CSE 686 Internet Programming

Week 6: JHTTP

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

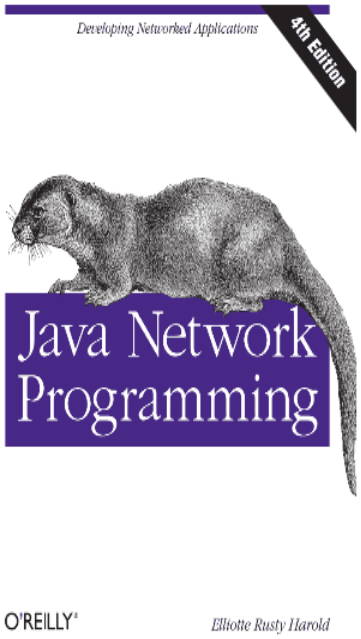
A Full-Fledged HTTP/Web Server

- This example is from our Textbook #2, pp. 319-324.

- You can obtain its source code from the book site:
  http://www.cafeaulait.org/books/jnp4/

Java Network Programming, 4th Edition

[Table of Contents | Examples | Order]



Java's growth over the last 20 years has been nothing short of phenomenal. Given Java's rapid rise to prominence and the even more spectacular growth of the Internet, it's a little surprising that network programming in Java is still so mysterious to so many. It doesn't have to be. In fact, writing network programs in Java is quite simple, as this book will show. Readers with previous experience in network programming in a Unix, Windows, or Macintosh environment should be pleasantly surprised at how much easier it is to write equivalent programs in Java. The Java core API includes well-designed interfaces to most network features. Indeed, there is very little application layer network software you can write in C or C++ that you can't write more easily in Java. Java Network Programming endeavors to show you how to take advantage Java's network class library to quickly and easily write programs that accomplish many common networking tasks. Some of these include:

- Browsing pages on the Web
- Parsing and rendering HTML
- Writing multithreaded servers
- Encrypting communications for confidentiality, authentication, and guaranteed message integrity
- Designing GUI clients for network services
- Posting data to server side programs
- Looking up hosts using DNS
- Downloading files with anonymous FTP
- Connecting sockets for low-level network communication
- Multicasting to all hosts on the network

Java is the first (though no longer the only) language to provide such a powerful cross-platform network library for handling all these diverse tasks. *Java Network Programming* exposes the power and sophistication of this library. This book's goal is to enable you to start using Java as a platform for serious network programming. To do so, this book provides a general background in network fundamentals, as well as detailed discussions of Java's facilities for writing network programs. You'll learn how to write Java programs that share data across the Internet for games, collaboration, software updates, file transfer, and more. You'll also get a behind-the-scenes look at HTTP, UDP, TCP/IP, and the other protocols that support the Internet and the Web. When you finish this book, you'll have the

http://www.cafeaulait.org/books/jnp4/examples/src/ Index of /books/jnp4/exam...

JHTTP.java	05-Oct-2013 17:31	2.0K
JoinDigestUserInterface.java	05-Oct-2013 17:31	869
Last24.java	05-Oct-2013 17:31	1.1K
LastModified.java	05-Oct-2013 17:31	697
LocalPortScanner.java	05-Oct-2013 17:31	498
LocalProxySelector.java	05-Oct-2013 17:31	723
LogFile.java	05-Oct-2013 17:31	486
LoggingDaytimeServer.java	05-Oct-2013 17:31	2.0K
LookupTask.java	05-Oct-2013 17:31	589
LowPortScanner.java	05-Oct-2013 17:31	582
MaxFinderTest.java	05-Oct-2013 17:31	1.3K
MemoryCache.java	05-Oct-2013 17:31	1.5K
MulticastSender.java	05-Oct-2013 17:31	1.1K
MulticastSniffer.java	05-Oct-2013 17:31	1.1K
MultithreadedDaytimeServer.java	05-Oct-2013 17:31	1.2K
MultithreadedMaxFinder.java	05-Oct-2013 17:31	764
MyAddress.java	05-Oct-2013 17:31	312
NoGovernmentCookies.java	05-Oct-2013 17:31	335
NonblockingSingleFileHTTPServer.java	05-Oct-2013 17:31	3.7K
OReillyByName.java	05-Oct-2013 17:31	321
OptionSetExample.java	05-Oct-2013 17:31	340
OptionSupport.java	05-Oct-2013 17:31	756
Options.java	05-Oct-2013 17:31	1.0K
PooledDaytimeServer.java	05-Oct-2013 17:31	1.3K
PooledWeblog.java	05-Oct-2013 17:31	1.3K
ProtocolTester.java	05-Oct-2013 17:31	1.8K
QueryString.java	05-Oct-2013 17:31	793

http://www.cafeaulait.org/books/jnp4/examples/src/ Index of /books/jnp4/exam...

RequestProcessor.java	05-Oct-2013 17:31	4.7K
ReturnDigest.java	05-Oct-2013 17:31	747
ReturnDigestUserInterface.java	05-Oct-2013 17:31	542
ReverseTest.java	05-Oct-2013 17:31	240
SafeBufferedReader.java	05-Oct-2013 17:31	1.3K
SafeBufferedReaderTest.java	05-Oct-2013 17:31	2.0K
SafePrintWriter.java	05-Oct-2013 17:31	4.7K
SecureOrderTaker.java	05-Oct-2013 17:31	3.1K
SecureSourceViewer.java	05-Oct-2013 17:31	947
SenderThread.java	05-Oct-2013 17:31	1.0K
SimpleCacheRequest.java	05-Oct-2013 17:31	417
SimpleCacheResponse.java	05-Oct-2013 17:31	1.1K
SingleFileHTTPServer.java	05-Oct-2013 17:31	3.9K
SocketInfo.java	05-Oct-2013 17:31	733
SourceViewer.java	05-Oct-2013 17:31	933
SourceViewer2.java	05-Oct-2013 17:31	864
SourceViewer3.java	05-Oct-2013 17:31	1.3K
SourceViewer4.java	05-Oct-2013 17:31	871
SpamCheck.java	05-Oct-2013 17:31	846
Time.java	05-Oct-2013 17:31	1.9K
TimeServer.java	05-Oct-2013 17:31	1.4K
UDPDiscardClient.java	05-Oct-2013 17:31	845
UDPDiscardServer.java	05-Oct-2013 17:31	960
UDPDiscardServerWithChannels.java	05-Oct-2013 17:31	886
UDPEchoClient.java	05-Oct-2013 17:31	678
UDPEchoClientWithChannels.java	05-Oct-2013 17:31	2.6K
UDPEchoServer.java	05-Oct-2013 17:31	630

A Full-Fledged HTTP/Web Server

- This program, called JHTTP, shows a full-blown HTTP server that can:
  - Serves an entire document tree, including images, HTML files, text files, and more
  - Handles **GET** requests
- Yet it is still fairly lightweight
- Rather than processing each request as it arrives in the main thread of execution, you'll place incoming connections in a **Thread Pool**.
  - Separate instances of a **RequestProcessor** class will get the connections from the thread pool and process them.

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

    // get the Document root
    File docroot;
    try {
      docroot = new File(args[0]);
    } catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println("Usage: java JHTTP docroot port");
      return;
    }

    // set the port to listen on
    int port;
    try {
      port = Integer.parseInt(args[1]);
      if (port < 0 || port > 65535) port = 80;
    } catch (RuntimeException ex) {
      port = 80;
    }

    try {
      JHTTP webserver = new JHTTP(docroot, port);
      webserver.start();
    } catch (IOException ex) {
      logger.log(Level.SEVERE, "Server could not start", ex);
    }
  }
}
```

The main() method of JHTTP handles initialization, but other programs can also use this class to run basic web servers.

The main() method of the JHTTP class sets the document root directory from **args[0]**.

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

    // get the Document root
    File docroot;
    try {
      docroot = new File(args[0]);
    } catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println("Usage: java JHTTP docroot port");
      return;
    }

    // set the port to listen on
    int port;
    try {
      port = Integer.parseInt(args[1]);
      if (port < 0 || port > 65535) port = 80;
    } catch (RuntimeException ex) {
      port = 80;
    }

    try {
      JHTTP webserver = new JHTTP(docroot, port);
      webserver.start();
    } catch (IOException ex) {
      logger.log(Level.SEVERE, "Server could not start", ex);
    }
  }
}
```

compact1, compact2, compact3

java.io

Class File

java.lang.Object
    java.io.File

All Implemented Interfaces:

Serializable, Comparable<File>

```
public class File
extends Object
implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, "/" for the UNIX root directory, or "\\\\" for a Microsoft Windows UNC pathname, and
2. A sequence of zero or more string *names*.

The first name in an abstract pathname may be a directory name or, in the case of Microsoft Windows UNC pathnames, a hostname. Each subsequent name in an abstract pathname denotes a directory; the last name may denote either a directory or a file. The *empty* abstract pathname has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying

File

```
public File(String pathname)
```

Creates a new `File` instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.

Parameters:

`pathname` - A pathname string

Throws:

`NullPointerException` - If the pathname argument is null

File

```
public File(String parent,
            String child)
```

Creates a new `File` instance from a parent pathname string and a child pathname string.

If `parent` is `null` then the new `File` instance is created as if by invoking the single-argument `File` constructor on the given `child` pathname string.

Otherwise the `parent` pathname string is taken to denote a directory, and the `child` pathname string is taken to denote either a directory or a file. If the `child` pathname string is absolute then it is converted into a relative pathname in a system-dependent way. If `parent` is the empty string then the new `File` instance is created by converting `child` into an abstract pathname and resolving the result against a system-dependent default directory. Otherwise each pathname string is converted into an abstract pathname and the child abstract pathname is resolved against the parent.

Parameters:

`parent` - The parent pathname string

`child` - The child pathname string

Throws:

`NullPointerException` - If child is null

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

    // get the Document root
    File docroot;
    try {
      docroot = new File(args[0]);
    } catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println("Usage: java JHTTP docroot port");
      return;
    }

    // set the port to listen on
    int port;
    try {
      port = Integer.parseInt(args[1]);
      if (port < 0 || port > 65535) port = 80;
    } catch (RuntimeException ex) {
      port = 80;
    }

    try {
      JHTTP webserver = new JHTTP(docroot, port);
      webserver.start();
    } catch (IOException ex) {
      logger.log(Level.SEVERE, "Server could not start", ex);
    }
  }
}
```

The port is read from **args[1]** or 80 is used for a default.

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

    // get the Document root
    File docroot;
    try {
      docroot = new File(args[0]);
    } catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println("Usage: java JHTTP docroot port");
      return;
    }

    // set the port to listen on
    int port;
    try {
      port = Integer.parseInt(args[1]);
      if (port < 0 || port > 65535) port = 80;
    } catch (RuntimeException ex) {
      port = 80;
    }

    try {
      JHTTP webserver = new JHTTP(docroot, port);
      webserver.start();
    } catch (IOException ex) {
      logger.log(Level.SEVERE, "Server could not start", ex);
    }
  }
}
```

Then a new JHTTP object is constructed and started.

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.logging.*;

public class JHTTP {

  private static final Logger logger = Logger.getLogger(
      JHTTP.class.getCanonicalName());
  private static final int NUM_THREADS = 50;
  private static final String INDEX_FILE = "index.html";

  private final File rootDirectory;
  private final int port;

  public JHTTP(File rootDirectory, int port) throws IOException {

    if (!rootDirectory.isDirectory()) {
      throw new IOException(rootDirectory
          + " does not exist as a directory");
    }
    this.rootDirectory = rootDirectory;
    this.port = port;
  }

  public void start() throws IOException {
    ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
    try (ServerSocket server = new ServerSocket(port)) {
      logger.info("Accepting connections on port " + server.getLocalPort());
      logger.info("Document Root: " + rootDirectory);
```

The Logger Class Revisited

- The **Logger** class represents a logging facility that may be local or remote.
  - Through an instance of this class, we can record the various server activities as shown in EchoProtocol.java.
- You may use several loggers in your server, each serving a different purpose and potentially behaving in a different way.
  - For example, you may have separate loggers for operations, security, and error messages.
- In Java each logger is identified by a <u>globally unique name</u>.
- To get an instance of Logger, call the <u>static factory method</u> **Logger.getLogger()** as follows:

```
Logger logger = Logger.getLogger("practical");
```

The Logger Class Revisited

- Each instance of Logger has a current level
  - Each level has an associated integer **value**, so that levels are comparable and can be ordered.
  - Seven system-recognized instances of Level are defined.
  - The built-in levels (defined as static fields of the class Level) are: severe, warning, info, config, fine, finer, and finest.
  - Other user-specific levels can be created, but there is rarely any need to do so.

- Each message logged also has an associated level
  - Messages with levels below the Logger instance's current level are discarded (i.e., not logged).

The Logger Class Revisited

- When you log, where do the messages go?
  - The logger sends messages to one or more **Handlers**, which "handle" publishing the messages.
  - By default, a logger has a single **ConsoleHandler** that prints messages to **System.err**.
  - You can change the handler or add additional handlers to a logger (e.g., **FileHandler**).
  - Like a Logger, a handler has a minimum log level, so for a message to be published, its level must be above <u>both</u> the logger and handlers' threshold.

LoggerTest.java

```
import java.io.IOException;
import java.util.logging.*;
public class LoggerTest {
        public static void main(String[] args) {
                Logger logger = Logger.getLogger("TEST");
                logger.setLevel(Level.SEVERE);
                System.out.println(logger.getLevel());
                // Add a FileHandler
                try {
                        Handler h = new FileHandler("MyLog.txt");
                        logger.addHandler(h);
                        h.setLevel(Level.SEVERE);
                } catch (SecurityException | IOException e) {
                        // Do something
                }
                logger.severe("Testing...");
        }
}
```

getCanonicalName

Class

```
public String getCanonicalName()
```

Returns the canonical name of the underlying class as defined by the Java Language Specification. Returns null if the underlying class does not have a canonical name (i.e., if it is a local or anonymous class or an array whose component type does not have a canonical name).

Returns:

the canonical name of the underlying class if it exists, and `null` otherwise.

Since:

1.5

isAnonymousClass

```
public boolean isAnonymousClass()
```

Returns `true` if and only if the underlying class is an anonymous class.

Returns:

`true` if and only if this class is an anonymous class.

Since:

1.5

isLocalClass

```
public boolean isLocalClass()
```

Returns `true` if and only if the underlying class is a local class.

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.logging.*;

public class JHTTP {

  private static final Logger logger = Logger.getLogger(
      JHTTP.class.getCanonicalName());
  private static final int NUM_THREADS = 50;
  private static final String INDEX_FILE = "index.html";

  private final File rootDirectory;
  private final int port;

  public JHTTP(File rootDirectory, int port) throws IOException {

    if (!rootDirectory.isDirectory()) {
      throw new IOException(rootDirectory
          + " does not exist as a directory");
    }
    this.rootDirectory = rootDirectory;
    this.port = port;
  }

  public void start() throws IOException {
    ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
    try (ServerSocket server = new ServerSocket(port)) {
      logger.info("Accepting connections on port " + server.getLocalPort());
      logger.info("Document Root: " + rootDirectory);
```

The JHTTP constructor

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.logging.*;

public class JHTTP {

  private static final Logger logger = Logger.getLogger(
      JHTTP.class.getCanonicalName());
  private static final int NUM_THREADS = 50;
  private static final String INDEX_FILE = "index.html";

  private final File rootDirectory;
  private final int port;

  public JHTTP(File rootDirectory, int port) throws IOException {

    if (!rootDirectory.isDirectory()) {
      throw new IOException(rootDirectory
          + " does not exist as a directory");
    }
    this.rootDirectory = rootDirectory;
    this.port = port;
  }

  public void start() throws IOException {
    ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
    try (ServerSocket server = new ServerSocket(port)) {
      logger.info("Accepting connections on port " + server.getLocalPort());
      logger.info("Document Root: " + rootDirectory);
```

Checks if arg[0] is a directoty

compact1, compact2, compact3

java.io

Class File

java.lang.Object
    java.io.File

All Implemented Interfaces:

Serializable, Comparable<File>

```
public class File
extends Object
implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, "/" for the UNIX root directory, or "\\\\" for a Microsoft Windows UNC pathname, and
2. A sequence of zero or more string *names*.

The first name in an abstract pathname may be a directory name or, in the case of Microsoft Windows UNC pathnames, a hostname. Each subsequent name in an abstract pathname denotes a directory; the last name may denote either a directory or a file. The *empty* abstract pathname has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying

String	**getPath**() Converts this abstract pathname into a pathname string.
long	**getTotalSpace**() Returns the size of the partition named by this abstract pathname.
long	**getUsableSpace**() Returns the number of bytes available to this virtual machine on the partition named by this abstract pathname.
int	**hashCode**() Computes a hash code for this abstract pathname.
boolean	**isAbsolute**() Tests whether this abstract pathname is absolute.
boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.
boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.
boolean	**isHidden**() Tests whether the file named by this abstract pathname is a hidden file.
long	**lastModified**() Returns the time that the file denoted by this abstract pathname was last modified.
long	**length**() Returns the length of the file denoted by this abstract pathname.
String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[]	**list**(FilenameFilter filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[]	**listFiles**(FileFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**(FilenameFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.logging.*;

public class JHTTP {

  private static final Logger logger = Logger.getLogger(
      JHTTP.class.getCanonicalName());
  private static final int NUM_THREADS = 50;
  private static final String INDEX_FILE = "index.html";

  private final File rootDirectory;
  private final int port;

  public JHTTP(File rootDirectory, int port) throws IOException {

    if (!rootDirectory.isDirectory()) {
      throw new IOException(rootDirectory
          + " does not exist as a directory");
    }
    this.rootDirectory = rootDirectory;
    this.port = port;
  }

  public void start() throws IOException {
    ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
    try (ServerSocket server = new ServerSocket(port)) {
      logger.info("Accepting connections on port " + server.getLocalPort());
      logger.info("Document Root: " + rootDirectory);
```

JHTTP creates a thread pool to handle requests and repeatedly accepts incoming connections.

The Executor Interface Revisited

- We can change dispatching strategies simply by changing the kind of **Executor** we instantiate.
  - For example, if we wanted to use a fixed-size thread pool as in our **TCPEchoServerPool**.java example, it is a matter of changing one line associated with setting the dispatch service:

```
Executor service = Executors.newFixedThreadPool(threadPoolSize);
```

- We could switch to a single thread to execute all connections either by specifying a pool size of 1, or by the following call:

```
Executor service = Executors.newSingleThreadExecutor();
```

  - In this Executor approach, if the single "worker" thread dies because of some failure, the Executor will replace it with a new thread.

```
public JHTTP(File rootDirectory, int port) throws IOException {

  if (!rootDirectory.isDirectory()) {
    throw new IOException(rootDirectory
        + " does not exist as a directory");
  }
  this.rootDirectory = rootDirectory;
  this.port = port;
}

public void start() throws IOException {
  ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
  try (ServerSocket server = new ServerSocket(port)) {
    logger.info("Accepting connections on port " + server.getLocalPort());
    logger.info("Document Root: " + rootDirectory);

    while (true) {
      try {
        Socket request = server.accept();
        Runnable r = new RequestProcessor(
            rootDirectory, INDEX_FILE, request);
        pool.submit(r);
      } catch (IOException ex) {
        logger.log(Level.WARNING, "Error accepting connection", ex);
      }
    }
  }
}

public static void main(String[] args) {

  // get the Document root
```

Instantiate **RequestProcessor** to create a runnable object r

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess

cafeaulait.org

```
import java.io.*;
import java.net.*;
import java.nio.file.Files;
import java.util.*;
import java.util.logging.*;

public class RequestProcessor implements Runnable {

  private final static Logger logger = Logger.getLogger(
      RequestProcessor.class.getCanonicalName());

  private File rootDirectory;
  private String indexFileName = "index.html";
  private Socket connection;

  public RequestProcessor(File rootDirectory,
      String indexFileName, Socket connection) {

    if (rootDirectory.isFile()) {
      throw new IllegalArgumentException(
          "rootDirectory must be a directory, not a file");
    }
    try {
      rootDirectory = rootDirectory.getCanonicalFile();
    } catch (IOException ex) {
    }
    this.rootDirectory = rootDirectory;

    if (indexFileName != null) this.indexFileName = indexFileName;
    this.connection = connection;
  }

  @Override
  public void run() {
    // for security checks
    String root = rootDirectory.getPath();
```

The **RequestProcessor** constructor

```java
public JHTTP(File rootDirectory, int port) throws IOException {

  if (!rootDirectory.isDirectory()) {
    throw new IOException(rootDirectory
        + " does not exist as a directory");
  }
  this.rootDirectory = rootDirectory;
  this.port = port;
}

public void start() throws IOException {
  ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
  try (ServerSocket server = new ServerSocket(port)) {
    logger.info("Accepting connections on port " + server.getLocalPort());
    logger.info("Document Root: " + rootDirectory);

    while (true) {
      try {
        Socket request = server.accept();
        Runnable r = new RequestProcessor(
            rootDirectory, INDEX_FILE, request);
        pool.submit(r);
      } catch (IOException ex) {
        logger.log(Level.WARNING, "Error accepting connection", ex);
      }
    }
  }
}

public static void main(String[] args) {

  // get the Document root
```

You submit one **RequestProcessor** thread per incoming connection into the pool.

```java
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

Each connection is handled by the **run()** method of the RequestProcessor class.

```
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

It gets input and output streams from the socket and chains them to a reader and a writer.

```java
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

The reader reads the first line of the client request to determine the version of HTTP that the client supports and the requested file.

```java
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

The reader reads the first line of the client request to determine the version of HTTP that the client supports and the requested file.

split

```
public String[] split(String regex)
```

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument `split` method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string `"boo:and:foo"`, for example, yields the following results with these expressions:

Regex	Result
`:`	`{ "boo", "and", "foo" }`
`o`	`{ "b", "", ":and:f" }`

Parameters:

`regex` - the delimiting regular expression

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

Throws:

`PatternSyntaxException` - if the regular expression's syntax is invalid

Since:

1.4

See Also:

`Pattern`

toLowerCase

compact1, compact2, compact3

java.util.regex

Class Pattern

java.lang.Object
    java.util.regex.Pattern

All Implemented Interfaces:

Serializable

```
public final class Pattern
extends Object
implements Serializable
```

A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

A typical invocation sequence is thus

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

A `matches` method is defined by this class as a convenience for when a regular expression is used just once. This method compiles an expression and matches an input sequence against it in a single invocation. The statement

`\n`	The newline (line feed) character (`'\u000A'`)
`\r`	The carriage-return character (`'\u000D'`)
`\f`	The form-feed character (`'\u000C'`)
`\a`	The alert (bell) character (`'\u0007'`)
`\e`	The escape character (`'\u001B'`)
`\cx`	The control character corresponding to *x*

Character classes

`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]`(subtraction)

Predefined character classes

`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\h`	A horizontal whitespace character: `[ \t\xA0\u1680\u180e\u2000-\u200a\u202f\u205f\u3000]`
`\H`	A non-horizontal whitespace character: `[^\h]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\v`	A vertical whitespace character: `[\n\x0B\f\r\x85\u2028\u2029]`
`\V`	A non-vertical whitespace character: `[^\v]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

POSIX character classes (US-ASCII only)

Linebreak matcher

\R	Any Unicode linebreak sequence, is equivalent to `\u000D\u000A|[\u000A\u000B\u000C\u000D\u0085\u2028\u2029]`

Greedy quantifiers

X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times

Reluctant quantifiers

X??	*X*, once or not at all
*X**?	*X*, zero or more times
X+?	*X*, one or more times
X{*n*}?	*X*, exactly *n* times
X{*n*,}?	*X*, at least *n* times
X{*n*,*m*}?	*X*, at least *n* but not more than *m* times

Possessive quantifiers

X?+	*X*, once or not at all
*X**+	*X*, zero or more times
X++	*X*, one or more times
X{*n*}+	*X*, exactly *n* times
X{*n*,}+	*X*, at least *n* times
X{*n*,*m*}+	*X*, at least *n* but not more than *m* times

Logical operators

XY	*X* followed by *Y*
X|*Y*	Either *X* or *Y*

```java
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

Assuming the method is GET, the file that is requested is converted to a filename on the local file system.

```
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

If the file requested is a directory, you add the name of an index file.

```java
@Override
public void run() {
  // for security checks
  String root = rootDirectory.getPath();
  try {
    OutputStream raw = new BufferedOutputStream(
                        connection.getOutputStream()
                       );
    Writer out = new OutputStreamWriter(raw);
    Reader in = new InputStreamReader(
                new BufferedInputStream(
                 connection.getInputStream()
                ),"US-ASCII"
               );
    StringBuilder requestLine = new StringBuilder();
    while (true) {
      int c = in.read();
      if (c == '\r' || c == '\n') break;
      requestLine.append((char) c);
    }

    String get = requestLine.toString();

    logger.info(connection.getRemoteSocketAddress() + " " + get);

    String[] tokens = get.split("\\s+");
    String method = tokens[0];
    String version = "";
    if (method.equals("GET")) {
      String fileName = tokens[1];
      if (fileName.endsWith("/")) fileName += indexFileName;
      String contentType =
          URLConnection.getFileNameMap().getContentTypeFor(fileName);
      if (tokens.length > 2) {
        version = tokens[2];
      }
```

You need to get the content type of the file, to be used to generate the **Content-Type** header in the <u>response</u>.

To figure out the content type, you call the following method to map file extensions such as *.html* onto MIME types such as text/html:

getFileNameMap

For URLConnection

```
public static FileNameMap getFileNameMap()
```

Loads filename map (a mimetable) from a data file. It will first try to load the user-specific table, defined by "content.types.user.table" property. If that fails, it tries to load the default built-in table at lib/content-types.properties under java home.

Returns:

the FileNameMap

Since:

1.2

See Also:

`setFileNameMap(java.net.FileNameMap)`

setFileNameMap

```
public static void setFileNameMap(FileNameMap map)
```

Sets the FileNameMap.

If there is a security manager, this method first calls the security manager's `checkSetFactory` method to ensure the operation is allowed. This could result in a SecurityException.

getContentTypeFor

For FileNameMap

```
String getContentTypeFor(String fileName)
```

Gets the MIME type for the specified file name.

Parameters:

`fileName` - the specified file name

Returns:

a `String` indicating the MIME type for the specified file name.

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

URLConnection Revisited

- After you've successfully created a URL object, you can call the URL object's **openConnection** method to get a **URLConnection** object
  - You can use this URLConnection object to setup parameters and general request properties that you may need before connecting.
  - Connection to the remote object represented by the URL is only initiated when the **URLConnection.connect**() method is called.
  - When you do this you are initializing a communication link between your Java program and the URL over the network:

```
try {
    URL myURL = new URL("http://example.com/");
    URLConnection myURLConnection = myURL.openConnection();
    myURLConnection.connect();
}
catch (MalformedURLException e) { // new URL() failed // ... }
catch (IOException e) { // openConnection() failed // ... }
```

openConnection

```
public URLConnection openConnection()
                              throws IOException
```

Returns a URLConnection instance that represents a connection to the remote object referred to by the URL.

A new instance of URLConnection is created every time when invoking the URLStreamHandler.openConnection(URL) method of the protocol handler for this URL.

It should be noted that a URLConnection instance does not establish the actual network connection on creation. This will happen only when calling URLConnection.connect().

If for the URL's protocol (such as HTTP or JAR), there exists a public, specialized URLConnection subclass belonging to one of the following packages or one of their subpackages: java.lang, java.io, java.util, java.net, the connection returned will be of that subclass. For example, for HTTP an HttpURLConnection will be returned, and for JAR a JarURLConnection will be returned.

Returns:

a URLConnection linking to the URL.

Throws:

IOException - if an I/O exception occurs.

See Also:

URL(java.lang.String, java.lang.String, int, java.lang.String)

openConnection

```
public URLConnection openConnection(Proxy proxy)
                              throws IOException
```

Same as openConnection(), except that the connection will be made through the specified proxy; Protocol handlers that do not support proxing will ignore ...king this method preempts the system's default ProxySelector settings.

URLConnection Revisited

- The following program performs the same function as the **URLReader** program shown earlier. But it explicitly retrieves a URLConnection object and gets an input stream from the connection, instead of getting an input stream directly from the URL.
- The connection is opened implicitly by calling getInputStream:

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL syr = new URL("http://www.syr.edu/");
        URLConnection yc = syr.openConnection();
        BufferedReader in = new BufferedReader(new InputStreamReader(
                                                    yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) System.out.println(inputLine);
        in.close();
    }
}
```

URLConnection Revisited

- Sometimes you need to write data to a **URLConnection**, for example, when you submit a form to a web server using **POST** or upload a file using **PUT**. (To be introduced later.)
  - The getOutputStream() method returns an OutputStream on which you can write data for transmission to a server:

```
public OutputStream getOutputStream()
```

- A URLConnection doesn't allow output by default, so you have to call **setDoOutput(true)** before asking for an output stream.
  - When you set doOutput to true for a URL, the request method is changed from GET to POST.

A Full-Fledged HTTP/Web Server

- This is all you'll need from the client, although a more advanced web server would read the rest of the MIME header the client sends.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
if (method.equals("GET")) {
  String fileName = tokens[1];
  if (fileName.endsWith("/")) fileName += indexFileName;
  String contentType =
      URLConnection.getFileNameMap().getContentTypeFor(fileName);
  if (tokens.length > 2) {
    version = tokens[2];
  }

  File theFile = new File(rootDirectory,
      fileName.substring(1, fileName.length()));

  if (theFile.canRead()
      // Don't let clients outside the document root
      && theFile.getCanonicalPath().startsWith(root)) {
    byte[] theData = Files.readAllBytes(theFile.toPath());
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
    }

    // send the file; it may be an image or other binary data
    // so use the underlying output stream
    // instead of the writer
    raw.write(theData);
    raw.flush();
  } else { // can't find the file
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 404 File Not Found",
          "text/html; charset=utf-8", body.length());
    }
    out.write(body);
```

You use the canonical path to make sure that the requested file doesn't come from outside the document root directory. Otherwise, a sneaky client could walk all over the local file system by including .. in URLs to walk up the directory hierarchy.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
if (method.equals("GET")) {
  String fileName = tokens[1];
  if (fileName.endsWith("/")) fileName += indexFileName;
  String contentType =
      URLConnection.getFileNameMap().getContentTypeFor(fileName);
  if (tokens.length > 2) {
    version = tokens[2];
  }

  File theFile = new File(rootDirectory,
      fileName.substring(1, fileName.length()));

  if (theFile.canRead()
      // Don't let clients outside the document root
      && theFile.getCanonicalPath().startsWith(root)) {
    byte[] theData = Files.readAllBytes(theFile.toPath());
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
    }

    // send the file; it may be an image or other binary data
    // so use the underlying output stream
    // instead of the writer
    raw.write(theData);
    raw.flush();
  } else { // can't find the file
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 404 File Not Found",
          "text/html; charset=utf-8", body.length());
    }
    out.write(body);
```

Next, the requested file is opened and its contents are read into a byte array.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
if (method.equals("GET")) {
  String fileName = tokens[1];
  if (fileName.endsWith("/")) fileName += indexFileName;
  String contentType =
      URLConnection.getFileNameMap().getContentTypeFor(fileName);
  if (tokens.length > 2) {
    version = tokens[2];
  }

  File theFile = new File(rootDirectory,
      fileName.substring(1, fileName.length()));

  if (theFile.canRead()
      // Don't let clients outside the document root
      && theFile.getCanonicalPath().startsWith(root)) {
    byte[] theData = Files.readAllBytes(theFile.toPath());
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
    }

    // send the file; it may be an image or other binary data
    // so use the underlying output stream
    // instead of the writer
    raw.write(theData);
    raw.flush();
  } else { // can't find the file
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 404 File Not Found",
          "text/html; charset=utf-8", body.length());
    }
    out.write(body);
```

Next, the byte array containing the file's contents is written onto the output stream and the connection is closed.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
if (method.equals("GET")) {
  String fileName = tokens[1];
  if (fileName.endsWith("/")) fileName += indexFileName;
  String contentType =
      URLConnection.getFileNameMap().getContentTypeFor(fileName);
  if (tokens.length > 2) {
    version = tokens[2];
  }

  File theFile = new File(rootDirectory,
      fileName.substring(1, fileName.length()));

  if (theFile.canRead()
      // Don't let clients outside the document root
      && theFile.getCanonicalPath().startsWith(root)) {
    byte[] theData = Files.readAllBytes(theFile.toPath());
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
    }

    // send the file; it may be an image or other binary data
    // so use the underlying output stream
    // instead of the writer
    raw.write(theData);
    raw.flush();
  } else { // can't find the file
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 404 File Not Found",
          "text/html; charset=utf-8", body.length());
    }
    out.write(body);
```

If the file cannot be found or opened, you send the client a **404** response instead.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess

cafeaulait.org

```
if (method.equals("GET")) {
  String fileName = tokens[1];
  if (fileName.endsWith("/")) fileName += indexFileName;
  String contentType =
      URLConnection.getFileNameMap().getContentTypeFor(fileName);
  if (tokens.length > 2) {
    version = tokens[2];
  }

  File theFile = new File(rootDirectory,
      fileName.substring(1, fileName.length()));

  if (theFile.canRead()
      // Don't let clients outside the document root
      && theFile.getCanonicalPath().startsWith(root)) {
    byte[] theData = Files.readAllBytes(theFile.toPath());
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
    }

    // send the file; it may be an image or other binary data
    // so use the underlying output stream
    // instead of the writer
    raw.write(theData);
    raw.flush();
  } else { // can't find the file
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 404 File Not Found",
          "text/html; charset=utf-8", body.length());
    }
    out.write(body);
```

If the file cannot be found or opened, you send the client a **404** response instead.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
      String body = new StringBuilder("<HTML>\r\n")
          .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
          .append("</HEAD>\r\n")
          .append("<BODY>")
          .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
          .append("</BODY></HTML>\r\n").toString();
      if (version.startsWith("HTTP/")) { // send a MIME header
        sendHeader(out, "HTTP/1.0 501 Not Implemented",
                   "text/html; charset=utf-8", body.length());
      }
      out.write(body);
      out.flush();
    }
  } catch (IOException ex) {
    logger.log(Level.WARNING,
        "Error talking to " + connection.getRemoteSocketAddress(), ex);
  } finally {
    try {
      connection.close();
    }
    catch (IOException ex) {}
  }
}

private void sendHeader(Writer out, String responseCode,
    String contentType, int length)
    throws IOException {
  out.write(responseCode + "\r\n");
  Date now = new Date();
  out.write("Date: " + now + "\r\n");
  out.write("Server: JHTTP 2.0\r\n");
  out.write("Content-length: " + length + "\r\n");
  out.write("Content-type: " + contentType + "\r\n\r\n");
  out.flush();
}
}
```

The **sendHeader()** method adds some basic HTTP **headers** to the server **response**

Responses Revisited

- HTTP response messages have the form:

  Response = Status-Line

  *((general-header | response-header | entity-header) **CRLF**)

  CRLF

  [message-body]

- This says that a response is a status line, followed by some number of headers, a blank line and an optional message body.
- The status line specifies the **HTTP protocol version**, **status code** and **phrase**.
- There are five groups of status codes:

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess

cafeaulait.org

```
if (method.equals("GET")) {
  String fileName = tokens[1];
  if (fileName.endsWith("/")) fileName += indexFileName;
  String contentType =
      URLConnection.getFileNameMap().getContentTypeFor(fileName);
  if (tokens.length > 2) {
    version = tokens[2];
  }

  File theFile = new File(rootDirectory,
      fileName.substring(1, fileName.length()));

  if (theFile.canRead()
      // Don't let clients outside the document root
      && theFile.getCanonicalPath().startsWith(root)) {
    byte[] theData = Files.readAllBytes(theFile.toPath());
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
    }

    // send the file; it may be an image or other binary data
    // so use the underlying output stream
    // instead of the writer
    raw.write(theData);
    raw.flush();
  } else { // can't find the file
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 404 File Not Found",
          "text/html; charset=utf-8", body.length());
    }
    out.write(body);
```

Send message-body (aka **Entity**)

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
    } else { // can't find the file
      String body = new StringBuilder("<HTML>\r\n")
          .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
          .append("</HEAD>\r\n")
          .append("<BODY>")
          .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
          .append("</BODY></HTML>\r\n").toString();
      if (version.startsWith("HTTP/")) { // send a MIME header
        sendHeader(out, "HTTP/1.0 404 File Not Found",
            "text/html; charset=utf-8", body.length());
      }
      out.write(body);
      out.flush();
    }
  } else { // method does not equal "GET"
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 501 Not Implemented",
                "text/html; charset=utf-8", body.length());
    }
    out.write(body);
    out.flush();
  }
} catch (IOException ex) {
  logger.log(Level.WARNING,
      "Error talking to " + connection.getRemoteSocketAddress(), ex);
} finally {
  try {
    connection.close();
  }
  catch (IOException ex) {}
}
```

If the client sends a method you don't support, such as POST, you send back a **501** error.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
    } else { // can't find the file
      String body = new StringBuilder("<HTML>\r\n")
          .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
          .append("</HEAD>\r\n")
          .append("<BODY>")
          .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
          .append("</BODY></HTML>\r\n").toString();
      if (version.startsWith("HTTP/")) { // send a MIME header
        sendHeader(out, "HTTP/1.0 404 File Not Found",
            "text/html; charset=utf-8", body.length());
      }
      out.write(body);
      out.flush();
    }
  } else { // method does not equal "GET"
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 501 Not Implemented",
                "text/html; charset=utf-8", body.length());
    }
    out.write(body);
    out.flush();
  }
} catch (IOException ex) {
  logger.log(Level.WARNING,
      "Error talking to " + connection.getRemoteSocketAddress(), ex);
} finally {
  try {
    connection.close();
  }
  catch (IOException ex) {}
}
```

If the client sends a method you don't support, such as POST, you send back a **501** error.

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
    } else { // can't find the file
      String body = new StringBuilder("<HTML>\r\n")
          .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
          .append("</HEAD>\r\n")
          .append("<BODY>")
          .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
          .append("</BODY></HTML>\r\n").toString();
      if (version.startsWith("HTTP/")) { // send a MIME header
        sendHeader(out, "HTTP/1.0 404 File Not Found",
            "text/html; charset=utf-8", body.length());
      }
      out.write(body);
      out.flush();
    }
  } else { // method does not equal "GET"
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 501 Not Implemented",
                "text/html; charset=utf-8", body.length());
    }
    out.write(body);
    out.flush();
  }
} catch (IOException ex) {
  logger.log(Level.WARNING,
      "Error talking to " + connection.getRemoteSocketAddress(), ex);
} finally {
  try {
    connection.close();
  }
  catch (IOException ex) {}
}
```

Send message-body, which just those headers (above)

http://www.cafeaulait.org/books/jnp4/examples/src/RequestProcess cafeaulait.org

```
    } else { // can't find the file
      String body = new StringBuilder("<HTML>\r\n")
          .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
          .append("</HEAD>\r\n")
          .append("<BODY>")
          .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
          .append("</BODY></HTML>\r\n").toString();
      if (version.startsWith("HTTP/")) { // send a MIME header
        sendHeader(out, "HTTP/1.0 404 File Not Found",
            "text/html; charset=utf-8", body.length());
      }
      out.write(body);
      out.flush();
    }
  } else { // method does not equal "GET"
    String body = new StringBuilder("<HTML>\r\n")
        .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
        .append("</HEAD>\r\n")
        .append("<BODY>")
        .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
        .append("</BODY></HTML>\r\n").toString();
    if (version.startsWith("HTTP/")) { // send a MIME header
      sendHeader(out, "HTTP/1.0 501 Not Implemented",
                "text/html; charset=utf-8", body.length());
    }
    out.write(body);
    out.flush();
  }
} catch (IOException ex) {
  logger.log(Level.WARNING,
      "Error talking to " + connection.getRemoteSocketAddress(), ex);
} finally {
  try {
    connection.close();
  }
  catch (IOException ex) {}
}
```

If an exception occurs, you log it, close the connection, and continue.