Learning Network Programming with Java

Network-aware applications are becoming more prevalent and play an ever-increasing role in the world today. Connecting and using an Internet-based service is a frequent requirement for many applications. Java provides numerous classes that have evolved over the years to meet evolving network needs. These range from low-level socket- and IP-based approaches to those encapsulated in software services.

This book explores how Java supports networks, starting with the basics and then advancing to more complex topics. An overview of each relevant network technology is presented, followed by detailed examples of how to use Java to support these technologies.

Who this book is written for

Learning Network Programming with Java is oriented toward developers who wish to use network technologies to enhance the utility of their applications. You should have a working knowledge of Java and an interest in learning the latest in network programming techniques using Java. No prior experience with network development or special software beyond the Java SDK is needed. Upon completion of the book, beginners and experienced developers will be able to use Java to access resources across a network and the Internet.

What you will learn from this book

- Connect to other applications using sockets
- Use channels and buffers to enhance communication between applications
- Access network services and develop client/server applications
- Explore the critical elements of peer-to-peer applications and the current technologies available
- Use UDP to perform multicasting
- Address scalability through the use of core and advanced threading techniques
- Incorporate techniques into an application to make it more secure
- Configure and address interoperability issues to enable your applications to work in a heterogeneous environment
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 *Client/Server Development*
- A synopsis of the book’s content
- More information on *Learning Network Programming with Java*
Richard M Reese has worked in both industry and academia. For 17 years, he worked in the telephone and aerospace industries, serving in several capacities, including research and development, software development, supervision, and training. He currently teaches at Tarleton State University, where he has the opportunity to apply his years of industry experience to enhance his teaching.

Richard has written several Java books and a C Pointer book. He uses a concise and easy-to-follow approach to topics at hand. His Java books have addressed EJB 3.1, updates to Java 7 and 8, certification, functional programming, jMonkeyEngine, and natural language processing.
Preface

The world is becoming interconnected on an unprecedented scale with more services being provided on the Internet. Applications ranging from business transactions to embedded applications, such as those found in refrigerators, are connecting to the Internet. With isolated applications no longer being the norm, it is becoming increasingly important for applications to be network enabled.

The goal of this book is to provide the reader with the necessary skills to develop Java applications that connect and work with other applications and services across a network. You will be introduced to a wide range of networking options that are available using Java, which will enable you to develop applications using the appropriate technology for the task at hand.

What this book covers

Chapter 1, Getting Started with Network Programming, introduces the essential network terminology and concepts. The networking support that Java provides is illustrated with brief examples. A simple client/server application is presented along with a threaded version of the server.

Chapter 2, Network Addressing, explains how nodes on a network use addresses. How Java represents these addresses is introduced along with support for IPv4 and IPv6. This chapter also covers how Java can configure various network properties.

Chapter 3, NIO Support for Networking, explains how the NIO package provides support for communication using buffers and channels. These techniques are illustrated with a client/server application. The support that NIO provides for asynchronous communication is also demonstrated.
Chapter 4, Client/Server Development, covers how HTTP is an important and widely-used protocol. Java provides support for this protocol in a variety of ways. These techniques are illustrated along with a demonstration of how cookies are handled in Java.

Chapter 5, Peer-to-Peer Networks, discusses how peer-to-peer networks provide a flexible alternative to the traditional client/server architecture. The basic peer-to-peer concepts are introduced along with demonstrations of how Java supports this architecture. FreePastry is used to illustrate one open source peer-to-peer solution framework.

Chapter 6, UDP and Multicasting, explains how UDP is an alternative to TCP. It provides a less reliable but more efficient way for applications to communicate across the Internet. Java's extensive support for this protocol is demonstrated, including NIO support, and how UDP can support streaming media.

Chapter 7, Network Scalability, explains how, as more demands are placed on a server, systems need to scale to address these demands. Several threading techniques supporting this need are demonstrated, including thread pools, futures, and the NIO's selector.

Chapter 8, Network Security, discusses how applications need to protect against a variety of threats. This is supported in Java using encryption and secure hashing techniques. Symmetric and asymmetric encryption techniques are illustrated. In addition, the use of TLS/SSL is demonstrated.

Chapter 9, Network Interoperability, covers how Java applications may need to exchange information with other applications that are written in different languages. The issues that impact an application's interoperability are examined, including byte order. Communication between different implementations is demonstrated using sockets and middleware.
In this chapter, we will explore the process of developing a client/server application that is primarily oriented around HTTP. This is an important protocol, and it serves as the primary communication medium for a multitude of applications. We will examine the protocol, the requirements placed on a client, and the requirements placed on a server for various versions of the protocol.

Specifically, we will:

- Examine the nature of the HTTP protocol
- Demonstrate how low-level sockets can support the protocol
- Use the `HttpURLConnection` and `HTTPServer` classes to create an HTTP server
- Examine various open source Java HTTP servers
- Investigate various configuration issues and how cookies are handled

HTTP servers are used extensively, so a good understanding of how Java supports them is important.

**The HTTP protocol structure**

HTTP is a network protocol that is used to deliver resources across the World Wide Web (WWW). Resources are usually HyperText Markup Language (HTML) files, but they also include a number of other file types, such as images, audio, and video. Users often enter a URL into a browser to obtain a resource. The term URL stands for Uniform Resource Locator with the emphasis here on resource.
Most people use a browser to communicate across WWW. The browser represents a client application, while the web server responds to client requests. The default port used by these servers is port 80.

HTTP has evolved over the years. HTTP/1.0 originated in the 1980s and 1990s with the first documentation released in 1991. The latest definition of HTTP/1.1 was released as a six-part specification in June 2014. A Request For Comments (RFC) for HTTP 2.0 was released in May 2015. HTTP is an evolving standard.

The following links may prove useful for the interested reader:

<table>
<thead>
<tr>
<th>Version</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.0</td>
<td><a href="http://www.w3.org/Protocols/HTTP/1.0/spec.html">http://www.w3.org/Protocols/HTTP/1.0/spec.html</a></td>
</tr>
<tr>
<td>HTTP/1.1</td>
<td><a href="http://tools.ietf.org/html/rfc2616">http://tools.ietf.org/html/rfc2616</a></td>
</tr>
<tr>
<td>HTTP/2</td>
<td><a href="https://en.wikipedia.org/wiki/HTTP/2">https://en.wikipedia.org/wiki/HTTP/2</a></td>
</tr>
</tbody>
</table>

HTTP servers are used in a variety of situations. The most common use is within organizations to support the dissemination of information to users. Often this is supported by production-quality servers, such as those provided by the Apache Software Foundation (http://www.apache.org/foundation/), or Gemini (http://www.eclipse.org/gemini/).

However, not all servers need to support the level of service typified by production servers. They can be quite small and even embedded in remote devices where they may affect a change in a device instead of only supplying information.

This chapter will examine the various network technologies that are supported by Java to address these types of concerns. These include the following:

- An overview of HTTP protocol syntax
- Low-level socket support for clients/servers
Chapter 4

- Using the URLConnection class
- Using the HTTPServer class
- An overview of open source Java servers

HTTP is a complex topic, and we are only able to skim its surface.

Robots, often called spiders, are applications that automatically follow links, frequently to collect web pages for use by search engines. If you desire to develop such an application, research their use and how they are built (http://www.robotstxt.org/). These types of applications can be disruptive if not designed carefully.

The nature of HTTP messages

Let’s examine the format of an HTTP message. Messages are either a request message sent from a client to a server, or a response message sent from a server to a client. Based on an understanding of the format, we will show you how Java supports these messages. HTTP messages are, for the most part, readable by humans. Both the request and response messages use this structure:

- A line indicating the type of message
- Zero or more header lines
- A blank line
- An optional message body containing data

The following is an example of an HTTP request:

GET /index HTTP/1.0

User-Agent: Mozilla/5.0

A client request message consists of an initial request line and zero or more header lines. A response message consists of an initial response line (called the status line), zero or more header lines, and an optional message body.

Let’s examine these elements in more detail.
Initial request line format

The formats of the request and response initial lines differ. The request line consists of three parts separated by spaces:

- Request method name
- Local path of the resource
- The HTTP version

The method names refer to the action requested by the client. The most common method used is the GET method, which simply requests that a specific resource be returned. The POST command is also common and is used to insert and update data. A list of HTTP/1.0 methods names is found at http://www.w3.org/Protocols/HTTP/1.0/spec.html#Methods. HTTP/1.1 method names can be found at http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html. Method names are always written in uppercase.

The local path typically references the resource desired. It follows the hostname in the URL request. For example, in the following URL, the local path is /books/info/packt/faq/index.html:

www.packtpub.com/books/info/packt/faq/index.html

The HTTP version is always in uppercase and consists of the acronym, HTTP, followed by a forward slash, and then the version number:

HTTP/x.x

The following is an example of a request initial line:

GET /index HTTP/1.0

The response initial line consists of three parts separated by spaces, as follows:

- The HTTP version
- A response status code
- A response phrase describing the code

The following line is an example of a response initial line. The response code reflects the status of the result and is easily interpreted by a computer. The reason phrase is meant to be human readable.

HTTP/1.0 404 Not Found

The HTTP version uses the same format that is used for the request line.
Chapter 4

The following table contains a list of more commonly used codes. A complete list can be found at https://en.wikipedia.org/wiki/List_of_HTTP_status_codes:

<table>
<thead>
<tr>
<th>Status code</th>
<th>Standard text</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>This indicates that the request was a success</td>
</tr>
<tr>
<td>301</td>
<td>Moved Permanently</td>
<td>This indicates that the URL has been moved permanently and the link should be updated</td>
</tr>
<tr>
<td>302</td>
<td>Found</td>
<td>This indicates that the resource is temporarily located somewhere else, but the URL should still be used</td>
</tr>
<tr>
<td>307</td>
<td>Temporary Redirect</td>
<td>This is similar to 302, but the method used should not be changed, which may happen with 302</td>
</tr>
<tr>
<td>308</td>
<td>Permanent Redirect</td>
<td>This is similar to 301, but the method used should not be changed, which may happen with 301</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>This indicates that request access was incorrect</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td>This indicates that the resource is restricted often because the login attempt failed</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
<td>This indicates that access to the requested resource is forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
<td>This indicates that the resource is no longer available</td>
</tr>
<tr>
<td>500</td>
<td>Internal server error</td>
<td>This reflects some sort of error with the server</td>
</tr>
<tr>
<td>502</td>
<td>Bad Gateway</td>
<td>This indicates that the gateway server received an invalid response from another server</td>
</tr>
<tr>
<td>503</td>
<td>Service Unavailable</td>
<td>This indicates that the server is not available</td>
</tr>
<tr>
<td>504</td>
<td>Gateway Timeout</td>
<td>This indicates that the gateway server did not receive a response from another server in a timely manner</td>
</tr>
</tbody>
</table>

The status code is a three-digit number. The first digit of this number reflects the category of the code:

- 1xx: This represents an informational message
- 2xx: This represents a success
- 3xx: This redirects the client to another URL
- 4xx: This represents a client error
- 5xx: This represents a server error
Header lines

Headers lines provide information regarding the request or response, such as the e-mail address of the sender, and an application identifier. The header consists of a single line. The format of this line starts with the header identifier, followed by a colon, spaces, and then the value assigned to the header. The following header illustrates the User-Agent header that is used by Firefox 36.0. This header identifies the application as the Firefox browser running on a Windows platform:

**User-Agent: Mozilla/5.0 (Windows NT 6.3; rv:36.0) Gecko/20100101 Firefox/36.0**


HTTP 1.0 defines 16 headers (http://www.w3.org/Protocols/HTTP/1.0/spec.html#HeaderFields), while HTTP 1.1 has 47 headers (http://tools.ietf.org/html/rfc2616#section-14). Its Host header is required.

Headers are useful in helping troubleshoot problems when they occur. It is a good idea to include the From and User-Agent headers for requests so that the server can be in a better position to respond to the request.

Message body

This is the data constituting the message. While normally a message body is included, it is optional and is not needed for some messages. When a body is included, the Content-Type and Content-Length headers are included to provide more information about the body.

For example, the following headers can be used for a message body:

**Content-type: text/html**

**Content-length: 105**

The message body may appear as follows:

```html
<html><h1>HTTPServer Home Page....</h1><br><b>Welcome to the new and improved web server!</b><BR></html>
```
Client/Server interaction example
The following interaction is a simple demonstration of a client sending a request and the server responding. The client request message uses the GET method against a path of \\index:

GET /index HTTP/1.0
User-Agent: Mozilla/5.0

The server will respond with the following message, assuming that it was able to process the request. The Server, Content-Type, and Content-Length headers are used. A blank line separates the headers and the HTML message body:

HTTP/1.0 200 OK
Server: WebServer
Content-Type: text/html
Content-Length: 86

<html><h1>WebServer Home Page....</h1><br><b>Welcome to my web server!</b><br><BR></html>

Other headers lines can be included.

Java socket support for HTTP client/server applications
An HTTP client will make a connection to an HTTP server. The client will send a request message to the server. The server will send back a response message, frequently, as an HTML document. In the early HTTP version, once the response was sent, the server would terminate the connection. This is sometimes referred to as a stateless protocol because the connection is not maintained.

With HTTP/1.1, persistent connections can be maintained. This improves the performance by eliminating the need to open and close connections when multiple pieces of data need to be transferred between the server and a client.

We will focus on creating an HTTP server and an HTTP client. While browsers typically serve as HTTP clients, other applications can also access web servers. In addition, it helps illustrate the nature of HTTP requests. Our server will support a subset of the HTTP/1.0 specification.
Building a simple HTTP server

We will use a class called WebServer to support the HTTP/1.0 protocol. The server will use a ClientHandler class to handle a client. The server will be limited to handling only GET requests. However, this will be adequate to illustrate the basic server elements needed. Support of other methods can be easily added.

The WebServer definition is shown next. The ServerSocket class is the foundation of the server. Its accept method will block until a request is made. When this happens, a new thread based on the ClientHandler class will be started:

```java
public class WebServer {

    public WebServer() {
        System.out.println("Webserver Started");
        try (ServerSocket serverSocket = new ServerSocket(80)) {
            while (true) {
                System.out.println("Waiting for client request");
                Socket remote = serverSocket.accept();
                System.out.println("Connection made");
                new Thread(new ClientHandler(remote)).start();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new WebServer();
    }
}
```

Mac users may encounter an error when using port 80. Use port 3000 or 8080 instead. Threads are concurrently executing sequences of code within a process. In Java, a thread is created using the Thread class. The constructor's argument is an object that implements the Runnable interface. This interface consists of a single method: run. When the thread is started using the start method, a separate program stack is created for the new thread, and the run method executes on this stack. When the run method terminates, the thread terminates. The ClientHandler class, shown next, implements the Runnable interface. Its constructor is passed to the socket representing the client. When the thread starts, the run method executes. The method displays, starting and terminating messages. The actual work is performed in the handleRequest method:

```java
public class ClientHandler implements Runnable {

    private final Socket socket;

    public void run() {
        System.out.println("Starting ClientHandler");
        try {
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8");
            String request = in.readLine();
            System.out.println("Received request: "+ request);
            String response = "HTTP/1.0 200 OK
Content-Type: text/html

<html>
<head>
<title>Simple Server Response</title>
</head>
<body>
Hello, World!
</body>
</html>";
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            out.println(response);
            out.close();
            System.out.println("Sent response");
        } catch (IOException e) {
            System.err.println("Error sending request: "+ e.getMessage());
        }
    }
}
```
public ClientHandler(Socket socket) {
    this.socket = socket;
}

@Override
public void run() {
    System.out.println("ClientHandler Started for " +
    this.socket);
    handleRequest(this.socket);
    System.out.println("ClientHandler Terminated for " +
    this.socket + ":n");
}

The handleRequest method uses the input and output streams to communicate with
the server. In addition, it determines what request was made and then processes that
request.

In the code that follows, the input and output streams are created and the first line
of the request is read. The StringTokenizer class is used to token this line. When
the nextToken method is invoked, it returns the first word of the line, which should
correspond to an HTTP method:

    public void handleRequest(Socket socket) {
        try (BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));) {
            String headerLine = in.readLine();
            StringTokenizer tokenizer =
                new StringTokenizer(headerLine);
            String httpMethod = tokenizer.nextToken();
            ...
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

A tokenizer is a process that splits text into a series of tokens. Frequently, these
tokens are simple words. The StringTokenizer class's constructor is passed the text
to be tokenized. The nextToken method will return the next available token.
The next code sequence handles the GET method. A message is displayed on the server side to indicate that a GET method is being processed. This server will return a simple HTML page. The page is built using the StringBuilder class where the append methods are used in a fluent style. The sendResponse method is then invoked to actually send the response. If some other method was requested, then a 405 status code is returned:

```java
if (httpMethod.equals("GET")) {
    System.out.println("Get method processed");
    String httpQueryString = tokenizer.nextToken();
    StringBuilder responseBuffer = new StringBuilder();
    responseBuffer
        .append("<html><h1>WebServer Home Page.... </h1><br>
        .append("<b>Welcome to my web server!</b><BR>
        .append("</html>");
    sendResponse(socket, 200, responseBuffer.toString());
} else {
    System.out.println("The HTTP method is not recognized");
    sendResponse(socket, 405, "Method Not Allowed");
}
```

If we wanted to handle other methods, then a series of else-if clauses can be added. To further process the GET method, we will need to parse the remainder of the initial request line. The following statement will give us a string that we can process:

```java
String httpQueryString = tokenizer.nextToken();
```

The previous statement is not needed for this example and should not be included in the code. It simply offers one possible way of further processing HTTP queries.

Once we have created a response, we will use the sendResponse method to send it to the client as shown next. This method is passed the socket, a status code, and the response string. An output stream is then created:

```java
public void sendResponse(Socket socket, int statusCode, String responseString) {
    String statusLine;
    String serverHeader = "Server: WebServer\r\n";
    String contentTypeHeader = "Content-Type: text/html\r\n";

    try (DataOutputStream out =
        new DataOutputStream(socket.getOutputStream())) {
        ... out.close();
    } catch (IOException ex) {
```
If the status code is 200, then a simple HTML page is returned. If the status code is 405, then a single status code line is returned. Otherwise, a 404 response is sent. As we used the DataOutputStream class to write, we use its writeBytes method to handle strings:

```java
if (statusCode == 200) {
    statusLine = "HTTP/1.0 200 OK" + \r\n;
    String contentLengthHeader = "Content-Length: " + responseString.length() + \r\n;
    out.writeBytes(statusLine);
    out.writeBytes(serverHeader);
    out.writeBytes(contentTypeHeader);
    out.writeBytes(contentLengthHeader);
    out.writeBytes("\r\n");
    out.writeBytes(responseString);
} else if (statusCode == 405) {
    statusLine = "HTTP/1.0 405 Method Not Allowed" + \r\n;
    out.writeBytes(statusLine);
    out.writeBytes("\r\n");
} else {
    statusLine = "HTTP/1.0 404 Not Found" + \r\n;
    out.writeBytes(statusLine);
    out.writeBytes("\r\n");
}
```

When the server starts, it will display the following:

**Connection made**

**Waiting for client request**

When a client makes a GET request, output similar to the following one will be displayed:

**ClientHandler Started for Socket[addr=/127.0.0.1,port=50573,localport=80]**

**Get method processed**

**ClientHandler Terminated for Socket[addr=/127.0.0.1,port=50573,localport=80]**

With a simple server in place, let's take a look at how we can build an HTTP client application.
Building a simple HTTP client

We will use the following `HTTPClient` class to access our HTTP server. In its constructor, a socket connecting to the server is created. The `Socket` class's `getInputStream` and `getOutputStream` return input and output streams for the socket, respectively. The `sendGet` method is called, which sends a request to the server. The `getResponse` method returns the response, which is then displayed:

```java
public class HTTPClient {

    public HTTPClient() {
        System.out.println("HTTP Client Started");
        try {
            InetAddress serverInetAddress =
                InetAddress.getByName("127.0.0.1");
            Socket connection = new Socket(serverInetAddress, 80);

            try (OutputStream out = connection.getOutputStream();
                 BufferedReader in =
                     new BufferedReader(new
                         InputStreamReader(
                             connection.getInputStream()))) {
                sendGet(out);
                System.out.println(getResponse(in));
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    ... 

    public static void main(String[] args) {
        new HTTPClient();
    }
}

The `sendGet` method follows this, which sends a GET method request using a simple path. This is followed by a User-Agent header. We used an instance of the `OutputStream` class with the `write` method. The `write` method requires an array of bytes. The `getBytes` method returns this array of bytes:

```java
private void sendGet(OutputStream out) {
    try {
        out.write("GET /default\r\n".getBytes());
```

```
out.write("User-Agent: Mozilla/5.0\r\n".getBytes());
} catch (IOException ex) {
    ex.printStackTrace();
}
}

The `getResponse` method is as follows and is passed a `BufferedReader` instance to get the response from the server. It returns a string created using the `StringBuilder` class:

```java
private String getResponse(BufferedReader in) {
    try {
        String inputLine;
        StringBuilder response = new StringBuilder();
        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine).append("\n");
        }
        return response.toString();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return "";
}
```

When the client application executes, we get the following output reflecting the server's response:

HTTP Client Started

HTTP/1.0 200 OK

Server: WebServer

Content-Type: text/html

Content-Length: 86

<h1>WebServer Home Page.... </h1><br/><b>Welcome to my web server!</b><br/><BR>
If we use the same request from a browser, we will get the following:

```
127.0.0.1
```

**WebServer Home Page....**

*Welcome to my web server!*

These client and server applications can be further enhanced. However, we can use the `HttpURLConnection` class to achieve similar results.

### Client/server development using standard Java classes

Specifically, we will use the `HttpURLConnection` and `HTTPServer` classes to implement a client and server application. These classes support much of the functionality required for clients and servers. Using these classes will avoid writing low-level code to implement HTTP functionality. Low-level code refers to the non-specialized classes, such as the `Socket` class. Higher-level and more specialized classes, such as the `HttpURLConnection` and `HTTPServer` classes, supplement and provide additional support for specialized operations.

The `HttpURLConnection` class is derived from the `HttpConnection` class. This base class has a number of methods that not are directly concerned with the HTTP protocol.

### Using the HttpURLConnection class

The `HttpURLConnection` class provides a convenient technique to access a web server. With this class, we can connect to a site, make a request, and access the respond headers and the response message.

We will use the `HttpURLConnectionExample` class that is defined as follows. A `sendGet` method supports transmitting the GET method request to the server. The `HttpURLConnectionExample` class supports other HTTP methods. For this example, we are only using the GET method:

```java
public class HttpURLConnectionExample {
    public static void main(String[] args) throws Exception {
        // ...
    }
```
The `sendGet` method implementation is shown next. A Google query (http://
www.google.com/search?q=java+sdk&ie=utf-8&oe=utf-8) is used to illustrate
the process where we search for "java sdk". The latter part of the query, &ie=utf-
8&oe=utf-8, is additional information attached to the query by the Google search
engine. The `openConnection` method will connect to the Google server:

```java
private void sendGet() throws Exception {
    String query =
        "http://www.google.com/search?q=java+sdk&ie=utf-8&oe=utf-8";
    URL url = new URL(query);
    HttpURLConnection connection =
        (HttpURLConnection) url.openConnection();
...
}
```

Using this connection, the `setRequestMethod` and `setRequestProperty` methods
set the request method and user agent, respectively:

```java
connection.setRequestMethod("GET");
connection.setRequestProperty("User-Agent",
    "Mozilla/5.0");
```

The response code is retrieved, and if we are successful, the `getResponse` method
will retrieve the response and then display it as follows:

```java
int responseCode = connection.getResponseCode();
System.out.println("Response Code: " + responseCode);
if (responseCode == 200) {
    String response = getResponse(connection);
    System.out.println("response: " +
        response.toString());
} else {
    System.out.println("Bad Response Code: " +
        responseCode);
}
```
Client/Server Development

The `getResponse` method is shown next. The `HttpURLConnection` class's `getInputStream` method returns an input stream, which is used to create an instance of the `BufferedReader` class. A `StringBuilder` instance is used along with this reader to create and return a string:

```java
private String getResponse(HttpURLConnection connection) {
    try (BufferedReader br = new BufferedReader(
        new InputStreamReader(
            connection.getInputStream()));) {
        String inputLine;
        StringBuilder response = new StringBuilder();
        while ((inputLine = br.readLine()) != null) {
            response.append(inputLine);
        }
        br.close();
        return response.toString();
    } catch (IOException ex) {
        // Handle exceptions
    }
    return "";
}
```

When this program executes, you will get output as follows. Due to the length of the output, it has been truncated:

**Sent Http GET request**

**Response Code: 200**

response: `<!doctype html><html itemscope="" ...`
If we used this query in a browser, we will get output similar to the following:


**URL encoding**

When a URL is formed, a specific URL format needs to be used. Some of the characters of this format are reserved and others are unreserved. Reserved characters have special meaning, such as the forward slash, which is used to separate parts of a URL. Unreserved characters do not have any special meaning.
When a reserved character needs to be used in a non-reserved context, URL encoding, also known as percent-encoding, is used to represent these characters using special character sequences. More information about this process can be found at https://en.wikipedia.org/wiki/Percent-encoding.

In Java, we can perform URL encoding using the URLEncoder class. Specifically, the URLEncoder class has an encode method to convert a string that complies with the application/x-www-form-urlencoded MIME format.

This method is overloaded. The single argument method has been deprecated. The two-argument method accepts a string to be converted and a string that specifies the character encoding scheme. For HTTP messages, use the UTF-8 format.

Previously, we used the following string to create a new URL instance:

```java
String query =
    "http://www.google.com/search?q=java+sdk&ie=utf-8&oe=utf-8";
```

This string was actually formatted by the browser. Instead of using the browser, the following code illustrates how to use the encode method to achieve similar results:

```java
String urlQuery = "http://www.google.com/search?q=";
String userQuery = "java sdk";
String urlEncoded = urlQuery + URLEncoder.encode(
    userQuery, "UTF-8");
```

This will produce the string: http://www.google.com/search?q=java+sd. You can see how the blanks have been converted to + symbols for this URL. The latter part of the original query, &ie=utf-8&oe=utf-8, is not included in our URL encoded string.

The URLDecoder class is available to decode URL encoded strings if necessary. For a comprehensive discussion of URL encoding, see: http://blog.lunatech.com/2009/02/03/what-every-web-developer-must-know-about-url-encoding.

### Using the HTTPServer class

The HTTPServer class is found in the com.sun.net.httpserver package. It provides a powerful set of features to support a simple HTTP server. Many of the tasks that we had to perform manually with our previous servers are simplified with this server. The interaction between a client and server is referred to as an exchange.

This and other supporting classes and interfaces are members of the com.sun.net.httpserver package. They are normally included with most IDEs. The API documentation can be found at http://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/index.html?com/sun/net/httpserver/package-summary.html.
This package consists of a number of classes. The primary classes that we will use include:

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HttpServer</td>
<td>This class supports the basic functionality of an HTTP server</td>
</tr>
<tr>
<td>HttpExchange</td>
<td>This class encapsulates the request and response associated with a client/server exchange</td>
</tr>
<tr>
<td>HttpHandler</td>
<td>This class defines a handle method used to process specific exchanges</td>
</tr>
<tr>
<td>HttpContext</td>
<td>This class maps a URI path to an HttpHandler instance</td>
</tr>
<tr>
<td>Filter</td>
<td>This class supports the preprocessing and post-processing of requests</td>
</tr>
</tbody>
</table>

The server uses an HttpHandler derived class to process client requests. For example, one handler can process requests for basic web pages, while another handler may process service related requests.

The HttpExchange class supports the life-cycle activities of an exchange between a client and a server. It possesses a number of methods providing access to request and response information. These methods are listed in the following table in the order that they are normally used. Not all methods need to be used for all requests:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRequestMethod</td>
<td>This method returns the HTTP method requested</td>
</tr>
<tr>
<td>getRequestHeaders</td>
<td>This method returns the request headers</td>
</tr>
<tr>
<td>getRequestBody</td>
<td>This method returns an InputStream instance for the request body</td>
</tr>
<tr>
<td>getResponseHeaders</td>
<td>This method returns the response headers except for content-length</td>
</tr>
<tr>
<td>sendResponseHeaders</td>
<td>This method sends the response headers</td>
</tr>
<tr>
<td>getResponseBody</td>
<td>This method returns an OutputStream instance used to send the response</td>
</tr>
</tbody>
</table>

An exchange is closed when the input and output streams are closed. The sendResponseHeaders method must be used before the getResponseBody method is invoked.

The performance of the initial version of this class was not very good. However, newer versions have better performance. In addition, the filter facility can assist in processing exchanges.
Using the `com.sun.*` classes can be used without concerns. Problems can occur if the `sun.*` classes are used with different JREs. The `HTTPServer` class fully supports HTTP/1.0, but it only provides partial support for HTTP/1.1.

### Implementing a simple HTTPServer class

The class that follows implements a simple server using the `HTTPServer` class. An instance of the `HttpServer` class is created using the localhost and port 80 (3000 or 8080 on a Mac). The `createContext` method associates the `/index` path with an instance of the `IndexHandler` class. This handler will process the request. The `start` method starts the server. The server will continue running, processing multiple requests until it is manually stopped:

```java
public class MyHTTPServer {
    public static void main(String[] args) throws Exception {
        System.out.println("MyHTTPServer Started");
        HttpServer server = HttpServer.create(
            new InetSocketAddress(80), 0);
        server.createContext("/index", new IndexHandler());
        server.start();
    }
}
```

When the `createContext` method matches the path expressed as a string to a handler, it uses a specific matching process. The details of this process are explained in the *Mapping request URIs to HttpContext paths* section of the `HTTPServer` class documentation found at [http://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/HttpServer.html](http://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/HttpServer.html).

The `IndexHandler` class is declared next. It implements the `HttpHandler` interface by overriding the `handle` method. The `handle` method is passed an `HttpExchange` instance, which we can use to process the request.

In this method, we perform the following actions:

- Display the address of the client
- Send back a request with a status code of 200
- Send the response to the client
The `sendResponseHeaders` method will send an initial response line for status code 200 and a header for the content length. The `getResponseBody` method returns an output stream used to send the message body. The stream is then closed terminating the exchange:

```java
static class IndexHandler implements HttpHandler {
    @Override
    public void handle(HttpExchange exchange) throws IOException {
        System.out.println(exchange.getRemoteAddress());
        String response = getResponse();
        exchange.sendResponseHeaders(200, response.length());
        OutputStream out = exchange.getResponseBody();
        out.write(response.toString().getBytes());
        out.close();
    }
}
```

The `sendResponseHeaders` method uses two parameters. The first is the response code, and the second controls the transmission of the message body, as detailed in the next table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than zero</td>
<td>This is the length of the message. The server must send this number of bytes.</td>
</tr>
<tr>
<td>Zero</td>
<td>This is used for chunked transfer where an arbitrary number of bytes is sent.</td>
</tr>
<tr>
<td>-1</td>
<td>This is when no response body is sent.</td>
</tr>
</tbody>
</table>

The `getResponse` method uses the `StringBuilder` class to construct a string:

```java
public String getResponse() {
    StringBuilder responseBuffer = new StringBuilder();
    responseBuffer
        .append("<html><h1>HTTPServer Home Page.... </h1><br>")
        .append("<b>Welcome to the new and improved web "
            + "server!</b><BR>")
        .append("</html>");
    return responseBuffer.toString();
}
```
When the server is started, the following output is displayed:

**MyHTTPServer Started**

If we enter the URL http://127.0.0.1/index in a browser, the browser will display the page similar to the one in the image in the section *Building a simple HTTP client*.

The server will display the following for each request:

/127.0.0.1:50273

This class is instrumental in processing client requests. Here, we will illustrate several of this class’s methods using a different handler called `DetailHandler`, as declared next:

```java
static class DetailHandler implements HttpHandler {
  @Override
  public void handle(HttpExchange exchange)
  throws IOException {
    ...
  }
}
```

To use this handler, replace the `createContext` method, and call in the `MyHTTPServer` with this statement:

```java
server.createContext("/index", new DetailHandler());
```

Let’s start by examining the use of the `getRequestHeaders` method, which returns an instance of the `Headers` class. This will permit us to display each request header sent by the client and perform additional processing based on the headers if needed.

Add the following code to the `handle` method. The `keySet` method returns a `Set` of key/values pairs for each header. In the for-each statement, the `Set` interface’s `get` method returns a list of values for each header. This list is used to display the headers:

```java
Headers requestHeaders = exchange.getRequestHeaders();
Set<String> keySet = requestHeaders.keySet();
for (String key : keySet) {
  List values = requestHeaders.get(key);
  String header = key + " = " + values.toString() + "\n";
  System.out.print(header);
}
```
Using the previous URL (http://127.0.0.1/index) from the Firefox browser, we get the following output:

Accept-encoding = [gzip, deflate]

Accept = [text/html,application/xhtml+xml,application/xml;q=0.9,*/;q=0.8]

Connection = [keep-alive]

Host = [127.0.0.1]

User-agent = [Mozilla/5.0 (Windows NT 10.0; WOW64; rv:40.0) Gecko/20100101 Firefox/40.0]

Accept-language = [en-US, en; q=0.5]

Cache-control = [max-age=0]

Using a different browser may return a different set of request headers. The `getRequestMethod` method returns the name of the request method, as shown here:

```java
String requestMethod = exchange.getRequestMethod();
```

We can use this to differentiate between client requests.

Some request methods will pass a message body along with the request. The `getRequestBody` method will return a `InputStream` instance to access this body.

The following code illustrates how we can obtain and display the message body:

```java
InputStream in = exchange.getRequestBody();
if (in != null) {
    try (BufferedReader br = new BufferedReader(new InputStreamReader(in));) {
        String inputLine;
        StringBuilder response = new StringBuilder();
        while ((inputLine = br.readLine()) != null) {
            response.append(inputLine);
        }
        br.close();
        System.out.println(inputLine);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
} else {
    System.out.println("Request body is empty");
}
```

As our request did not have a body, nothing is displayed.
Managing response headers

The server can send back response headers using the `sendResponseHeaders` method. However, these headers need to be created using a combination of the `getResponseHeaders` method and the `set` methods.

In the next code sequence, the `getResponseHeaders` method will return an instance of the `Header` class:

```java
Headers responseHeaders = exchange.getResponseHeaders();
```

We use the `getResponse` method to get our response. We will need this to compute the content length. The `set` method is then used to create `Content-Type` and `Server` headers:

```java
String responseMessage = HTTPServerHelper.getResponse();
responseHeaders.set("Content-Type", "text/html");
responseHeaders.set("Server", "MyHTTPServer/1.0");
```

The headers are sent using the `sendResponseHeaders` method described earlier, shown as follows:

```java
exchange.sendResponseHeaders(200, responseMessage.getBytes().length);
```

These response headers can be displayed using the following code sequence. This performs the same functionality as the for-each statement that we used to display the request headers. However, this implementation uses a Java 8 Stream class and two lambda expressions instead:

```java
Set<String> responseHeadersKeySet = responseHeaders.keySet();
responseHeadersKeySet
    .stream()
    .map((key) -> {
        List values = responseHeaders.get(key);
        String header = key + " = " +
            values.toString() + "\n";
        return header;
    })
    .forEach((header) -> {
        System.out.print(header);
    });
```

This implementation uses a stream. The `stream` method returns the keys found in the set. The `map` method processes each key using it to look up a list of values associated with the key. The list is converted into a string. The `forEach` method will then display each of these strings.
The HTTPServer, and its accompanying classes provide a simple, but convenient to use technique to implement an HTTP server. Support is also provided for secure communications using the HttpsServer class, which is discussed in Chapter 8, Network Security.

Open source Java HTTP servers

While we can develop a web server using any of the technologies discussed in this chapter, another option is to use any of a number of open source Java-based HTTP servers. Such servers frequently provide a number of features, including:

- Full compliancy with HTTP standards
- Support for logging and monitoring
- Handling of virtual hosts
- Performance tuning capability
- Scalable
- Chunked data transfer
- Configurability
- Support for NIO (Grizzly)

Leveraging these systems can save you a lot of time and effort that would otherwise be devoted to building a custom server. A partial list of a few Java-based servers include the following:

- Jakarta Tomcat (http://tomcat.apache.org/)
- Jetty (http://www.eclipse.org/jetty/)
- JLHTTP (http://www.freeutils.net/source/jlhttp/)
- GlassFish (https://glassfish.java.net/)
- Grizzly (https://grizzly.java.net/)
- Simple (http://www.simpleframework.org/)

One list of open source Java servers is found at http://java-source.net/open-source/web-servers.

At a higher level, Java EE is frequently used to support Web Servers. While this edition has evolved over the years, servlets form the basis to handle web requests. A servlet is a Java application that hides much of the detail surrounding the low-level processing of requests and responses. This permits the developer to focus on processing requests.
Client/Server Development

Servlets are held in containers that provide support for tasks, such as database access, managing performance, and providing security. A simple servlet is shown next to give you a feel as to how they are structured.

The `doGet` and `doPost` methods handle `GET` and `POST` type messages, respectively. However, as the differences between these two HTTP messages are hidden, only one is needed. The `HttpServletRequest` class represents an HTTP request and the `HttpServletResponse` class represents the response. These classes provide access to the messages. For example, the `getWriter` method returns a `PrintWriter` class instance, which allows us to write the HTML response in a clearer fashion:

```java
public class ServletExample extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>" + "Message to be sent" + "</h1>" + "Message to be sent" + "</h1>");
    }

    @Override
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws IOException, ServletException {
        doGet(request, response);
    }
}
```

Servlets are normally developed using the Java EE SDK. The previous example will not compile correctly unless developed using this API.

Many technologies have evolved and have hidden servlets. Over the years, this has included JavaServer Pages (JSP) and JavaServer Faces (JSF), which have largely eliminated the need to write servlets directly.

There are a number of web servers for Java. A comparison of some of these is found at https://en.wikipedia.org/wiki/Comparison_of_application_servers#Java.
Server configuration

The configuration of a server depends on the technology that was used to build it. Here, we will focus on the configuration of the `URLConnection` class. This class has a number of protected fields that control how the connection behaves. These fields are accessed using corresponding get and set methods.

One field deals with user interactions. When set to `true`, it allows users to engage in interactions, such as responding to an authentication dialog box. A connection can be used for input and/or output. The connection can be configured to disallow input or output.

When data is transferred between a client and a server, it may be cached. The `UseCaches` variable determines whether caches are ignored or not. If set to `true`, then caches are used as appropriate. If `false`, caching is not performed.

The `ifModifiedSince` variable controls whether the retrieval of an object occurs. It is a long value that represents time as the number of milliseconds since the epoch (January 1, 1970, GMT). If the object has been modified more recently than that time, then it is fetched.

The following table summarizes the methods that are used to configure a connection established using the `URLConnection` class. Each of these methods have a corresponding `GET` method:

<table>
<thead>
<tr>
<th>Method</th>
<th>Default</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setAllowUserInteraction</code></td>
<td>NA</td>
<td>This method controls user interaction</td>
</tr>
<tr>
<td><code>setDoInput</code></td>
<td><code>true</code></td>
<td>If its argument is set to <code>true</code>, then input is allowed</td>
</tr>
<tr>
<td><code>setDoInput</code></td>
<td><code>true</code></td>
<td>If its argument is set to <code>true</code>, then output is allowed</td>
</tr>
<tr>
<td><code>setIfModifiedSince</code></td>
<td>NA</td>
<td>This sets the <code>ifModifiedSince</code> variable</td>
</tr>
<tr>
<td><code>setUseCaches</code></td>
<td><code>true</code></td>
<td>This sets the <code>UseCaches</code> variable</td>
</tr>
</tbody>
</table>

More sophisticated servers, such as Tomcat, have many more options to control how it is configured.
When an application is deployed, there are numerous configuration options found in the `deployment.properties` file. Many of these options are low level, and JRE related. A description of the options is found at https://docs.oracle.com/javase/8/docs/technotes/guides/deploy/properties.html. The 21.2.4 Networking section discusses the network options, while the 21.2.5 Cache and Optional Package Repository section is concerned with the configuration of caches.

An **HTTP proxy** is a server that acts as an intermediary between a client and a server. A proxy is frequently used to manage the network, monitor traffic, and improve network performance.

Generally, we are not concerned with the use or configuration of a proxy. However, if a proxy needs to be configured, we can control it either using the JVM command line or within the code using the `System` class's `getProperties` method. We can control the proxy used and specify the user and password to access it if needed. A short discussion of these capabilities is found at http://viralpatel.net/blogs/http-proxy-setting-java-setting-proxy-java/.

**Handling cookies**

A cookie is a string containing a key/value pair representing information of interest to the server such as user preferences. It is sent from a server to a browser. The browser should save the cookie to a file so that it can be used later.

A cookie is a string that consists of a name followed by an equal sign and then a value. The following is one possible cookie:

```java
userID=Cookie Monster
```

A cookie can have multiple values. These values will be separated by a semicolon and white space.

We will use the `HTTPServer` class and the `HttpURLConnection` classes to demonstrate the handling of cookies. In the `MyHTTPServer` class server's handler class's `handle` method, add the following code after the other headers:

```java
    responseHeaders.set("Set-cookie", "userID=Cookie Monster");
```

When the server responds, it will send that cookie.
In the HttpURLConnectionExample class's getResponse method, add the following code at the beginning of its try block. A string is built containing the cookie text. Multiple substring and indexOf methods are used to extract the cookie's name and then its value:

```java
Map<String, List<String>> requestHeaders =
    connection.getHeaderFields();
Set<String> keySet = requestHeaders.keySet();
for (String key : keySet) {
    if ("Set-cookie".equals(key)) {
        List values = requestHeaders.get(key);
        String cookie = key + " = " +
            values.toString() + "\n";
        String cookieName =
            cookie.substring(0, cookie.indexOf("="));
        String cookieValue = cookie.substring(
            cookie.indexOf("=") + 1, cookie.length());
        System.out.println(cookieName + ":" + cookieValue);
    }
}
```

When the server sends a response, it will include the cookie. The client will then receive the cookie. In the server and the client, you should see the following output displaying the cookie:

```
Set-cookie : [userID=Cookie Monster]
```

The previous example handles simple single-value cookies. The code to handle multiple values is left as an exercise for the reader.

**Summary**

In this chapter, we examined the various Java approaches that can be used to develop HTTP client/server applications. Communication using HTTP is a common practice. Understanding how Java supports this process is a valuable skill to possess.

We started with an overview of HTTP messages. We examined the format of the initial request and response lines. Headers lines were also examined, which are used to convey information about the message. An optional message body may appear in an HTTP message. This is more common in a response where the body is often an HTML document.
We demonstrated how a client/server can be developed using simple socket. While possible, this approach requires a lot of work to develop a fully functional HTTP server. This discussion was followed by the use of the `HTTPServer` and `HttpURLConnection` classes to support a server and client, respectively. The use of these classes made the development process much easier.

There are a number of open source Java-based HTTP servers available. These may be viable candidates for some environments. The more complex web servers, typified by Apache Tomcat, were also discussed. They work with servlets and hide much of the low-level HTTP details from the developer.

We wrapped the chapter up with a brief discussion of server configuration issues and how cookies are created and consumed by servers and clients.

While the client/server architecture is very common, the peer-to-peer architecture is an alternative to share information across a network. We will delve into this topic in the next chapter.
Where to buy this book

You can buy Learning Network Programming with Java from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.