” drive of a Windows machine

Starting Tomcat

- To start Tomcat, navigate to the ‘bin’ folder of the Tomcat installation directory, e.g.

```
C:\apache-tomcat-10\bin
```

- In the ‘bin’ folder there will be a file called ‘startup’ that can be used to start the server
- On first installation, startup will fail unless the JAVA_HOME environment variable has not been set to point to an installation of Java SE



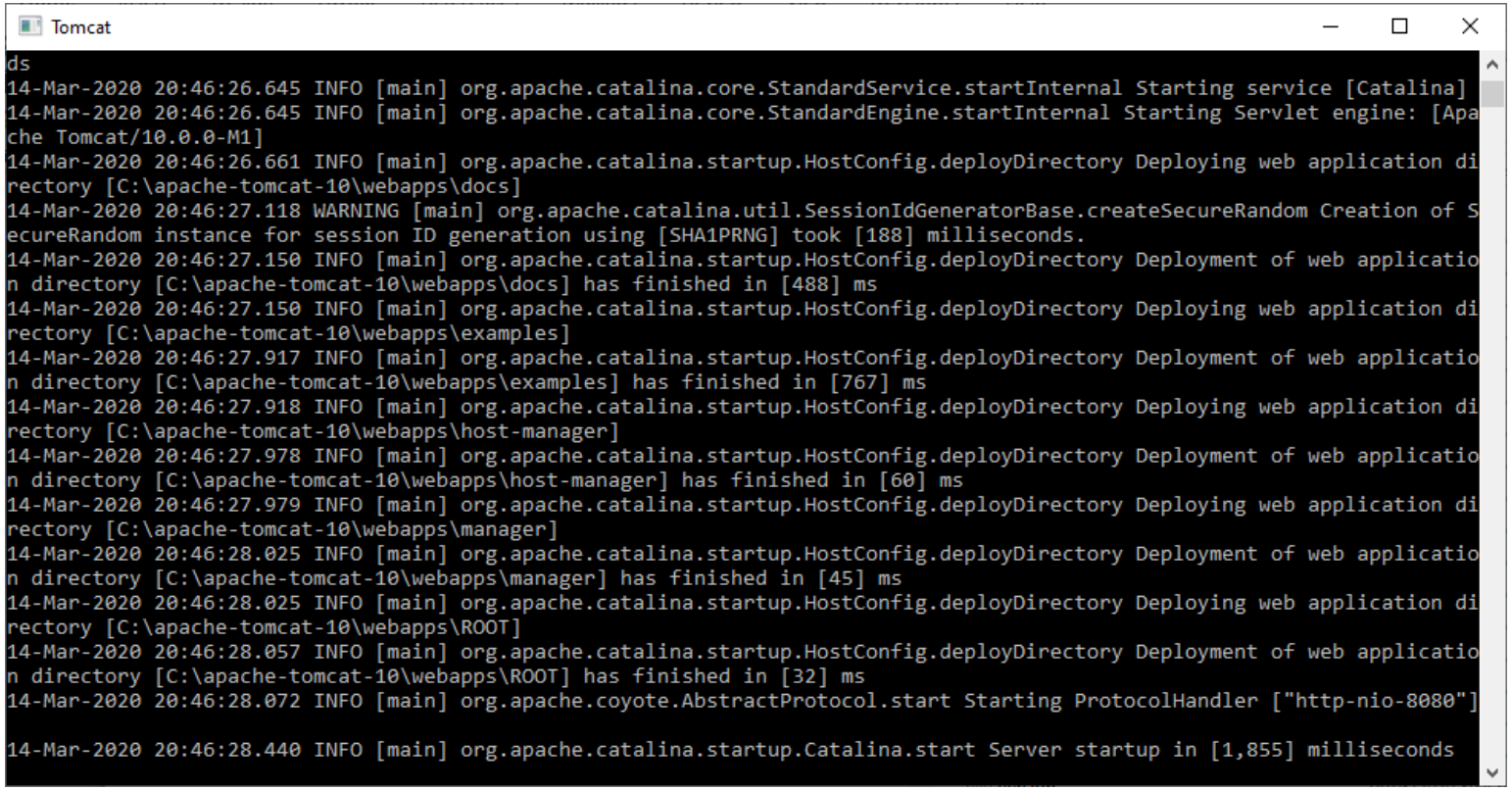
Setting JAVA_HOME

- The best way to set the JAVA_HOME variable is to use a plain text editor (such as the file editor in Eclipse) to create a file called “setev.bat”
- Add this file to the “bin” folder of your Tomcat installation
- The content of this text file would be something like (depending on where you have installed Java):

```
set JAVA_HOME=C:\Program Files\Java\jdk-15
```

The Tomcat Window

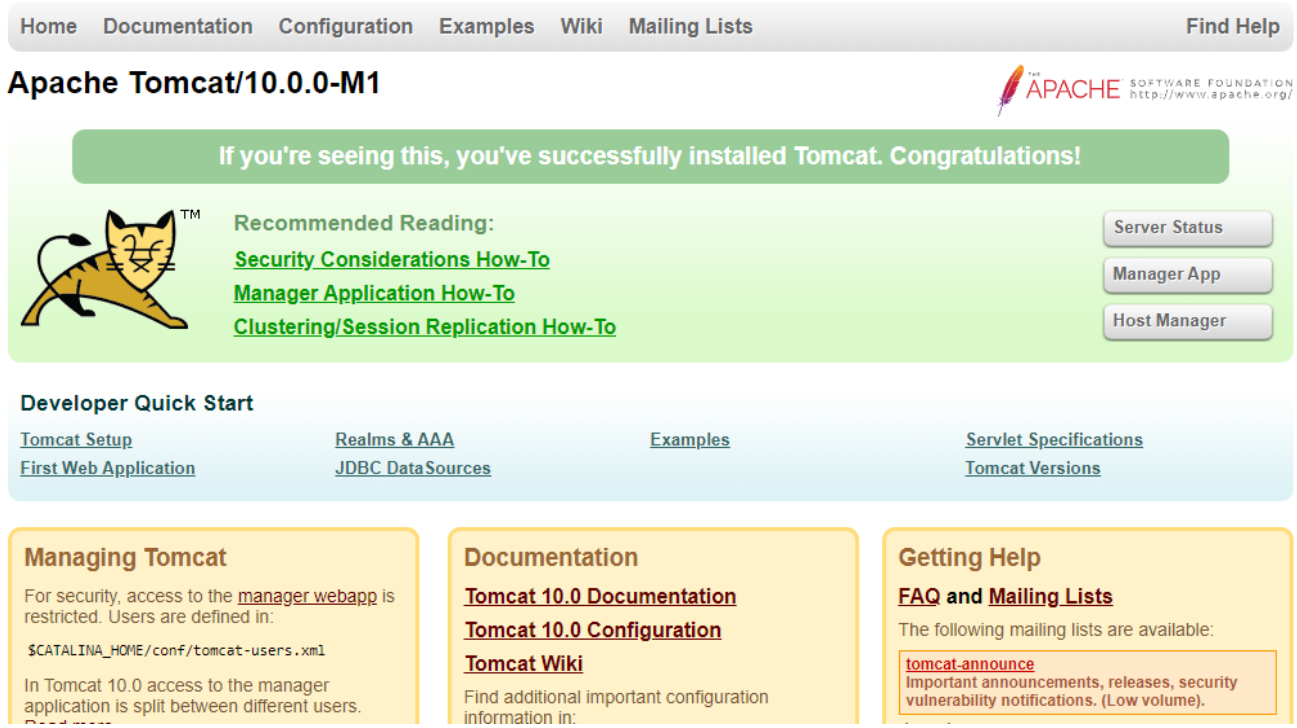
- Once your “setenv.bat” file has been added to the “bin” folder, running the startup script should result in a successful server start
- You should see a separate “Tomcat” window appear, with some log messages
- Do not close this window, as this will stop the server



The ‘localhost’ URL and port number

- When we run a test server on the local machine, we use the loopback address
  - IP address 127.0.0.1, which is also known as ‘localhost’
- The HTTP server that is included with Tomcat runs by default on port 8080

  http://localhost:8080
- To check that Tomcat is running correctly, open a Web browser and direct it to this URL



Web Application Structure and Deployment

- Jakarta EE specifies some folders and files that are used to deploy Java web applications
- The static contact (e.g. HTML files) can be put into the root of the folder structure
- A Java web application may include a deployment descriptor
  - An XML file used to configure how the server deploys the application
  - Called ‘web.xml’, put into a special folder called ‘WEB-INF’
- Jakarta EE application components may be packaged in JAR files
  - a JAR file for a web application archive is given a ‘.war’ extension

XML Deployment Descriptors

- The ‘web.xml’ deployment descriptor must be both well-formed and valid XML so that its elements can be processed by the application server.
- The following example uses an XML Schema for validation

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  "http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
 ...
</web-app>
```

- The root element is called ‘web-app’
  - Inside this element are many optional nested elements

Welcome File

- A useful element to add is the “welcome-file-list”.
  - Configures the default web page
- The “welcome-file-list” must contain at least one “welcome-file” entry
- In our example, the default response will be the “welcome.html” page

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <welcome-file-list>
    <welcome-file>welcome.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Deploying the WAR file with Ant

- Much of the build file used in this example uses tasks that should be familiar from Chapter 14
- First, a series of properties are set for various folders and file names, including the deployment folder for Tomcat web applications

```
<project name="webap" default="copy-war" basedir=".">
  <property name="deployfolder" value="c:\webapp" />
  <property name="sourcefolder" value="${deployfolder}\sources" />
<property name="configfolder"  value="${deployfolder}\config" />
<property name="webapp" value="${deployfolder}\foundjava.war" />
<property name="tomcat-deploy" value="C:\apache-tomcat-10\webapps" />
```

Properties

- deployfolder
  - The root folder for the various deployment files, which will also be used as the build destination folder for the web archive
- source
  - This folder will contain files that need to be in the root folder of the web application, in this case the “welcome.html” file
- configfolder
  - This folder will contain the “web.xml” file
- webapp
  - This specifies the file name of the web application to be built (“foundja-va.war”)
- tomcat-deploy
  - The deployment folder for the Tomcat web server (this is called “webapps” in the Tomcat installation)

Ant Targets

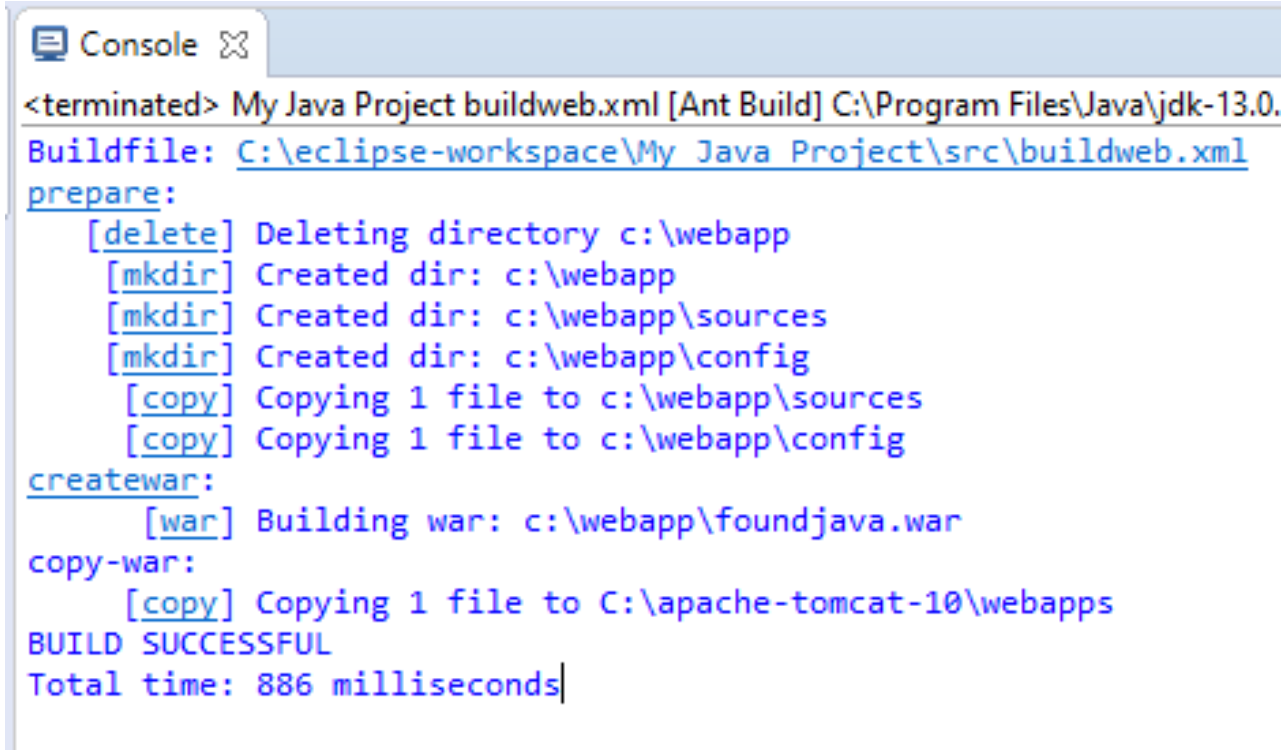
- The ‘prepare’ target creates the required folders and copies files into them from the Eclipse source folders
- The “createwar” target uses the “war” task to create a web archive
  - The “destfile” attribute is the web application to be created, and the “webxml” attribute is the location of the “web.xml” file
  - All the files specified in the “fileset” will also be added to the archive.

```
<target name="createwar" depends="prepare">
  <war destfile="${webapp}" webxml="${configfolder}/web.xml" >
    <fileset dir="${sourcefolder}"/>
  </war>
</target>
```

- The ‘copy-war’ target copies the war file to the web server’s deployment folder

Running the Build File

- The WAR file is built, and then it is deployed to the server
- When the war is copied to Tomcat, the Web application is rebuilt and redeployed



The Deployed Application

- The application will be deployed using the name of the WAR file as the name of the web application

```
http://localhost:8080/foundjava
```

- Because ‘welcome.html’ is the default welcome file, this page should appear without needing to be specifically requested
- We could alternatively invoke it directly

```
http://localhost:8080/welcome.html
```

Exercise 20.1

- Download and install Tomcat
- Add a “setenv.bat” file to the “bin” folder of the Tomcat installation that sets the value of the JAVA_HOME environment variable
- Check that you can start the server from the Command Prompt and view the Tomcat home page by opening a browser on http://localhost:8080

Exercise 20.2

- Create a short HTML file and an XML deployment descriptor in the “src” folder of your Eclipse project
- Create an Ant build file to deploy these resources in a “war” file to the “webapps” folder of your server
- Check that the web app deploys successfully
- Connect to the web page using the name of your web app

The HttpClient Class

- The HttpClient can make a connection to a server either synchronously (blocking) or asynchronously (unblocking)
- These examples use asynchronous connections
- The code structure is like that of the collection streams that we looked at in Chapter 12, again based on underlying lambda expressions.
- In the follwing example of using the HttpClient to make an asynchronous connection there are two parts to the code
- creating an HTTP request to send to the server
- receiving the HTTP response and doing something with it

How Applets Differ from Applications

- The first step is to create an HTTPClient object using the static “newHttpClient” method of the HttpClient class

```
HttpClient client = HttpClient.newHttpClient();
```

- Creating the request is done via the HttpClient.Builder interface with the “uri” and “build” methods of the builder being chained together to create the HttpRequest

```
HttpRequest request = HttpRequest.newBuilder()
  .uri(URI.create("http://localhost:8080/foundjava/"))
  .build();
```

Sending the Request ("sendAsync")

- The BodyHandlers interface can create different types of BodyHandler from its methods
  - The "ofString" method creates a BodyHandler that will handle the body of the response as a String
- The "sendAsync" method returns a CompletableFuture" object, to which the "thenApply", "thenAccept" and "join" methods are chained
  - The 'thenAccept" method executes when the body is available, and in this case sends it to the "println" method of "System.out"

```
client.sendAsync(request, BodyHandlers.ofString())
  .thenApply(HttpResponse::body)
  .thenAccept(System.out::println)
  .join();
```

"getHttpString" Method

- This example method encapsulates HttpClient code that can read the response from a URL and return it as a String

```
public static String getHttpString(String uri)
{
    StringBuffer htmlText = new StringBuffer();
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create(uri))
      .build();
    client.sendAsync(request, BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenAccept(htmlText::append)
      .join();
    return htmlText.toString();
}
```

Getting a Response from Tomcat

- The following class tests the method by passing in the URL of the web app running on Tomcat.
- However, any URL can be passed into the “getHttpString” method.

```
public class HttpSourceReaderRunner {
  public static void main(String[] args) {
    String htmlString = HttpSourceReader.getHttpString("http://localhost:8080/foundjava/");
    System.out.println(htmlString);
  }
}
```

- Connecting to the example web application running on Tomcat, the output from the program is simply the source of the HTML page.

The Swing HTMLEditorKit

- One way of using an HTML file that has been reads from a server is to display it in a client-side application using an HTMLEditorKit
- Here, an HTMLEditorKit is added to a JEditorPane
  - a document suitable for HTML is added
- The JEditorPane is put into a BorderLayout and a scroll-bar is added

```
JEditorPane htmlPane = new JEditorPane();
HTMLEditorKit kit = new HTMLEditorKit();
htmlPane.setEditorKit(kit);
Document document = kit.createDefaultDocument();
htmlPane.setDocument(document);
htmlPane.setLayout(new BorderLayout());
getContentPane().add(htmlPane, BorderLayout.CENTER);
getContentPane().add(new JScrollPane(htmlPane));
```

Reading HTML into a Swing HTMLEditorKit

- The “getHttpString” method reads an HTML page from the server
- This can be added to the pane
- It will then be displayed and is able to be edited
  - editing can be disabled by using “setEditable(false)”

```
// read some HTML from a URL
String htmlString = HttpSourceReader.getHttpString ("http://localhost:8080/foundjava/");
htmlPane.setText(htmlString);
// uncomment this line to make the text read-only
// htmlPane.setEditable(false);
```

RESTful Web Services

- The web also acts as the platform for many services other than HTML pages
- These services come in many forms and use a range of protocols, but a common approach is to use RESTful web services
- Representational state transfer (REST) is not a specific protocol, rather it is an architectural style of services used over HTTP
- A common way of implementing these services is to provide them using JavaScript Object Notation (JSON)
  - text-based documents that structure data using attribute-value pairs and arrays
- A web service will provide an API that can be used to assemble HTTP requests for making queries against the service, returning a JSON response.

JSON Weather Services

- A very common type of web service is a weather service, where the service can be queried for a specific set of weather information based on various parameters
- The next example will use the “OpenWeather” service, which provides free access to weather data through several different APIs
- The one we will use in the next example is the JSON version of the “current weather data” API, which provides current weather data for one location.
- To use the API you will have to first set up an account to access an API key that you can use with calls to the service.

OpenWeather API Calls

- API calls to the server have simple formats, such as this one that requests the weather using the name of a city

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid={api key}
```

- e.g. to query the weather in London
  - adding the “units” property to get the temperature in Centigrade (the default is Kelvin)

```
http://api.openweathermap.org/data/2.5/weather?q=London&units=metric&appid=...
```

OpenWeather API Response

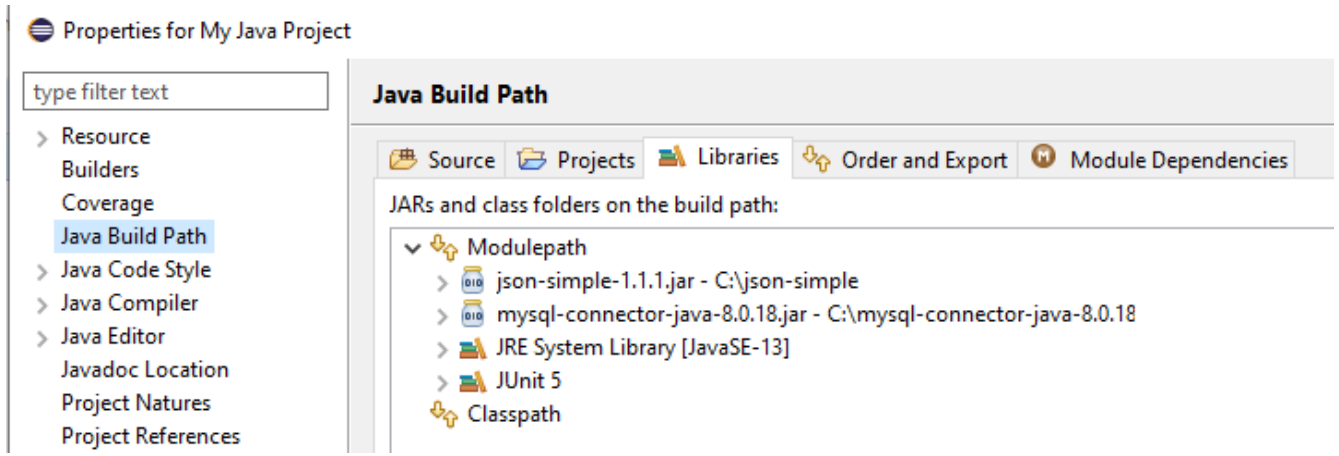
- If you put this API call into the address bar of a browser, the returned data would look something like this


```
{"coord":{"lon":-0.13,"lat":51.51},"weather":[{"id":801,"main":"Clouds","description":"few
clouds","icon":"02d"}],"base":"stations","main":
{"temp":7.06,"feels_like":5.03,"temp_min":3.89,"temp_max":10.56,"pressure":1030,"humidity":70}
,"visibility":10000,"wind":{"speed":0.5},"clouds":{"all":24},"dt":1586244659,"sys":
{"type":1,"id":1414,"country":"GB","sunrise":1586236880,"sunset":1586285000},"timezone":3600,"
id":2643743,"name":"London","cod":200}
```


- The curly braces contain sets of attribute-value pairs, which may be nested inside each other
- The square brackets indicate an array
  - "weather" is an array, though in this example it is an array with only one element.

json.simple

- There is no standard Java library for parsing JSON, but there are several third-party tools available
- The one used in this example is json.simple
- Needs the “json.simple.jar” downloaded from:
  - https://cliftonlabs.github.io/json-simple/
- Once the JAR file has been downloaded, add it to the modulepath of your project



Processing JSON Data

- The json-simple classes can parse JSON data from the weather service
- The next example uses the following classes from json-simple:

```
import com.github.cliftonlabs.json_simple.JsonArray;
import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.Jsoner;;
```

- Jsoner has static methods to serialize JSON data
- Convert the String data from the service into a JsonObject so it can be processed.

```
JsonObject jsonData = (JsonObject)Jsoner.deserialize(jsonString.toString());
```

Accessing JSON Object Values

- Access values in a JSON object using the attribute name as an argument to the “get” method
  - e.g. “name” attribute is the city name

```
System.out.println("\nCity: " + jsonData.get("name"));
```

- Handling array data
  - Retrieve a JsonArray from the main JsonObject
  - Access each JsonObject inside the “weather” array

```
JsonArray jsonWeatherData = (JsonArray) jsonData.get("weather");
JsonObject jsonWeather = null;
// loop through the "weather" array
for(int i=0;i<jsonWeatherData.size();i++)
{
// Get the data from the array elements using the index number
            jsonWeather = (JsonObject)jsonWeatherData.get(i);
// etc.
```

Exercise 20.3

- Using the HttpSourceReader to read JSON data from the OpenWeather web service, create a Swing frame that displays selected parts of the weather data.

Summary

- Setting up a Tomcat web server
- Creating and deploying a web application, containing an HTML page and a “web.xml” deployment descriptor
- Packaging and deploying a “war” archive using an Ant build file
- Connecting to a web server using the HttpClient class to download data to be processed locally on the client
  - Connecting to an HTML page and presenting it in an editable window
  - reading weather data from a JSON web service.