How Tomcat Works

lanng

Published with GitBook

Table of Contents

OverView

Welcome to How Tomcat Works. This book dissects Tomcat 4.1.12 and 5.0.18 and explains the internal workings of its free, open source, and most popular servlet container code-named Catalina. Tomcat is a complex system, consisting of many different components. Those who want to learn how Tomcat works often do know where to start. What this book does is provide the big picture and then build a simpler version of each component to make understanding that component easier. Only after that will the real component be explained.

You should start by reading this Introduction as it explains the structure of the book and gives you the brief outline of the applications built. The section "Preparing the Prerequisite Software" gives you instructions on what software you need to download, how to make a directory structure for your code, etc.

Who This Book Is for

This book is for anyone working with the Java technology.

- This book is for you if you are a servlet/JSP programmer or a Tomcat user and you are interested in knowing how a servlet container works.
- It is for you if you want to join the Tomcat development team because you need to first learn how the existing code works.
- If you have never been involved in web development but you have interest in software development in general, then you can learn from this book how a large application such as Tomcat was designed and developed.
- If you need to configure and customize Tomcat, you should read this book.

To understand the discussion in this book, you need to understand object-oriented programming in Java as well as servlet programming. If you are not familiar with the latter, there are a plethora of books on sendets, including Budi's Java for the Web with Servlets, JSP, and EJB. To make the material easier to understand, each chapter starts with background information that will be required to understand the topic in discussion.

How A Servlet Container Works

A servlet container is a complex system. However, basically there are three things that a servlet container does to service a request for a servlet:

- Creating a request object and populate it with information that may be used by the invoked servlet, such as parameters, headers, cookies, query string, URI, etc. A request object is an instance of the javax.servlet.ServletRequest interface or the javax.servlet.http.ServletRequest interface.
- Creating a response object that the invoked servlet uses to send the response to the web client. A response object is an instance of the javax.servlet.ServletResponse interface or the javax.servlet.http.ServletResponse interface.
- Invoking the service method of the servlet, passing the request and response objects. Here the servlet reads the values from the request object and writes to the response object.

As you read the chapters, you will find detailed discussions of Catalina servlet container.

Catalina Block Diagram

Catalina is a very sophisticated piece of software, which was elegantly designed and developed. It is also modular too. Based on the tasks mentioned in the section "How A Servlet Container Works", you can view Catalina as consisting of two main modules: the connector and the container.

The block diagram in Figure I.1 is, of course, simplistic. Later in the following chapters you will unveil all smaller components one by one.

Figure I.1: Catalina's main modules

Now, back to Figure I.1, the connector is there to connect a request with the container. Its job is to construct a request object and a response object for each HTTP request it receives. It then passes processing to the container. The container receives the request and response objects from the connector and is responsible for invoking the servlet's service method.

Bear in mind though, that the description above is only the tip of the iceberg. There are a lot of things that a container does. For example, before it can invoke a servlet's service method, it must load the servlet, authenticate the user (if required), update the session for that user, etc. It's not surprising then that a container uses many different modules for processing. For example, the manager module is for processing user sessions, the loader is for loading servlet classess, etc.

Tomcat 4 and 5

This book covers both Tomcat 4 and 5. Here are some of the differences between the two:

- Tomcat 5 supports Servlet 2.4 and JSP 2.0 specifications, Tomcat 4 supports Servlet 2.3 and JSP 1.2.
- Tomcat 5 has a more efficient default connector than Tomcat 4.
- Tomcat 5 shares a thread for background processing whereas Tomcat 4's
- components all have their own threads for background processing. Therefore, Tomcat 5 uses less resources in this regard.
- Tomcat 5 does not need a mapper component to find a child component, therefore simplifying the code.

Overview of Each Chapter

There are 20 chapters in this book. The first two chapters serve as an introduction. Chapter 1 explains how an HTTP server works and Chapter 2 features a simple servlet container. The next two chapters focus on the connector and Chapters 5 to 20 cover each of the components in the container. The following is the summary of each of the chapters.

Note For each chapter, there is an accompanying application similar to the component being explained.

Chapter 1 starts this book by presenting a simple HTTP server. To build a working HTTP server, you need to know the internal workings of two classes in the java.net package: Socket and ServerSocket. There is sufficient background information in this chapter about these two classes for you to understand how the accompanying application works.

Chapter 2 explains how simple servlet containers work. This chapter comes with two servlet container applications that can service requests for static resources as well as very simple servlets. In particular, you will learn how you can create request and response objects and pass them to the requested servlet's service method. There is also a servlet that can be run inside the servlet containers and that you can invoke from a web browser.

Chapter 3 presents a simplified version of Tomcat 4's default connector. The application built in this chapter serves as a learning tool to understand the connector discussed in Chapter 4.

Chapter 4 presents Tomcat 4's default connector. This connector has been deprecated in favor of a faster connector called Coyote. Nevertheless, the default connector is simpler and easier to understand.

Chapter 5 discusses the container module. A container is represented by the org.apache.catalina.Container interface and there are four types of containers: engine, host, context, and wrapper. This chapter offers two applications that work with contexts and wrappers.

Chapter 6 explains the Lifecycle interface. This interface defines the lifecycle of a Catalina component and provides an elegant way of notifying other components of events that occur in that component. In addition, the Lifecycle interface provides an elegant mechanism for starting and stopping all the components in Catalina by one single start/stop.

Chapter 7 covers loggers, which are components used for recording error messages and other messages.

Chapter 8 explains about loaders. A loader is an important Catalina module responsible for loading servlet and other classes that a web application uses. This chapter also shows how application reloading is achieved.

Chapter 9 discusses the manager, the component that manages sessions in session management. It explains the various types of managers and how a manager can persist session objects into a store. At the end of the chapter, you will learn how to build an application that uses a StandardManager instance to run a servlet that uses session objects to store values.

Chapter 10 covers web application security constraints for restricting access to certain contents. You will learn entities related to security such as principals, roles, login config, authenticators, etc. You will also write two applications that install an authenticator valve in the StandardContext object and uses basic authentication to authenticate users.

Chapter 11 explains in detail the org.apache.catalina.core.StandardWrapper class that represents a servlet in a web application. In particular, this chapter explains how filters and a servlet's service method are invoked. The application accompanying this chapter uses StandardWrapper instances to represents servlets.

Chapter 12 covers the org.apache.catalina.core.StandardContext class that represents a web application. In particular this chapter discusses how a StandardContext object is configured, what happens in it for each incoming HTTP request, how it supports automatic reloading, and how Tomcat 5 shares a thread that executes periodic tasks in its associated components.

Chapter 13 presents the two other containers: host and engine. You can also find the standard implementation of these two containers:

- org.apache.catalina.core.StandardHost
- org.apache.catalina.core.StandardEngine

Chapter 14 offers the server and service components. A server provides an elegant start and stop mechanism for the whole servlet container, a service serves as a holder for a container and one or more connectors. The application accompanying this chapter shows how to use a server and a service.

Chapters 15 explains the configuration of a web application through Digester, an exciting open source project from the Apache Software Foundation. For those not initiated, this chapter presents a section that gently introduces the digester library and how to use it to convert the nodes in an XML document to Java objects. It then explains the ContextConfig object that configures a StandardContext instance.

Chapter 16 explains the shutdown hook that Tomcat uses to always get a chance to do clean-up regardless how the user stops it (i.e. either appropriately by sending a shutdown command or inappropriately by simply closing the console.)

Chapter 17 discusses the starting and stopping of Tomcat through the use of batch files and shell scripts.

Chapter 18 presents the deployer, the component responsible for deploying and installing web applications.

Chapter 19 discusses a special interface, ContainerServlet, to give a servlet access to the Catalina internal objects. In particular, it discusses the Manager application that you can use to manage deployed applications.

Chapter 20 discusses JMX and how Tomcat make its internal objects manageable by creating MBeans for those objects.

The Application for Each Chapter

Each chapter comes with one or more applications that focus on a specific component in Catalina. Normally you'll find the simplified version of the component being explained or code that explains how to use a Catalina component. All classes and interfaces in the chapters' applications reside in the ex[chapter number].pyrmont package or its subpackages. For example, the classes in the application in **Chapter 1** are part of the ex01.pyrmont package.

Preparing the Prerequisite Software

The applications accompanying this book run with J2SE version 1.4. The zipped source files can be downloaded from the authors' web site www.brainysoftware.com. It contains the source code for Tomcat 4.1.12 and the applications used in this book. Assuming you have installed J2SE 1.4 and your path environment variable includes the location of the JDK, follow these steps:

1. Extractthezipfiles.Allextractedfileswillresideinanewdirectorycalled **HowTomcatWorks**. HowTomcatWorks is your

working directory. There will be several subdirectories under HowTomcatWorks, including **lib** (containing all needed libraries), **src** (containing the source files), **webroot** (containing an HTML file and three sample servlets), and **webapps** (containing sample applications).
2. Changedirectorytotheworkingdirectoryandcompilethejavafiles.Ifyou are using Windows, run the **win-compile.bat** file. If your computer is a Linux machine, type the following: (don't forget to chmod the file if necessary)./linux-compile.sh

Note More information can be found in the Readme.txt file included in the ZIP file.

Note: All examples with Servlet API 3.1.0 and JDK 6

本章概要

本章节主要介绍Java Web服务的工作原理。

Web服务使用HTTP协议和客户端交互，因此又被成为HTTP服务，而和它交互的客户端一般都是网页浏览器。 一个基于Java实现的Web服务器通常用到两个重要的类：java.net.Socket 和 java.net.ServerSocket，而这两个类之间的交互都是通过HTTP消息实现的。

因此，本章就从讨论HTTP和这两个类开始。最后我们会通过动手实现一个简单的Web服务应用来结束本章。

The Hypertext Transfer Protocol (HTTP)

HTTP is the protocol that allows web servers and browsers to send and receive data over the Internet. It is a request and response protocol. The client requests a file and the server responds to the request. HTTP uses reliable TCP connections—by default on TCP port 80. The first version of HTTP was HTTP/0.9, which was then overridden by HTTP/1.0. Replacing HTTP/1.0 is the current version of HTTP/1.1, which is defined in Request for Comments (RFC) 2616 and downloadable from http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf.

Note: This section covers HTTP 1.1 only briefly and is intended to help you understand the messages sent by web server applications. If you are interested in more details, read RFC 2616.

In HTTP, it is always the client who initiates a transaction by establishing a connection and sending an HTTP request. The web server is in no position to contact a client or make a callback connection to the client. Either the client or the server can prematurely terminate a connection. For example, when using a web browser you can click the Stop button on your browser to stop the download process of a file, effectively closing the HTTP connection with the web server.

HTTP Requests

An HTTP request consists of three components:

- Method—Uniform Resource Identifier (URI)—Protocol/Version
- Request headers
- Entity body

An example of an HTTP request is the following:

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html Accept-Language: en-gb Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98) Content-Length: 33
Content-Type: application/x-www-form-urlencoded Accept-Encoding: gzip, deflate
lastName=Franks&firstName=Michael
```

The method—URI—protocol version appears as the first line of the request.

```
POST /examples/default.jsp HTTP/1.1
```

where POST is the request method, /examples/default.jsp represents the URI and HTTP/1.1 the Protocol/Version section.

Each HTTP request can use one of the many request methods as specified in the HTTP standards. The HTTP 1.1 supports seven types of request: GET, POST, HEAD, OPTIONS, PUT, DELETE, and TRACE. GET and POST are the most commonly used in Internet applications.

The URI specifies an Internet resource completely. A URI is usually interpreted as being relative to the server's root directory. Thus, it should always begin with a forward slash /. A Uniform Resource Locator (URL) is actually a type of URI (see http://www.ietf.org/rfc/rfc2396.txt). The protocol version represents the version of the HTTP protocol being used.

The request header contains useful information about the client environment and the entity body of the request. For example, it could contain the language the browser is set for, the length of the entity body, and so on. Each header is separated by a carriage return/linefeed (CRLF) sequence.

Between the headers and the entity body, there is a blank line (CRLF) that is important to the HTTP request format. The CRLF tells the HTTP server where the entity body begins. In some Internet programming books, this CRLF is considered the fourth component of an HTTP request.

Between the headers and the entity body, there is a blank line (CRLF) that is important to the HTTP request format. The

CRLF tells the HTTP server where the entity body begins. In some Internet programming books, this CRLF is considered the fourth component of an HTTP request.

In the previous HTTP request, the entity body is simply the following line:

```
lastName=Franks&firstName=Michael
```

The entity body can easily become much longer in a typical HTTP request.

HTTP Responses

Similar to an HTTP request, an HTTP response also consists of three parts:

- Protocol—Status code—Description
- Response headers
- Entity body

The following is an example of an HTTP response:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 5 Jan 2004 13:13:33 GMT Content-Type: text/html
Last-Modified: Mon, 5 Jan 2004 13:13:12 GMT     Content-Length: 112
<html>
<head>
<title>HTTP Response Example</title> </head>
<body>
Welcome to Brainy Software
</body>
</html>
```

The first line of the response header is similar to the first line of the request header. The first line tells you that the protocol used is HTTP version 1.1, the request succeeded (200 = success), and that everything went okay.

The response headers contain useful information similar to the headers in the request. The entity body of the response is the HTML content of the response itself. The headers and the entity body are separated by a sequence of CRLFs.

The Socket Class

A socket is an endpoint of a network connection. A socket enables an application to read from and write to the network. Two software applications residing on two different computers can communicate with each other by sending and receiving byte streams over a connection. To send a message from your application to another application, you need to know the IP address as well as the port number of the socket of the other application. In Java, a socket is represented by the java.net.Socket class.

To create a socket, you can use one of the many constructors of the Socket class. One of these constructors accepts the host name and the port number:

```
public Socket (java.lang.String host, int port)
```

where host is the remote machine name or IP address and port is the port number of the remote application. For example, to connect to yahoo.com at port 80, you would construct the following Socket object:

```
new Socket ("yahoo.com", 80);
```

Once you create an instance of the Socket class successfully, you can use it to send and receive streams of bytes. To send byte streams, you must first call the Socket class's getOutputStream method to obtain a java.io.OutputStream object. To send text to a remote application, you often want to construct a java.io.PrintWriter object from the OutputStream object returned. To receive byte streams from the other end of the connection, you call the Socket class's getInputStream method that returns a java.io.InputStream.

The following code snippet creates a socket that can communicate with a local HTTP server (127.0.0.1 denotes a local host), sends an HTTP request, and receives the response from the server. It creates a StringBuffer object to hold the response and prints it on the console.

```
Socket socket = new Socket("127.0.0.1", "8080");
OutputStream os = socket.getOutputStream();
boolean autoflush = true;
PrintWriter out = new PrintWriter(socket.getOutputStream(), autoflush);
BufferedReader in = new BufferedReader(new InputStreamReader( socket.getInputstream() ));
// send an HTTP request to the web server          out.println("GET /index.jsp HTTP/1.1");          out.println("Host: localhost:8080");
out.println("Connection: Close");
out.println();
// read the response
boolean loop = true;
StringBuffer sb = new StringBuffer(8096);
while (loop) {
    if ( in.ready() ) {
        int i=0;
        while (i!=-1) {
            i = in.read();
            sb.append((char) i);
        }
        loop = false; }
        Thread.currentThread().sleep(50);
    }
    // display the response to the out console
System.out.println(sb.toString());
socket.close();
```

Note that to get a proper response from the web server, you need to send an HTTP request that complies with the HTTP protocol. If you have read the previous section, The Hypertext Transfer Protocol (HTTP), you should be able to understand the HTTP request in the code above.

Note: You can use the com.brainysoftware.pyrmont.util.HttpSniffer class included with this book to send an HTTP request and display the response. To use this Java program, you must be connected to the Internet. Be warned, though, that it may not work if you are behind a firewall.

The ServerSocket Class

The Socket class represents a "client" socket, i.e. a socket that you construct whenever you want to connect to a remote server application. Now, if you want to implement a server application, such as an HTTP server or an FTP server, you need a different approach. This is because your server must stand by all the time as it does not know when a client application will try to connect to it. In order for your application to be able to stand by all the time, you need to use the **java.net.ServerSocket class**. This is an implementation of a server socket.

ServerSocket is different from Socket. The role of a server socket is to wait for connection requests from clients. Once the server socket gets a connection request, it creates a Socket instance to handle the communication with the client.

To create a server socket, you need to use one of the four constructors the **ServerSocket** class provides. You need to specify the IP address and port number the server socket will be listening on. Typically, the IP address will be 127.0.0.1, meaning that the server socket will be listening on the local machine. The IP address the server socket is listening on is referred to as the binding address. Another important property of a server socket is its backlog, which is the maximum queue length of incoming connection requests before the server socket starts to refuse the incoming requests.

One of the constructors of the **ServerSocket** class has the following signature:

```
public ServerSocket(int port, int backLog, InetAddress bindingAddress);
```

Notice that for this constructor, the binding address must be an instance of **java.net.InetAddress**. An easy way to construct an **InetAddress** object is by calling its static method **getByName**, passing a String containing the host name, such as in the following code.

```
InetAddress.getByName("127.0.0.1");
```

The following line of code constructs a ServerSocket that listens on port 8080 of the local machine. The ServerSocket has a backlog of 1.

```
new ServerSocket(8080, 1, InetAddress.getByName("127.0.0.1"));
```

Once you have a **ServerSocket** instance, you can tell it to wait for an incoming connection request to the binding address at the port the server socket is listening on. You do this by calling the **ServerSocket** class's accept method. This method will only return when there is a connection request and its return value is an instance of the Socket class. This Socket object can then be used to send and receive byte streams from the client application, as explained in the previous section, "The Socket Class". Practically, the accept method is the only method used in the application accompanying this chapter.

The Application

Our web server application is part of the ex01.pyrmont package and consists of three **classes**:

- **HttpServer**
- **Request**
- **Response**

The entry point of this application (the static main method) can be found in the **HttpServer** class. The main method creates an instance of **HttpServer** and calls its await method. The await method, as the name implies, waits for HTTP requests on a designated port, processes them, and sends responses back to the clients. It keeps waiting until a shutdown command is received.

The application cannot do more than sending static resources, such as HTML files and image files, residing in a certain directory. It also displays the incoming HTTP request byte streams on the console. However, it does not send any header, such as dates or cookies, to the browser.

We will now take a look at the three classes in the following subsections.

The HttpServer Class

The HttpServer class represents a web server and is presented in Listing 1.1. Note that the await method is given in Listing 1.2 and is not repeated in Listing 1.1 to save space.

Listing 1.1: The HttpServer class

```
pakage ex01.pyrmont;

import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.File;
public class HttpServer {
/** WEB_ROOT is the directory where our HTML and other files reside.
 * For this package, WEB_ROOT is the "webroot" directory under the
 * working directory.
 * The working directory is the location in the file     system
 * from where the java command was invoked.
 */
    public static final String WEB_ROOT = System.getProperty("user.dir") + File.separator + "webroot";
    // shutdown command
    private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
    // the shutdown command received
    private boolean shutdown = false;
    public static void main(String[] args) {
        HttpServer     server = new HttpServer();
        server.await();
    }
    public void await() {
        ...
    }
}
```

Listing 1.2: The HttpServer class's await method

```
public void await() {
    ServerSocket serverSocket = null; int port = 8080;
    try {
        serverSocket = new ServerSocket(port, 1,InetAddress.getByName("127.0.0.1"));
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
    // Loop waiting for a request
    while (!shutdown) {
        Socket socket = null;
        InputStream input = null;
        OutputStream output = null;
        try {
            socket = serverSocket.accept();
            input = socket.getInputStream();
            output = socket.getOutputStream();
            // create Request object and parse
            Request request = new Request(input);
            request.parse();
            // create Response object
            Response response = new Response(output);
            response.setRequest(request);
            response.sendStaticResource();
            // Close the socket socket.close();
            //check if the previous URI is a shutdown command
            shutdown = request.getUri().equals(SHUTDOWN_COMMAND); }
        catch (Exception e) {
            e.printStackTrace ();
            continue;
        }
    }
}
```

This web server can serve static resources found in the directory indicated by the public static final WEB_ROOT and all subdirectories under it. WEB_ROOT is initialized as follows:

```
public static final String WEB_ROOT = System.getProperty("user.dir") + File.separator + "webroot";
```

The code listings include a directory called webroot that contains some static resources that you can use for testing this application. You can also find several servlets in the same directory for testing applications in the next chapters.

To request for a static resource, you type the following URL in your browser's Address or URL box:

```
http://machineName:port/staticResource
```

For instance, if you are using the same computer to test the application and you want to ask the HttpServer object to send the index.html file, you use the following URL:

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

To stop the server, you send a shutdown command from a web browser by typing the pre-defined string in the browser's Address or URL box, after the host:port section of the URL. The shutdown command is defined by the SHUTDOWN static final variable in the HttpServer class:

```
private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
```

Therefore, to stop the server, you use the following URL:

```
http://localhost:8080/SHUTDOWN
```

Now, let's look at the await method printed in Listing 1.2.

The method name await is used instead of wait because wait is an important method in the java.lang.Object class for working with threads.

The await method starts by creating an instance of ServerSocket and then going into a while loop.

```
serverSocket = new ServerSocket(port, 1, InetAddress.getByName("127.0.0.1"));
...
// Loop waiting for a request
while (!shutdown) {
...
}
```

The code inside the while loop stops at the accept method of ServerSocket, which returns only when an HTTP request is received on port 8080:

```
socket = serverSocket.accept();
```

Upon receiving a request, the await method obtains java.io.InputStream and java.io.OutputStream objects from the Socket instance returned by the accept method.

```
input = socket.getInputStream();
output = socket.getOutputStream();
```

The await method then creates an ex01.pyrmont.Request object and calls its parse method to parse the HTTP request raw data.

```
// create Request object and parse
Request request = new Request(input);
request.parse ();
```

Afterwards, the await method creates a Response object, sets the Request object to it, and calls its sendStaticResource method.

```
// create Response object
Response response = new Response(output);
response.setRequest(request);
response.sendStaticResource();
```

Finally, the await method closes the Socket and calls the getUri method of Request to check if the URI of the HTTP request is a shutdown command. If it is, the shutdown variable is set to true and the program exits the while loop.

```
// Close the socket socket.close ();
//check if the previous URI is a shutdown command
shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
```

The Request Class

The ex01.pyrmont.Request class represents an HTTP request. An instance of this class is constructed by passing the InputStream object obtained from a Socket that handles the communication with the client. You call one of the read methods of the InputStream object to obtain the HTTP request raw data.

The Request class is offered in Listing 1.3. The Request class has two public methods, parse and getUri, which are given in Listings 1.4 and 1.5, respectively.

Listing 1.3: The Request class

```
package ex01.pyrmont;

import java.io.InputStream;
import java.io.IOException;

public class Request {
   private InputStream input;
   private String uri;
   public Request(InputStream input) {
      this.input = input;
   }
   public void parse() {
      ...
   }
   private String parseUri(String requestString) {
      ...
   }
   public String getUri() {
      return uri;
   }
}
```

Listing 1.4: The Request class's parse method

```
public void parse() {
// Read a set of characters from the socket
   StringBuffer request = new StringBuffer(2048);
   int i;
   byte[] buffer = new byte[2048];
   try {
      i = input.read(buffer);
   }catch (IOException e) {
      e.printStackTrace();
      i = -1;
   }
   for (int j=0; j < i; j++) {
      request.append((char) buffer[j]);
   }
   System.out.print(request.toString());
   uri = parseUri(request.toString());
}
```

Listing 1.5: the Request class's parseUri method

```
private String parseUri(String requestString) {
   int index1, index2;
   index1 = requestString.indexOf(' ');
   if (index1 != -1) {
      index2 = requestString.indexOf(' ', index1 + 1);     if (index2 > index1)
      return requestString.substring(index1 + 1, index2);
   }
   return null;
}
```

The parse method parses the raw data in the HTTP request. Not much is done by this method. The only information it makes available is the URI of the HTTP request that it obtains by calling the private method parseUri. The parseUri method stores the URI in the uri variable. The public getUri method is invoked to return the URI of the HTTP request.

Note: More processing of the HTTP request raw data will be done in the applications accompanying Chapter 3 and the subsequent chapters.

To understand how the parse and parseUri methods work, you need to know the structure of an HTTP request, discussed in the previous section, "The Hypertext Transfer Protocol (HTTP)". In this chapter, we are only interested in the first part of the HTTP request, the request line. A request line begins with a method token, followed by the request URI and the protocol version, and ends with carriage-return linefeed (CRLF) characters. Elements in a request line are separated by a space character. For instance, the request line for a request for the index.html file using the GET method is as follows.

```
GET /index.html HTTP/1.1
```

The parse method reads the whole byte stream from the socket's InputStream that is passed to the Request object and stores the byte array in a buffer. It then populates a StringBuffer object called request using the bytes in the buffer byte array, and passes the String representation of the StringBuffer to the parseUri method.

The parse method is given in Listing 1.4.

The parseUri method then obtains the URI from the request line. Listing 1.5 presents the parseUri method. The parseUri method searches for the first and the second spaces in the request and obtains the URI from it.

The Response Class

The ex01.pyrmont.Response class represents an HTTP response and is given in Listing 1.6.

Listing 1.6: The Response class

```
package ex01.pyrmont;

import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.File;

/*
HTTP Response = Status-Line
*(( general-header | response-header | entity-header ) CRLF)
CRLF
[ message-body ]
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
*/

public class Response {
    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;
    public Response(OutputStream output) {
    this.output = output;
    }
    public void setRequest(Request request) {
    this.request = request;
    }
    public void sendStaticResource() throws IOException {     byte[] bytes = new byte[BUFFER_SIZE];
    FileInputStream fis = null;
    try {
        File file = new File(HttpServer.WEB_ROOT, request.getUri());
        if (file.exists()) {
        fis = new FileInputStream(file);
        int ch = fis.read(bytes, 0, BUFFER_SIZE);
        while (ch!=-1) {
            output.write(bytes, 0, ch);
            ch = fis.read(bytes, 0, BUFFER_SIZE);
            }
    } else {
    // file not found
        String errorMessage = "HTTP/1.1 404 File Not Found\r\n" + "Content-Type: text/html\r\n" + "Content-Length: 23\r\n" +"\r\n" +"<\h
        output.write(errorMessage.getBytes());
    } }catch (Exception e) {
    // thrown if cannot instantiate a File object
    System.out.println(e.toString() );
    } finally {
        if (fis!=null) fis.close();
    } }
}
```

First note that its constructor accepts a java.io.OutputStream object, such as the following.

```
public Response(OutputStream output) {
    this.output = output;
}
```

A Response object is constructed by the HttpServer class's await method by passing the OutputStream object obtained from the socket.

The Response class has two public methods: setRequest and sendStaticResource method. The setRequest method is used to pass a Request object to the Response object.

The sendStaticResource method is used to send a static resource, such as an HTML file. It first instantiates the java.io.File class by passing the parent path and child path to the File class's constructor.

```
File file = new File(HttpServer.WEB_ROOT, request.getUri());
```

It then checks if the file exists. If it does, sendStaticResource constructs a java.io.FileInputStream object by passing the File object. Then, it invokes the read method of the FileInputStream and writes the byte array to the OutputStream output. Note that in this case the content of the static resource is sent to the browser as raw data.

```
if (file.exists()) {
    fis = new FileInputstream(file);
    int ch = fis.read(bytes, 0, BUFFER_SIZE);
    while (ch!=-1) {
        output.write(bytes, 0, ch);
        ch = fis.read(bytes, 0, BUFFER_SIZE);
    }
}
```

If the file does not exist, the sendStaticResource method sends an error message to the browser.

```
String errorMessage = "HTTP/1.1 404 File Not Found\r\n" + "Content-Type: text/html\r\n" +"Content-Length: 23\r\n" + "\r\n" + "<h1>Fi
output.write(errorMessage.getBytes());
```

Running the Application

To run the application, from the working directory, type the following:

```
java ex01.pyrmont.HttpServer
```

To test the application, open your browser and type the following in the URL or Address box:

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

You will see the index.html page displayed in your browser, as in Figure 1.1.

Figure 1.1: The output from the web server

On the console, you can see the HTTP request similar to the following:

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/msword, application/vnd.ms- pov
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: localhost:8080
Connection: Keep-Alive

GET /images/logo.gif HTTP/1.1
Accept: */*
Referer: http://localhost:8080/index.html
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: localhost:8080
Connection: Keep-Alive
```

Summary

In this chapter you have seen how a simple web server works. The application accompanying this chapter consists of only three classes and is not fully functional. Nevertheless, it serves as a good learning tool. The **next chapter** will discuss the processing of dynamic contents.

Overview

This chapter explains how you can develop your own servlet container by presenting two applications. The first application has been designed to be as simple as possible to make it easy for you to understand how a servlet container works. It then evolves into the second servlet container, which is slightly more complex.

Note: Every servlet container application in each chapter gradually evolves from the application in the previous chapter, until a fully-functional Tomcat servlet container is built in Chapter 17.

Both servlet containers can process simple servlets as well as static resources. You can use PrimitiveServlet to test this container. PrimitiveServlet is given in Listing 2.1 and its class file can be found in the webroot directory. More complex servlets are beyond the capabilities of these containers, but you will learn how to build more sophisticated servlet containers in the next chapters.

Listing 2.1: PrimitiveServlet.java

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;
public class PrimitiveServlet implements Servlet {
   public void init(ServletConfig config) throws ServletException {
      System.out.println("init");
   }
   public void service(ServletRequest request,ServletResponse response) throws ServletException, IOException {
      System.out.println("from service");
      PrintWriter out = response.getWriter();
      out.println("Hello. Roses are red.");
      out.print("Violets are blue.");
   }
   public void destroy() {
      System.out.println("destroy");
   }
   public String getServletInfo() {
      return null;
   }
   public ServletConfig getServletConfig() {
      return null;
   }
}
```

The classes for both applications are part of the ex02.pyrmont package. To understand how the applications work, you need to be familiar with the javax.servlet.Servlet interface. To refresh your memory, this interface is discussed in the first section of this chapter. After that, you will learn what a servlet container has to do to serve HTTP requests for a servlet.

The javax.servlet.Servlet Interface

Servlet programming is made possible through the classes and interfaces in two packages: javax.servlet and javax.servlet.http. Of those classes and interfaces, the javax.servlet.Servlet interface is one of the most importance. All servlets must implement this interface or extend a class that does.

The Servlet interface has five methods whose signatures are as follows.

```
public void init(ServletConfig config) throws ServletException
public void service(ServletRequest request, ServletResponse response) throws ServletException, java.io.IOException
public void destroy()
public ServletConfig getServletConfig()
public java.lang.String getServletInfo()
```

Of the five methods in Servlet, the init, service, and destroy methods are the servlet's life cycle methods. The init method is called by the servlet container after the servlet class has been instantiated. The servlet container calls this method exactly once to indicate to the servlet that the servlet is being placed into service. The init method must complete successfully before the servlet can receive any requests. A servlet programmer can override this method to write initialization code that needs to run only once, such as loading a database driver, initializing values, and so on. In other cases, this method is normally left blank.

The servlet container calls the service method of a servlet whenever there is a request for the servlet. The servlet container passes a javax.servlet.ServletRequest object and a javax.servlet.ServletResponse object. The ServletRequest object contains the client's HTTP request information and the ServletResponse object encapsulates the servlet's response. The service method is invoked many times during the life of the servlet.

The servlet container calls the destroy method before removing a servlet instance from service. This normally happens when the servlet container is shut down or the servlet container needs some free memory. This method is called only after all threads within the servlet's service method have exited or after a timeout period has passed. After the servlet container has called the destroy method, it will not call the service method again on the same servlet. The destroy method gives the servlet an opportunity to clean up any resources that are being held, such as memory, file handles, and threads, and make sure that any persistent state is synchronized with the servlet's current state in memory.

Listing 2.1 presents the code for a servlet named PrimitiveServlet, which is a very simple servlet that you can use to test the servlet container applications in this chapter. The PrimitiveServlet class implements javax.servlet.Servlet (as all servlets must) and provides implementations for all the five methods of Servlet. What PrimitiveServlet does is very simple. Each time any of the init, service, or destroy methods is called, the servlet writes the method's name to the standard console. In addition, the service method obtains the java.io.PrintWriter object from the ServletResponse object and sends strings to the browser.

Application 1

Now, let's examine servlet programming from a servlet container's perspective. In a nutshell, a fully-functional servlet container does the following for each HTTP request for a servlet:

- When the servlet is called for the first time, load the servlet class and call the servlet's init method (once only)
- For each request, construct an instance of javax.servlet.ServletRequest and an instance of javax.servlet.ServletResponse.
- Invoke the servlet's service method, passing the ServletRequest and ServletResponse objects.
- When the servlet class is shut down, call the servlet's destroy method and unload the servlet class.

The first servlet container for this chapter is not fully functional. Therefore, it cannot run other than very simple servlets and does not call the servlets' init and destroy methods. Instead, it does the following:

- Wait for HTTP requests.
- Construct a ServletRequest object and a ServletResponse object.
- If the request is for a static resource, invoke the process method of the StaticResourceProcessor instance, passing the ServletRequest and ServletResponse objects.
- If the request is for a servlet, load the servlet class and invoke the service method of the servlet, passing the ServletRequest and ServletResponse objects.

Note: In this servlet container, the servlet class is loaded every time the servlet is requested.

The first application consists of six classes:

- HttpServer1
- Request
- Response
- StaticResourceProcessor
- ServletProcessor1
- Constants

Figure 2.1 displays the UML diagram of the first servlet container.

Figure 2.1: The UML diagram of the first servlet container

The entry point of this application (the static main method) is in the HttpServer1 class. The main method creates an instance of HttpServer1 and calls its await method. The await method waits for HTTP requests, creates a Request object and a Response object for every request, and dispatch them either to a StaticResourceProcessor instance or a ServletProcessor instance, depending on whether the request is for a static resource or a servlet.

The Constants class contains the static final WEB_ROOT that is referenced from other classes. WEB_ROOT indicates the location of PrimitiveServlet and the static resource that can be served by this container.

The HttpServer1 instance keeps waiting for HTTP requests until a shutdown command is received. You issue a shutdown command the same way as you did it in Chapter 1.

Each of the classes in the application is discussed in the following sections.

The HttpServer1 Class

The HttpServer1 class in this application is similar to the HttpServer class in the simple web server application in Chapter 1. However, in this application the HttpServer1 class can serve both static resources and servlets. To request a static resource, you type a URL in the following format in your browser's Address or URL box:

```
http://machineName:port/staticResource
```

This is exactly how you requested a static resource in the web server application in Chapter 1.

To request a servlet, you use the following URL:

```
http://machineName:port/servlet/servletClass
```

Therefore, if you are using a browser locally to request a servlet called PrimitiveServlet, you enter the following URL in the browser's Address or URL box:

```
http://localhost:8080/servlet/PrimitiveServlet
```

This servlet container can serve PrimitiveServlet. However, if you invoke the other servlet, ModernServlet, the servlet container will throw an exception. At the later chapters, you will build applications that can process both.

The HttpServer1 class is presented in Listing 2.2.

Listing 2.2: The HttpServer1 Class's await method

```
package ex02.pyrmont;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

public class HttpServer1 {
/** WEB_ROOT is the directory where our HTML and other files reside.
* For this package, WEB_ROOT is the "webroot" directory under the
* working directory.
* The working directory is the location in the file system
* from where the java command was invoked.
*/
// shutdown command
private static final String SHUTDOWN_COMMAND = "/SHUTDOWN";
// the shutdown command received
private boolean     shutdown = false;
public static void main(String[] args) {
   HttpServer1 server = new HttpServer1(); server.await();
   }
public void await() {
   ServerSocket serverSocket = null;
   int port = 8080;
try {
   serverSocket = new ServerSocket(port, 1, InetAddress.getByName("127.0.0.1"));
   }catch (IOException e) {
      e.printStackTrace(); System.exit(1);
   }
   // Loop waiting for a request
   while (!shutdown) {
   Socket socket = null;
   InputStream input = null;
   OutputStream output = null;
   try {
      socket = serverSocket.accept();
      input = socket.getInputstream();
      output = socket.getOutputStream();
      // create Request object and parse
      Request request = new Request(input);
      request.parse();
      // create Response object
      Response response = new Response(output);          response.setRequest(request);
      // check if this is a request for a servlet or
      // a static resource
      // a request for a servlet begins with "/servlet/"
   if (request.getUri().startsWith("/servlet/")) {
      ServletProcessor1 processor = new ServletProcessor1();
      processor.process(request, response);
      }else {
      StaticResoureProcessor processor =
new StaticResourceProcessor();
      processor.process(request, response);
   }
   // Close the socket
   socket.close();
   //check if the previous URI is a shutdown command
   shutdown = request.getUri().equals(SHUTDOWN_COMMAND);
   }catch (Exception e) {
   e.printStackTrace();
   System.exit(1);
}}}}
```

The class's await method waits for HTTP requests until a shutdown command is issued, and reminds you of the await method in Chapter 1. The difference between the await method in Listing 2.2 and the one in Chapter 1 is that in Listing 2.2 the request can be dispatched to either a StaticResourceProcessor or a ServletProcessor. The request is forwarded to the latter if the URI contains the string /servlet/.Otherwise, the request is passed to the StaticResourceProcessor instance. Notice that that part is greyed in Listing 2.2.

The Request Class

A servlet's service method receives a javax.servlet.ServletRequest instance and a javax.servlet.ServletResponse instance from the servlet container. This is to say that for every HTTP request, a servlet container must construct a ServletRequest object and a ServletResponse object and pass them to the service method of the servlet it is serving.

The ex02.pyrmont.Request class represents a request object to be passed to the servlet's service method. As such, it must implement the javax.servlet.ServletRequest interface. This class has to provide implementations for all methods in the interface. However, we would like to make it very simple and provide the implementations of some of the methods only we leave the full method implementations for the chapters to come. In order to compile the Request class, you need to provide "blank" implementations for those methods. If you look at the Request class in Listing 2.3, you will see that all methods whose signatures return an object instance return a null.

Listing 2.3: The Request class

```
package ex02.pyrmont;
import java.io.InputStream;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.UnsupportedEncodingException;
import java.util.Enumeration;
import java.util.Locale;
import java.util.Map;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletRequest;
public class Request implements ServletRequest {
    private InputStream input;
    private String uri;
    public Request(InputStream input){
        this.input = input;
    }
    public String getUri() {
        return uri;
    }
    private String parseUri(String requestString) {
        int index1, index2;
        index1 = requestString.indexOf(' ');
        if (index1 != -1) {
            index2 = requestString.indexOf(' ', index1 + 1);
        }
        if (index2 > index1)
            return requestString.substring(index1 + 1, index2);
        return null;
    }
    public void parse() {
    // Read a set of characters from the socket
        StringBuffer request = new StringBuffer(2048);
        int i;
        byte[] buffer = new byte[2048];
        try {
            i = input.read(buffer);
        }catch (IOException e) {
            e.printStackTrace();
            i = -1;
        }
        for (int j=0; j < i; j++) {
        request.append((char) buffer(j));
        }
        System.out.print(request.toString());
        uri = parseUri(request.toString());
    }
```

In addition, the Request class still has the parse and the getUri methods which were discussed in Chapter 1.

The Response Class

The ex02.pyrmont.Response class, given in Listing 2.4, implements javax.servlet.ServletResponse. As such, the class must provide implementations for all the methods in the interface. Similar to the Request class, we leave the implementations of all methods "blank", except for the getWriter method.

Listing 2.4: The Response class

```
package ex02.pyrmont;

import java.io.OutputStream;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.File;
import java.io.PrintWriter;
import java.util.Locale;
import javax.servlet.ServletResponse;
import javax.servlet.ServletOutputStream;

public class Response implements ServletResponse {
    private static final int BUFFER_SIZE = 1024;
    Request request;
    OutputStream output;
    PrintWriter writer;
    public Response(OutputStream output) {
    this.output = output;
    }
    public void setRequest(Request request) {
        this.request = request;
    }
    /* This method is used to serve static pages */
    public void sendStaticResource() throws IOException {
        byte[] bytes = new byte[BUFFER_SIZE];
        FileInputstream fis = null;
        try {
        /* request.getUri has been replaced by                  request.getRequestURI */
        File file = new File(Constants.WEB_ROOT, request.getUri());
        fis = new FileInputstream(file);
        /*
        HTTP Response = Status-Line
        *(( general-header | response-header | entity-header ) CRLF)
        CRLF
        [ message-body ]
        Status-Line = HTTP-Version SP Status-Code SP           Reason-Phrase CRLF
        */
        int ch = fis.read(bytes, 0, BUFFER_SIZE);
        while (ch!=-1) {
            output.write(bytes, 0, ch);
            ch = fis.read(bytes, 0, BUFFER_SIZE); }
        }catch (FileNotFoundException e) {
            String errorMessage = "HTTP/1.1 404 File Not Found\r\n" + "Content-Type: text/html\r\n" + "Content-Length: 23\r\n" + "\r\n" +"
            output.write(errorMessage.getBytes());
            }finally {
                if (fis!=null)
                fis.close(); }
            }
    /** implementation of ServletResponse */
    public void flushBuffer() throws IOException { }
    public int getBufferSize() {
        return 0;
    }
    public String getCharacterEncoding() {
        return null;
    }
    public Locale getLocale() {
        return null;
    }
    public ServletOutputStream getOutputStream() throws IOException {
        return null;
    }
    public boolean isCommitted() {
        return false;
    }
    public void reset() { }
    public void resetBuffer() { }
    public void setBufferSize(int size) { }
    public void setContentLength(int length) { }
    public void setContentType(String type) { }
    public void setLocale(Locale locale) { }
}
```

In the getWriter method, the second argument to the PrintWriter class's constructor is a boolean indicating whether or not autoflush is enabled. Passing true as the second argument will make any call to a println method flush the output. However, a print method does not flush the output.

Therefore, if a call to a print method happens to be the last line in a servlet's service method, the output will not be sent to the browser. This imperfection will be fixed in the later applications.

The Response class still has the sendStaticResource method discussed in Chapter 1.

The StaticResourceProcessor Class

The ex02.pyrmont.StaticResourceProcessor class is used to serve requests for static resources. The only method it has is the process method. Listing 2.5 offers the StaticResourceProcessor class.

Listing 2.5: The StaticResourceProcessor class

```
package ex02.pyrmont;

import java.io.IOException;
public class StaticResourceProcessor {
   public void process(Request request, Response response) {
   try {
      response.sendStaticResource();
      }catch (IOException e) {
      e.printStackTrace();
      }
   }
}
```

The process method receives two arguments: an ex02.pyrmont.Request instance and an ex02.pyrmont.Response instance. This method simply calls the sendStaticResource method on the Response object.

The ServletProcessor1 Class

The ex02.pyrmont.ServletProcessor1 class in Listing 2.6 is there to process HTTP requests for servlets.

Listing 2.6: The ServletProcessor1 class

```
package ex02.pyrmont;

import java.net.URL;
import java.net.URLClassLoader;
import java.net.URLStreamHandler;
import java.io.File;
import java.io.IOException;
import javax.servlet.Servlet;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class ServletProcessor1 {
   public void process(Request request, Response response) {
      String uri = request.getUri();
      String servletName = uri.substring(uri.lastIndexOf("/") + 1);
      URLClassLoader loader = null;
      try {
         // create a URLClassLoader
         URL[] urls = new URL[1];
         URLStreamHandler streamHandler = null;
         File classPath = new File(Constants.WEB_ROOT);
         // the forming of repository is taken from the
         // createClassLoader method in
         // org.apache.catalina.startup.ClassLoaderFactory
         String repository =
(new URL("file", null, classPath.getCanonicalPath() +
File.separator)).toString() ;
         // the code for forming the URL is taken from
         // the addRepository method in
         // org.apache.catalina.loader.StandardClassLoader.
         urls[0] = new URL(null, repository, streamHandler);
         loader = new URLClassLoader(urls);
      }catch (IOException e) {
         System.out.println(e.toString() );
      }
      Class myClass = null; try {
      myClass = loader.loadClass(servletName);
      }catch (ClassNotFoundException e) {
         System.out.println(e.toString());
      }
         Servlet servlet = null;
      try {
         servlet = (Servlet) myClass.newInstance();          servlet.service((ServletRequest) request, (ServletResponse) response);
      }catch (Exception e) {
         System.out.println(e.toString());
      }catch (Throwable e) {
         System.out.println(e.toString());
      }
   }
}
```

The ServletProcessor1 class is surprisingly simple, consisting only of one method: process. This method accepts two arguments: an instance of javax.servlet.ServletRequest and an instance of javax.servlet.ServletResponse. From the ServletRequest, the method obtains the URI by calling the getRequestUri method:

```
String uri = request.getUri();
```

Remember that the URI is in the following format:

```
/servlet/servletName
```

where servletName is the name of the servlet class.

To load the servlet class, we need to know the servlet name from the URI. We can get the servlet name using the next line of the process method:

```
String servletName = uri.substring(uri.lastIndexOf("/") + 1);
```

Next, the process method loads the servlet. To do this, you need to create a class loader and tell this class loader the location to look for the class to be loaded. For this servlet container, the class loader is directed to look in the directory pointed by Constants.WEB_ROOT, which points to the webroot directory under the working directory.

Note: Class loaders are discussed in detail in Chapter 8.

To load a servlet, you use the java.net.URLClassLoader class, which is an indirect child class of the java.lang.ClassLoader class. Once you have an instance of URLClassLoader, you use its loadClass method to load a servlet class. Instantiating the URLClassLoader class is straightforward. This class has three constructors, the simplest of which being:

```
public URLClassLoader(URL[] urls);
```

where urls is an array of java.net.URL objects pointing to the locations on which searches will be conducted when loading a class. Any URL that ends with a / is assumed to refer to a directory. Otherwise, the URL is assumed to refer to a JAR file, which will be downloaded and opened as needed.

Note: In a servlet container, the location where a class loader can find servlet classes is called a repository.

In our application, there is only one location that the class loader must look, i.e. the webroot directory under the working directory. Therefore, we start by creating an array of a single URL. The URL class provides a number of constructors, so there are many ways of constructing a URL object. For this application, we used the same constructor used in another class in Tomcat. The constructor has the following signature.

```
public URL(URL context, java.lang.String spec, URLStreamHandler hander) throws MalformedURLException
```

You can use this constructor by passing a specification for the second argument and null for both the first and the third arguments. However, there is another constructor that accepts three arguments:

```
public URL(java.lang.String protocol, java.lang.String host, java.lang.String file) throws MalformedURLException
```

Therefore, the compiler will not know which constructor you mean if you simply write the following code:

```
new URL(null, aString, null);
```

You can get around this by telling the compiler the type of the third argument, like this.

```
URLStreamHandler streamHandler = null;
new URL(null, aString, streamHandler);
```

For the second argument, you pass a String containing the repository (the directory where servlet classes can be found), which you form by using the following code:

```
String repository = (new URL("file", null, classPath.getCanonicalPath() + File.separator)).toString() ;
```

Combining all the pieces together, here is the part of the process method that constructs the appropriate URLClassLoader instance:

```
// create a URLClassLoader
URL[] urls = new URL[1];
URLStreamHandler streamHandler = null;
File classPath = new File(Constants.WEB_ROOT); String repository = (new URL("file", null,
classPath.getCanonicalPath() + File.separator)).toString() ;
urls[0] = new URL(null, repository, streamHandler);
loader = new URLClassLoader(urls);
```

Note: The code that forms the repository is taken from the createClassLoader method in org.apache.catalina.startup.ClassLoaderFactory and the code for forming the URL is taken from the addRepository method in org.apache.catalina.loader.StandardClassLoader. However, you don't have to worry about these classes until the later chapters.

Having a class loader, you can load a servlet class using the loadClass method:

```
Class myClass = null;
try {
   myClass = loader.loadClass(servletName);
   }catch (ClassNotFoundException e) {
   System.out.println(e.toString());
}
```

Next, the process method creates an instance of the servlet class loaded, downcasts it to javax.servlet.Servlet, and invokes the servlet's service method:

```
Servlet servlet = null;
try {
   servlet = (Servlet) myClass.newInstance();
   servlet.service((ServletRequest) request,
   (ServletResponse) response);
   }catch (Exception e) {
      System.out.println(e.toString());
   }catch (Throwable e) {
      System.out.println(e.toString());
   }
```

Running the Application

To run the application on Windows, type the following command from the working directory:

```
java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer1
```

In Linux, you use a colon to separate two libraries:

```
java -classpath ./lib/servlet.jar:./ ex02.pyrmont.HttpServer1
```

To test the application, type the following in your URL or Address box of your browser:

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

or

```
http://localhost:8080/servlet/PrimitiveServlet
```

When invoking PrimitiveServlet, you will see the following text in your browser: Hello. Roses are red.

Note that you cannot see the second string Violets are blue, because only the first string is flushed to the browser. We will fix this problem in Chapter 3, though.

Application 2

There is a serious problem in the first application. In the ServletProcessor1 class's process method, you upcast the instance of ex02.pyrmont.Request to javax.servlet.ServletRequest and pass it as the first argument to the servlet's service method. You also upcast the instance of ex02.pyrmont.Response to javax.servlet.ServletResponse and pass it as the second argument to the servlet's service method.

```
try {
   servlet = (Servlet) myClass.newInstance();
   servlet.service((ServletRequest) request,
   (ServletResponse) response);
   }
```

This compromises security. Servlet programmers who know the internal workings of this servlet container can downcast the ServletRequest and ServletResponse instances back to ex02.pyrmont.Request and ex02.pyrmont.Response respectively and call their public methods. Having a Request instance, they can call its parse method. Having a Response instance, they can call its sendStaticResource method.

You cannot make the parse and sendStaticResource methods private because they will be called from other classes. However, these two methods are not supposed to be available from inside a servlet. One solution is to make both Request and Response classes have default access modifier, so that they cannot be used from outside the ex02.pyrmont package. However, there is a more elegant solution: by using facade classes. See the UML diagram in Figure 2.2.

Figure 2.2: Façade classes

In this second application, we add two façade classes: RequestFacade and ResponseFacade.RequestFacade implements the ServletRequest interface and is instantiated by passing a Request instance that it assigns to a ServletRequest object reference in its constructor. Implementation of each method in the ServletRequest interface invokes the corresponding method of the Request object. However, the ServletRequest object itself is private and cannot be accessed from outside the class. Instead of upcasting the Request object to ServletRequest and passing it to the service method, we construct a RequestFacade object and pass it to the service method. Servlet programmers can still downcast the ServletRequest instance back to RequestFacade, however they can only access the methods available in the ServletRequest interface. Now, the parseUri method is safe.

Listing 2.7 shows an incomplete RequestFacade class.

Listing 2.7: The RequestFacade class

```
package ex02.pyrmont;

public class RequestFacade implements ServletRequest {
   private ServleLRequest request = null;
   public RequestFacade(Request request) {
      this.request = request;
   }
   /* implementation of the ServletRequest*/
   public Object getAttribute(String attribute) {
      return request.getAttribute(attribute);
   }
   public Enumeration getAttributeNames() {
      return request.getAttributeNames();
   }
   ...
}
```

Notice the constructor of RequestFacade. It accepts a Request object but immediately assigns it to the private servletRequest object reference. Notice also each method in the RequestFacade class invokes the corresponding method in the ServletRequest object.

The same applies to the ResponseFacade class.

Here are the classes used in Application 2:

- HttpServer2
- Request
- Response
- StaticResourceProcessor
- ServletProcessor2
- Constants

The HttpServer2 class is similar to HttpServer1, except that it uses ServletProcessor2 in its await method, instead of ServletProcessor1:

```
if (request.getUri().startWith("/servlet/")) {
    servletProcessor2 processor = new ServletProcessor2();
    processor.process(request, response);
    } else {
    ...
}
```

The ServletProcessor2 class is similar to ServletProcessor1, except in the following part of its process method:

```
Servlet servlet = null;
RequestFacade requestFacade = new RequestFacade(request);
ResponseFacade responseFacade = new ResponseFacade(response);
try {
    servlet = (Servlet) myClass.newInstance();
    servlet.service((ServletRequest) requestFacade, (ServletResponse) responseFacade);
    }
```

To run the application on Windows, type this from the working directory:

```
java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer2
```

In Linux, you use a colon to separate two libraries.

```
java -classpath ./lib/servlet.jar:./ ex02.pyrmont.HttpServer2
```

You can use the same URLs as in Application1 and you will get the same result.

Summary

This chapter discussed two simple servlet containers that can he used to serve static resources as well as process servlets as simple as PrimitiveServlet. Background information on the javax.servlet.Servlet interface and related types was also given.

Overview

As mentioned in Introduction, there are two main modules in Catalina: the connector and the container. In this chapter you will enhance the applications in Chapter 2 by writing a connector that creates better request and response objects. A connector compliant with Servlet 2.3 and 2.4 specifications must create instances of javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse to be passed to the invoked servlet's service method. In Chapter 2 the servlet containers could only run servlets that implement javax.servlet.Servlet and passed instances of javax.servlet.ServletRequest and javax.servlet.ServletResponse to the service method. Because the connector does not know the type of the servlet (i.e. whether it implements javax.servlet.Servlet, extends javax.servlet.GenericServlet, or extends javax.servlet.http.HttpServlet), the connector must always provide instances of HttpServletRequest and HttpServletResponse.

In this chapter's application, the connector parses HTTP request headers and enables a servlet to obtain headers, cookies, parameter names/values, etc. You will also perfect the getWriter method in the Response class in Chapter 2 so that it will behave correctly. Thanks to these enhancements, you will get a complete response from PrimitiveServlet and be able to run the more complex ModernServlet.

The connector you build in this chapter is a simplified version of the default connector that comes with Tomcat 4, which is discussed in detail in Chapter 4. Tomcat's default connector is deprecated as of version 4 of Tomcat, however it still serves as a great learning tool. For the rest of the chapter, "connector" refers to the module built in our application.

Note: Unlike the applications in the previous chapters, in this chapter's application the connector is separate from the container.

The application for this chapter can be found in the ex03.pyrmont package and its sub-packages. The classes that make up the connector are part of the ex03.pyrmont.connector and ex03.pyrmont.connector.http packages. Starting from this chapter, every accompanying application has a bootstrap class used to start the application. However, at this stage, there is not yet a mechanism to stop the application. Once run, you must stop the application abruptly by closing the console (in Windows) or by killing the process (in UNIX/Linux).

Before we explain the application, let me start with the StringManager class in the org.apache.catalina.util package. This class handles the internationalization of error messages in different modules in this application and in Catalina itself. The discussion of the accompanying application is presented afterwards.

The StringManager Class

A large application such as Tomcat needs to handle error messages carefully. In Tomcat error messages are useful for both system administrators and servlet programmers. For example, Tomcat logs error messages in order for system administrator to easily pinpoint any abnormality that happened. For servlet programmers, Tomcat sends a particular error message inside every javax.servlet.ServletException thrown so that the programmer knows what has gone wrong with his/her servlet.

The approach used in Tomcat is to store error messages in a properties file, so that editing them is easy. However, there are hundreds of classes in Tomcat. Storing all error messages used by all classes in one big properties file will easily create a maintenance nightmare. To avoid this, Tomcat allocates a properties file for each package. For example, the properties file in the org.apache.catalina.connector package contains all error messages that can be thrown from any class in that package. Each properties file is handled by an instance of the org.apache.catalina.util.StringManager class. When Tomcat is run, there will be many instances of StringManager, each of which reads a properties file specific to a package. Also, due to Tomcat's popularity, it makes sense to provide error messages in multi languages. Currently, three languages are supported. The properties file for English error messages is named LocalStrings.properties. The other two are for the Spanish and Japanese languages, in the LocalStrings_es.properties and LocalStrings_ja.properties files respectively.

When a class in a package needs to look up an error message in that package's properties file, it will first obtain an instance of StringManager. However, many classes in the same package may need a StringManager and it is a waste of resources to create a StringManager instance for every object that needs error messages. The StringManager class therefore has been designed so that an instance of StringManager is shared by all objects inside a package. If you are familiar with design patterns, you'll guess correctly that StringManager is a singleton class. The only constructor it has is private so that you cannot use the new keyword to instantiate it from outside the class. You get an instance by calling its public static method getManager, passing a package name. Each instance is stored in a Hashtable with package names as its keys.

```
private static Hashtable managers = new Hashtable();
public synchronized static StringManager
getManager(String packageName) {
   StringManager mgr = (StringManager)managers.get(packageName);
   if (mgr == null) {
      mgr = new StringManager(packageName);
      managers.put(packageName, mgr);
   }
   return mgr;
}
```

Note: An article on the Singleton pattern entitled "The Singleton Pattern" can be found in the accompanying ZIP file.

For example, to use StringManager from a class in the ex03.pyrmont.connector.http package, pass the package name to the StringManager class's getManager method:

```
StringManager sm = StringManager.getManager("ex03.pyrmont.connector.http");
```

In the ex03.pyrmont.connector.http package, you can find three properties files: LocalStrings.properties, LocalStrings_es.properties and LocalStrings_ja.properties. Which of these files will be used by the StringManager instance depends on the locale of the server running the application. If you open the LocalStrings.properties file, the first non-comment line reads:

httpConnector.alreadyInitialized=HTTP connector has already been initialized

To get an error message, use the StringManager class's getString, passing an error code. Here is the signature of one of its overloads:

```
public String getString(String key)
```

Calling getString by passing httpConnector.alreadyInitialized as the argument returns HTTP connector has already been initialized.

The Application

Starting from this chapter, the accompanying application for each chapter is divided into modules. This chapter's application consists of three modules: connector, startup, and core.

The startup module consists only of one class, Bootstrap, which starts the application. The connector module has classes that can be grouped into five categories:

- The connector and its supporting class (HttpConnector and HttpProcessor).
- The class representing HTTP requests (HttpRequest) and its supporting classes.
- The class representing HTTP responses (HttpResponse) and its supporting classes.
- Façade classes (HttpRequestFacade and HttpResponseFacade).
- The Constant class.

The core module consists of two classes: ServletProcessor and StaticResourceProcessor.

Figure 3.1 shows the UML diagram of the classes in this application. To make the diagram more readable, the classes related to HttpRequest and HttpResponse have been omitted. You can find UML diagrams for both when we discuss Request and Response objects respectively.

Figure 3.1: The UML diagram of the application

Compare the diagram with the one in Figure 2.1. The HttpServer class in Chapter 2 has been broken into two classes: HttpConnector and HttpProcessor, Request has been replaced by HttpRequest, and Response by HttpResponse. Also, more classes are used in this chapter's application.

The HttpServer class in Chapter 2 is responsible for waiting for HTTP requests and creating request and response objects. In this chapter's application, the task of waiting for HTTP requests is given to the HttpConnector instance, and the task of creating request and response objects is assigned to the HttpProcessor instance.

In this chapter, HTTP request objects are represented by the HttpRequest class, which implements javax.servlet.http.HttpServletRequest. An HttpRequest object will be cast to a HttpServletRequest instance and passed to the invoked servlet's service method. Therefore, every HttpRequest instance must have its fields properly populated so that the servlet can use them. Values that need to be assigned to the HttpRequest object include the URI, query string, parameters, cookies and other headers, etc. Because the connector does not know which values will be needed by the invoked servlet, the connector must parse all values that can be obtained from the HTTP request. However, parsing an HTTP request involves expensive string and other operations, and the connector can save lots of CPU cycles if it parses only values that will be needed by the servlet. For example, if the servlet does not need any request parameter (i.e. it does not call the getParameter, getParameterMap, getParameterNames, or getParameterValues methods of javax.servlet.http.HttpServletRequest), the connector does not need to parse these parameters from the query string and or from the HTTP request body. Tomcat's default connector (and the connector in this chapter's application) tries to be more efficient by leaving the parameter parsing until it is really needed by the servlet.

Tomcat's default connector and our connector use the SocketInputStream class for reading byte streams from the socket's InputStream. An instance of SocketInputStream wraps the java.io.InputStream instance returned by the socket's getInputStream method. The SocketInputStream class provides two important methods: readRequestLine and readHeader.readRequestLine returns the first line in an HTTP request, i.e. the line containing the URI, method and HTTP version. Because processing byte stream from the socket's input stream means reading from the first byte to the last (and never moves backwards), readRequestLine must be called only once and must be called before readHeader is called. readHeader is called to obtain a header name/value pair each time it is called and should be called repeatedly until all headers are read. The return value of readRequestLine is an instance of HttpRequestLine and the return value of readHeader is an HttpHeader object. We will discuss the HttpRequestLine and HttpHeader classes in the sections to come.

The HttpProcessor object creates instances of HttpRequest and therefore must populate fields in them. The HttpProcessor class, using its parse method, parses both the request line and headers in an HTTP request. The values resulting from the parsing are then assigned to the fields in the HttpProcessor objects. However, the parse method does not parse the parameters in the request body or query string. This task is left to the HttpRequest objects themselves. Only if the servlet needs a parameter will the query string or request body be parsed.

Another enhancement over the previous applications is the presence of the bootstrap class ex03.pyrmont.startup.Bootstrap to start the application.

We will explain the application in detail in these sub-sections:

- Starting the Application
- The Connector
- Creating an HttpRequest Object
- Creating an HttpResponse Object
- Static resource processor and servlet processor
- Running the Application

Starting the Application

You start the application from the ex03.pyrmont.startup.Bootstrap class. This class is given in Listing 3.1.

Listing 3.1: The Bootstrap class

```
package ex03.pyrmont.startup;

import ex03.pyrmont.connector.http.HttpConnector;

public final class Bootstrap {
public static void main(String[] args) {
HttpConnector connector = new HttpConnector();
connector.start();
}}
```

The main method in the Bootstrap class instantiates the HttpConnector class and calls its start method. The HttpConnector class is given in Listing 3.2.

Listing 3.2: The HttpConnector class's start method

```
package ex03.pyrmont.connector.http;

import java.io.IOException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class HttpConnector implements Runnable {
    boolean stopped;
    private String scheme = "http";
    public String getScheme() {
    return scheme;
    }
    public void run() {
    ServerSocket serverSocket = null;
    int port = 8080;
    try {
        serverSocket = new
    ServerSocket(port, 1, InetAddress.getByName("127.0.0.1"));
    }catch (IOException e) {
    e.printStackTrace(); System.exit(1);
    }
    while (!stopped) {
    // Accept the next incoming connection from the server         socket Socket socket = null;
    try {
        socket = serverSocket.accept();
    }catch (Exception e) {
        continue;
    }
    // Hand this socket off to an HttpProcessor
    HttpProcessor processor = new HttpProcessor(this);
    processor.process(socket);
    }}
    public void start() {
        Thread thread = new Thread(this);
        thread.start ();
    }
}
```

The Connector

The ex03.pyrmont.connector.http.HttpConnector class represents a connector responsible for creating a server socket that waits for incoming HTTP requests. This class is presented in Listing 3.2.

The HttpConnector class implements java.lang.Runnable so that it can be dedicated a thread of its own. When you start the application, an instance of HttpConnector is created and its run method executed.

Note: You can read the article "Working with Threads" to refresh your memory about how to create Java threads.

The run method contains a while loop that does the following:

- Waits for HTTP requests
- Creates an instance of HttpProcessor for each request.

- Calls the process method of the HttpProcessor.

Note The run method is similar to the await method of the HttpServer1 class in Chapter 2.

You can see right away that the HttpConnector class is very similar to the ex02.pyrmont.HttpServer1 class, except that after a socket is obtained from the accept method of java.net.ServerSocket, an HttpProcessor instance is created and its process method is called, passing the socket.

Note: The HttpConnector class has another method calls getScheme, which returns the scheme (HTTP).

The HttpProcessor class's process method receives the socket from an incoming HTTP request. For each incoming HTTP request, it does the following:

1. CreateanHttpRequestobject.
2. CreateanHttpResponseobject.
3. Parse the HTTP request's first line and header and populate the HttpRequest object.
4. Pass the HttpRequest and HttpResponse objects to either a ServletProcessoror a StaticResourceProcessor. Like in Chapter 2, the ServletProcessor invokes the service method of the requested servlet and the StaticResourceProcessor sends the content of a static resource.

The process method is given in Listing 3.3.

Listing 3.3: The HttpProcessor class's process method.

```
public void process(Socket socket) {
    SocketInputStream input = null;
    OutputStream output = null;
    try {
    input = new SocketInputStream(socket.getInputStream(), 2048);
    output = socket.getOutputStream();
    // create HttpRequest object and parse
    request = new HttpRequest(input);
    // create HttpResponse object
    response = new HttpResponse(output);
    response.setRequest(request);
    response.setHeader("Server", "Pyrmont Servlet Container");
    parseRequest(input, output); parseHeaders(input);
    //check if this is a request for a servlet or a static resource
    //a request for a servlet begins with "/servlet/"
    if (request.getRequestURI().startsWith("/servlet/")) {
        ServletProcessor processor = new ServletProcessor();
        processor.process(request, response);
    } else {
        StaticResourceProcessor processor = new
StaticResourceProcessor();
        processor.process(request, response);
    }
    // Close the socket socket.close();
    // no shutdown for this application
    }catch (Exception e) {
        e.printStackTrace ();
    }
}
```

The process method starts by obtaining the input stream and output stream of the socket. Note, however, in this method we use the SocketInputStream class that extends java.io.InputStream.

```
SocketInputStream input = null;
OutputStream output = null;
try {
    input = new SocketInputStream(socket.getInputStream(), 2048);
    output = socket.getOutputStream();
```

Then, it creates an HttpRequest instance and an HttpResponse instance and assigns the HttpRequest to the

HttpResponse.

```
// create HttpRequest object and parse
request = new HttpRequest(input);
// create HttpResponse object
response = new HttpResponse(output);
response.setRequest(request);
```

The HttpResponse class in this chapter's application is more sophisticated than the Response class in Chapter 2. For one, you can send headers to the client by calling its setHeader method.

```
response.setHeader("Server", "Pyrmont Servlet Container");
```

Next, the process method calls two private methods in the HttpProcessor class for parsing the request.

```
parseRequest(input, output);
parseHeaders (input);
```

Then, it hands off the HttpRequest and HttpResponse objects for processing to either a ServletProcessor or a StaticResourceProcessor, depending the URI pattern of the request.

```
if (request.getRequestURI().startsWith("/servlet/")) {
   ServletProcessor processor = new ServletProcessor();
   processor.process(request, response);
   } else {
   StaticResourceProcessor processor = new StaticResourceProcessor();
   processor.process(request, response);
}
```

Finally, it closes the socket.

```
socket.close();
```

Note also that the HttpProcessor class uses the org.apache.catalina.util.StringManager class for sending error messages:

```
protected StringManager sm = StringManager.getManager("ex03.pyrmont.connector.http");
```

The private methods in the HttpProcessor class--parseRequest, parseHeaders, and normalize-- are called to help populate the HttpRequest. These methods will be discussed in the next section, "Creating an HttpRequest Object".

Creating an HttpRequest Object

The HttpRequest class implements javax.servlet.http.HttpServletRequest. Accompanying it is a façade class called HttpRequestFacade. Figure 3.2 shows the UML diagram of the HttpRequest class and its related classes.

Figure 3.2: The HttpRequest class and related classes

Many of the methods in the HttpRequest class are left blank (you have to wait until Chapter 4 for a full implementation), but servlet programmers can already retrieve the headers, cookies and parameters of the incoming HTTP request. These three types of values are stored in the following reference variables:

```
protected HashMap headers = new HashMap();
protected ArrayList cookies = new ArrayList();
protected ParameterMap parameters = null;
```

Note ParameterMap class will be explained in the section "Obtaining Parameters".

Therefore, a servlet programmer can get the correct return values from the following methods in javax.servlet.http.HttpServletRequest:getCookies,getDateHeader, getHeader, getHeaderNames, getHeaders, getParameter, getPrameterMap, getParameterNames, and getParameterValues. Once you get headers, cookies, and parameters populated with the correct values, the implementation of the related methods are easy, as you can see in the HttpRequest class.

Needless to say, the main challenge here is to parse the HTTP request and populate the HttpRequest object. For headers and cookies, the HttpRequest class provides the addHeader and addCookie methods that are called from the parseHeaders method of HttpProcessor. Parameters are parsed when they are needed, using the HttpRequest class's parseParameters method. All methods are discussed in this section.

Since HTTP request parsing is a rather complex task, this section is divided into the following subsections:

- Reading the socket's input stream
- Parsing the request line
- Parsing headers
- Parsing cookies
- Obtaining parameters

Reading the Socket's Input Stream

In Chapters 1 and 2 you did a bit of request parsing in the ex01.pyrmont.HttpRequest and ex02.pyrmont.HttpRequest classes. You obtained the request line containing the method, the URI, and the HTTP version by invoking the read method of the java.io.InputStream class:

```
byte[] buffer = new byte [2048];
try {
// input is the InputStream from the socket.
i = input.read(buffer);
}
```

You did not attempt to parse the request further for the two applications. In the application for this chapter, however, you have the ex03.pyrmont.connector.http.SocketInputStream class, a copy of org.apache.catalina.connector.http.SocketInputStream. This class provides methods for obtaining not only the request line, but also the request headers.

You construct a SocketInputStream instance by passing an InputStream and an integer indicating the buffer size used in the instance. In this application, you create a SocketInputStream object in the process method of ex03.pyrmont.connector.http.HttpProcessor, as in the following code fragment:

```
SocketInputStream input = null;
OutputStream output = null;
try {
   input = new SocketInputStream(socket.getInputStream(), 2048);
   ...
```

As mentioned previously, the reason for having a SocketInputStream is for its two important methods: readRequestLine and readHeader. Read on.

Parsing the Request Line

The process method of HttpProcessor calls the private parseRequest method to parse the request line, i.e. the first line of an HTTP request. Here is an example of a request line:

```
GET /myApp/ModernServlet?userName=tarzan&password=pwd HTTP/1.1
```

The second part of the request line is the URI plus an optional query string. In the example above, here is the URI:

```
/myApp/ModernServlet
```

And, anything after the question mark is the query string. Therefore the query string is the following:

```
userName=tarzan&password=pwd
```

The query string can contain zero or more parameters. In the example above, there are two parameter name/value pairs: userName/tarzan and password/pwd. In servlet/JSP programming, the parameter name jsessionid is used to carry a session identifier. Session identifiers are usually embedded as cookies, but the programmer can opt to embed the session identifiers in query strings, for example if the browser's support for cookies is being turned off.

When the parseRequest method is called from the HttpProcessor class's process method, the request variable points to an instance of HttpRequest. The parseRequest method parses the request line to obtain several values and assigns these values to the HttpRequest object. Now, let's take a close look at the parseRequest method in Listing 3.4.

Listing 3.4: The parseRequest method in the HttpProcessor class

```
private void parseRequest(SocketInputStream input, OutputStream output) throws IOException, ServletException {
    // Parse the incoming request line
    input.readRequestLine(requestLine);
    String method =
new String(requestLine.method, 0, requestLine.methodEnd);
    String uri = null;
    String protocol = new String(requestLine.protocol, 0,requestLine.protocolEnd);

    // Validate the incoming request line
    if (method, length () < 1) {
        throw new ServletException("Missing HTTP request method");
    } else if (requestLine.uriEnd < 1) {
        throw new ServletException("Missing HTTP request URI");
    }
    // Parse any query parameters out of the request URI      int question = requestLine.indexOf("?");
    if (question >= 0) {
        request.setQueryString(new String(requestLine.uri, question + 1, requestLine.uriEnd - question - 1));
        uri = new String(requestLine.uri, 0, question);
    } else {
        request.setQueryString(null);
        uri = new String(requestLine.uri, 0, requestLine.uriEnd);
    }
    // Checking for an absolute URI (with the HTTP protocol)
    if (!uri.startsWith("/")) {
        int pos = uri.indexOf("://");
        // Parsing out protocol and host name
        if (pos != -1) {
            pos = uri.indexOf('/', pos + 3);
            if (pos == -1) {
                uri = "";
            } else {
                uri = uri.substring(pos);
            }
        }
    }
    // Parse any requested session ID out of the request URI
    String match = ";jsessionid=";
    int semicolon = uri.indexOf(match);
    if (semicolon >= 0) {
        String rest = uri.substring(semicolon + match,length());
        int semicolon2 = rest.indexOf(';');
    if (semicolon2 >= 0) {
        request.setRequestedSessionId(rest.substring(0, semicolon2));
        rest = rest.substring(semicolon2);
    } else {
        request.setRequestedSessionId(rest);
        rest = "";
        }
        request.setRequestedSessionURL(true);
        uri = uri.substring(0, semicolon) + rest;
    } else {
        request.setRequestedSessionId(null);
        request.setRequestedSessionURL(false);
    }
    // Normalize URI (using String operations at the moment)
    String normalizedUri = normalize(uri);
    // Set the corresponding request properties
    ((HttpRequest) request).setMethod(method);
    request.setProtocol(protocol);
    if (normalizedUri != null) {
        ((HttpRequest) request).setRequestURI(normalizedUri);
    } else {
        ((HttpRequest) request).setRequestURI(uri);
        }
        if (normalizedUri == null) {
            throw new ServletException("Invalid URI: " + uri + "'");
} }
```

The parseRequest method starts by calling the SocketInputStream class's readRequestLine method:

```
input.readRequestLine(requestLine);
```

where requestLine is an instance of HttpRequestLine inside HttpProcessor:

```
private HttpRequestLine requestLine = new HttpRequestLine();
```

Invoking its readRequestLine method tells the SocketInputStream to populate the HttpRequestLine instance.

Next, the parseRequest method obtains the method, URI, and protocol of the request line:

```
String method = new String(requestLine.method, 0, requestLine.methodEnd);
String uri = null;
String protocol = new String(requestLine.protocol, 0, requestLine.protocolEnd);
```

However, there may be a query string after the URI. If present, the query string is separated by a question mark. Therefore, the parseRequest method attempts to first obtain the query string and populates the HttpRequest object by calling its setQueryString method:

```
// Parse any query parameters out of the request URI
int question = requestLine.indexOf("?");
if (question >= 0) {
    // there is a query string.
    request.setQueryString(new String(requestLine.uri, question + 1, requestLine.uriEnd - question - 1));
    uri = new String(requestLine.uri, 0, question); }
else {
    request.setQueryString (null);
    uri = new String(requestLine.uri, 0, requestLine.uriEnd);
}
```

However, while most often a URI points to a relative resource, a URI can also be an absolute value, such as the following:

```
http://www.brainysoftware.com/index.html?name=Tarzan
```

The parseRequest method also checks this:

```
// Checking for an absolute URI (with the HTTP protocol)
if (!uri.startsWith("/")) {
    // not starting with /, this is an absolute URI
    int pos = uri.indexOf("://");
    // Parsing out protocol and host name
    if (pos != -1) {
        pos = uri.indexOf('/', pos + 3);
        if (pos == -1) {
        uri = "";
        } else {
        uri = uri.substring(pos);
    } }
}
```

Then, the query string may also contain a session identifier, indicated by the jsessionid parameter name. Therefore, the parseRequest method checks for a session identifier too. If jsessionid is found in the query string, the method obtains the session identifier and assigns the value to the HttpRequest instance by calling its setRequestedSessionId method:

```
// Parse any requested session ID out of the request URI
String match = ";
jsessionid=";
int semicolon = uri.indexOf(match);
if (semicolon >= 0) {
   String rest = uri.substring(semicolon + match.length());
   int semicolon2 = rest.indexOf(';');
   if (semicolon2 >= 0) {
      request.setRequestedSessionId(rest.substring(0, semicolon2));
      rest = rest.substring(semicolon2);
   } else {
      request.setRequestedSessionId(rest);
      rest = "";
   }
   request.setRequestedSessionURL (true);
   uri = uri.substring(0, semicolon) + rest;
} else {
   request.setRequestedSessionId(null);
   request.setRequestedSessionURL(false);
}
```

If jsessionid is found, this also means that the session identifier is carried in the query string, and not in a cookie. Therefore, pass true to the request's setRequestSessionURL method. Otherwise, pass false to the setRequestSessionURL method and null to the setRequestedSessionURL method.

At this point, the value of uri has been stripped off the jsessionid.

Then, the parseRequest method passes uri to the normalize method to correct an "abnormal" URI. For example, any occurrence of \ will be replaced by /. If uri is in good format or if the abnormality can be corrected, normalize returns the same URI or the corrected one. If the URI cannot be corrected, it will be considered invalid and normalize returns null. On such an occasion (normalize returning null), the parseRequest method will throw an exception at the end of the method.

Finally, the parseRequest method sets some properties of the HttpRequest object:

```
((HttpRequest) request).setMethod(method);
request.setProtocol(protocol);
if (normalizedUri != null) {
   ((HttpRequest) request).setRequestURI(normalizedUri);
} else {
   ((HttpRequest) request).setRequestURI(uri);
}
```

And, if the return value from the normalize method is null, the method throws an exception:

```
if (normalizedUri == null) {
   throw new ServletException("Invalid URI: " + uri + "");
}
```

Parsing Headers

An HTTP header is represented by the HttpHeader class. This class will be explained in detail in Chapter 4, for now it is sufficient to know the following:

- You can construct an HttpHeader instance by using its class's no-argument constructor.
- Once you have an HttpHeader instance, you can pass it to the readHeader method of SocketInputStream. If there is a header to read, the readHeader method will populate the HttpHeader object accordingly. If there is no more header to read, both nameEnd and valueEnd fields of the HttpHeader instance will be zero.
- To obtain the header name and value, use the following:
- String name = new String(header.name, 0, header.nameEnd);
- String value = new String(header.value, 0, header.valueEnd);

The parseHeaders method contains a while loop that keeps reading headers from the SocketInputStream until there is no more header. The loop starts by constructing an HttpHeader instance and passing it to the SocketInputStream class's readHeader:

```
HttpHeader header = new HttpHeader();
// Read the next
header input.readHeader(header);
```

Then, you can test whether or not there is a next header to be read from the input stream by testing the nameEnd and valueEnd fields of the HttpHeader instance:

```
if (header.nameEnd == 0) {
    if (header.valueEnd == 0) {
    return;
    } else {
        throw new ServletException
        (sm.getString("httpProcessor.parseHeaders.colon"));
    }
}
```

If there is a next header, the header name and value can then be retrieved:

```
String name = new String(header.name, 0, header.nameEnd);
String value = new String(header.value, 0, header.valueEnd);
```

Once you get the header name and value, you add it to the headers HashMap in the HttpRequest object by calling its addHeader method:

```
request.addHeader(name, value);
```

Some headers also require the setting of some properties. For instance, the value of the content-length header is to be returned when the servlet calls the getContentLength method of javax.servlet.ServletRequest, and the cookie header contains cookies to be added to the cookie collection. Thus, here is some processing:

```
if (name.equals("cookie")) {
    ... // process cookies here
} else if (name.equals("content-length")) {
    int n = -1;
try {
    n = Integer.parseInt (value);
    } catch (Exception e) {
        throw new ServletException(sm.getString(
"httpProcessor.parseHeaders.contentLength"));
    }
    request.setContentLength(n);
    } else if (name.equals("content-type")) {
        request.setContentType(value);
}
```

Cookie parsing is discussed in the next section, Parsing Cookies.

Parsing Cookies

Cookies are sent by a browser as an HTTP request header. Such a header has the name "cookie" and the value is the cookie name/value pair(s). Here is an example of a cookie header containing two cookies: userName and password.

```
Cookie: userName=budi; password=pwd;
```

Cookie parsing is done using the parseCookieHeader method of the org.apache.catalina.util.RequestUtil class. This method accepts the cookie header and returns an array of javax.servlet.http.Cookie. The number of elements in the array is the same as the number of cookie name/value pairs in the header. The parseCookieHeader method is given in Listing 3.5.

Listing 3.5: The org.apache.catalina.util.RequestUtil class's parseCookieHeader method

```
public static Cookie[] parseCookieHeader(String header) {
    if ((header == null) || (header.length 0 < 1) )
        return (new Cookie[0]);
        ArrayList cookies = new ArrayList();
        while (header.length() > 0) {
            int semicolon = header.indexOf(';');
            if (semicolon < 0)
                semicolon = header.length();
                if (semicolon == 0)
                    break;
                    String token = header.substring(0, semicolon);
                    if (semicolon < header.length())
                        header = header.substring(semicolon + 1);
                        else
                        header = "";
                        try {
                            int equals = token.indexOf('=');
                            if (equals > 0) {
                            String name = token.substring(0, equals).trim();
                            String value = token.substring(equals+1).trim();
                            cookies.add(new Cookie(name, value));
        } } catch (Throwable e) {
            ;
        }
    }
    return ((Cookie[]) cookies.toArray (new Cookie [cookies.size ()]));
}
```

And, here is the part of the HttpProcessor class's parseHeader method that processes the cookies:

```
else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    Cookie cookies[] = RequestUtil.ParseCookieHeader (value);
    for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("jsessionid")){
            // Override anything requested in the URL
            if (!request.isRequestedSessionIdFromCookie()) {
            // Accept only the first session id cookie               request.setRequestedSessionId(cookies[i].getValue());
            request.setRequestedSessionCookie(true);
            request.setRequestedSessionURL(false);
    } }
    request.addCookie(cookies[i]);
}
}
```

Obtaining Parameters

You don't parse the query string or HTTP request body to get parameters until the servlet needs to read one or all of them by calling the getParameter, getParameterMap, getParameterNames, or getParameterValues methods of javax.servlet.http.HttpServletRequest. Therefore, the implementations of these four methods in HttpRequest always start with a call to the parseParameter method.

The parameters only needs to be parsed once and may only be parsed once because if the parameters are to be found in the request body, parameter parsing causes the SocketInputStream to reach the end of its byte stream. The HttpRequest class employs a boolean called parsed to indicate whether or not parsing has been done.

Parameters can be found in the query string or in the request body. If the user requested the servlet using the GET method, all parameters are on the query string. If the POST method is used, you may find some in the request body too. All the name/value pairs are stored in a HashMap. Servlet programmers can obtain the parameters as a Map (by calling getParameterMap of HttpServletRequest) and the parameter name/value. There is a catch, though. Servlet programmers are not allowed to change parameter values. Therefore, a special HashMap is used: **org.apache.catalina.util.ParameterMap.**

The ParameterMap class extends java.util.HashMap and employs a boolean called locked. The name/value pairs can only be added, updated or removed if locked is false. Otherwise, an IllegalStateException is thrown. Reading the values, however, can be done any time. The ParameterMap class is given in Listing 3.6. It overrides the methods for adding, updating and removing values. Those methods can only be called when locked is false.

Listing 3.6: The org.apache.Catalina.util.ParameterMap class.

```
package org.apache.catalina.util;

import java.util.HashMap; import java.util.Map;

public final class ParameterMap extends HashMap {
    public ParameterMap() {
        super ();
    }
    public ParameterMap(int initialCapacity) {
        super(initialCapacity);
    }
    public ParameterMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
    }
    public ParameterMap(Map map) {
        super(map);
    }
    private boolean locked = false;
    public boolean isLocked() {
        return (this.locked);
    }
    public void setLocked(boolean locked) {
        this.locked = locked;
    }
    private static final StringManager sm = StringManager.getManager("org.apache.catalina.util");
    public void clear() {
        if (locked)
        throw new IllegalStateException
        (sm.getString("parameterMap.locked"));
        super.clear();
    }
    public Object put(Object key, Object value) {
        if (locked)
            throw new IllegalStateException(sm.getString("parameterMap.locked"));
            return (super.put(key, value));
    }
    public void putAll(Map map) {
        if (locked)
            throw new IllegalStateException(sm.getString("parameterMap.locked"));
            super.putAll(map);
    }
    public Object remove(Object key) {
        if (locked)
            throw new IllegalStateException(sm.getString("parameterMap.locked"));
            return (super.remove(key));
    }
}
```

Now, let's see how the parseParameters method works.

Because parameters can exist in the query string and or the HTTP request body, the parseParameters method checks both the query string and the request body. Once parsed, parameters can be found in the object variable parameters, so the method starts by checking the parsedboolean, which is true if parsing has been done before.

```
if (parsed) return;
```

Then, the parseParameters method creates a ParameterMap called results and points it to parameters. It creates a new ParameterMap if parameters is null.

```
ParameterMap results = parameters;
if (results == null)
results = new ParameterMap();
```

Then, the parseParameters method opens the parameterMap's lock to enable writing to it.

```
results.setLocked(false);
```

Next, the parseParameters method checks the encoding and assigns a default encoding if the encoding is null.

```
String encoding = getCharacterEncoding();
if (encoding == null)
encoding = "ISO-8859-1";
```

Then, the parseParameters method tries the query string. Parsing parameters is done using the parseParameters method of org.apache.Catalina.util.RequestUtil.

```
// Parse any parameters specified in the query string
String queryString = getQueryString();
try {
    RequestUtil.parseParameters(results, queryString, encoding); }
    catch (UnsupportedEncodingException e) {
    ;
}
```

Next, the method tries to see if the HTTP request body contains parameters. This happens if the user sends the request using the POST method, the content length is greater than zero, and the content type is application/x-www-form-urlencoded. So, here is the code that parses the request body.

```
// Parse any parameters specified in the input stream
String contentType = getContentType();
if (contentType == null)
    contentType = "";
    int semicolon = contentType.indexOf(';');
    if (semicolon >= 0) {
        contentType = contentType.substring (0, semicolon).trim();
        } else {
        contentType = contentType.trim();
        }
        if ("POST".equals(getMethod()) && (getContentLength() > 0)
&& "application/x-www-form-urlencoded".equals(contentType)) {
        try {
        int max = getContentLength();
        int len = 0;
        byte buf[] = new byte[getContentLength()];
        ServletInputStream is = getInputStream();
        while (len < max) {
            int next = is.read(buf, len, max - len);
            if (next < 0 ) {
                break;
            }
            len += next; }
            is.close();
            if (len < max) {
                throw new RuntimeException("Content length mismatch");
            }
                RequestUtil.parseParameters(results, buf, encoding);
                } catch (UnsupportedEncodingException ue){ ;
                } catch (IOException e) {
                throw new RuntimeException("Content read fail");
            }
}
```

Finally, the parseParameters method locks the ParameterMap back, sets parsed to true and assigns results to parameters.

```
// Store the final results results.setLocked(true);
parsed = true;
parameters = results;
```

Creating a HttpResponse Object

The HttpResponse class implements javax.servlet.http.HttpServletResponse. Accompanying it is a façade class named HttpResponseFacade. Figure 3.3 shows the UML diagram of HttpResponse and its related classes.

Figure 3.3: The HttpResponse class and related classes

In Chapter 2, you worked with an HttpResponse class that was only partially functional. For example, its getWriter method returned a java.io.PrintWriter object that does not flush automatically when one of its print methods is called. The application in this chapter fixes this problem. To understand how it is fixed, you need to know what a Writer is.

From inside a servlet, you use a PrintWriter to write characters. You may use any encoding you desire, however the characters will be sent to the browser as byte streams. Therefore, it's not surprising that in Chapter 2, the ex02.pyrmont.HttpResponse class has the following getWriter method:

```
public PrintWriter getWriter() {
// if autoflush is true, println() will flush,
// but print() will not.
// the output argument is an
OutputStream writer = new PrintWriter(output, true);
return writer;
}
```

See, how we construct a PrintWriter object? By passing an instance of java.io.OutputStream. Anything you pass to the print or println methods of PrintWriter will be translated into byte streams that will be sent through the underlying OutputStream.

In this chapter you use an instance of the ex03.pyrmont.connector.ResponseStream class as the OutputStream for the PrintWriter. Note that the ResponseStream class is indirectly derived from the java.io.OutputStream class.

You also have the ex03.pyrmont.connector.ResponseWriter class that extends the PrintWriter class. The ResponseWriter class overrides all the print and println methods and makes any call to these methods automatically flush the output to the underlying OutputStream. Therefore, we use a ResponseWriter instance with an underlying ResponseStream object.

We could instantiate the ResponseWriter class by passing an instance of ResponseStream object. However, we use a java.io.OutputStreamWriter object to serve as a bridge between the ResponseWriter object and the ResponseStream object.

With an OutputStreamWriter, characters written to it are encoded into bytes using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted. Each invocation of a write method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. The size of this buffer may be specified, but by default it is large enough for most purposes. Note that the characters passed to the write methods are not buffered.

Therefore, here is the getWriter method:

```
public PrintWriter getWriter() throws IOException {
    ResponseStream newStream = new ResponseStream(this);
    newStream.setCommit(false);
    OutputStreamWriter osr = new OutputStreamWriter(newStream, getCharacterEncoding());
    writer = new ResponseWriter(osr);
    return writer;
}
```

Static Resource Processor and Servlet Processor

The ServletProcessor class is similar to the ex02.pyrmont.ServletProcessor class in Chapter 2. They both have only one method: process. However, the process method in ex03.pyrmont.connector.ServletProcessor accepts an HttpRequest and an HttpResponse, instead of instances of Request and Response. Here is the signature of the process method in this chapter's application:

```
public void process(HttpRequest request, HttpResponse response) {
```

In addition, the process method uses HttpRequestFacade and HttpResponseFacade as facade classes for the request and the response. Also, it calls the HttpResponse class's finishResponse method after calling the servlet's service method.

```
servlet = (Servlet) myClass.newInstance();
HttpRequestFacade requestPacade = new HttpRequestFacade(request);
HttpResponseFacade responseFacade = new HttpResponseFacade(response);
servlet.service(requestFacade, responseFacade);
((HttpResponse) response).finishResponse();
```

The StaticResourceProcessor class is almost identical to the **ex02.pyrmont.StaticResourceProcessor** class.

Running the Application

To run the application in Windows, from the working directory, type the following:

```
java -classpath ./lib/servlet.jar;./ ex03.pyrmont.startup.Bootstrap
```

In Linux, you use a colon to separate two libraries.

```
java -classpath ./lib/servlet.jar:./ ex03.pyrmont.startup.Bootstrap
```

To display index.html, use the following URL:

```
http://localhost:808O/index.html
```

To invoke PrimitiveServlet, direct your browser to the following URL:

```
http://localhost:8080/servlet/PrimitiveServlet
```

You'll see the following on your browser:

```
Hello. Roses are red.
Violets are blue.
```

Note: Running PrimitiveServlet in Chapter 2 did not give you the second line.

You can also call ModernServet, which would not run in the servlet containers in Chapter 2. Here is the URL:

```
http://localhost:8080/servlet/ModernServlet
```

Note The source code for ModernServlet can be found in the webroot directory under the working directory.

You can append a query string to the URL to test the servlet. Figure 3.4 shows the result if you run ModernServlet with the following URL.

```
http://localhost:8080/servlet/ModernServlet?userName=tarzan&password=pwd
```

Figure 3.4: Running ModernServlet

Summary

In this chapter you have learned how connectors work. The connector built is a simplified version of the default connector in Tomcat 4. As you know, the default connector has been deprecated because it is not efficient. For example, all HTTP request headers are parsed, even though they might not be used in the servlet. As a result, the default connector is slow and has been replaced by Coyote, a faster connector, whose source code can be downloaded from the Apache Software Foundation's web site. The default connector, nevertheless, serves as a good learning tool and will be discussed in detail in Chapter 4.

Overview

The connector in Chapter 3 worked fine and could have been perfected to achieve much more. However, it was designed as an educational tool, an introduction to Tomcat 4's default connector. Understanding the connector in Chapter 3 is key to understanding the default connector that comes with Tomcat 4. Chapter 4 will now discuss what it takes to build a real Tomcat connector by dissecting the code of Tomcat 4's default connector.

Note: The "default connector" in this chapter refers to Tomcat 4's default connector. Even though the default connector has now been deprecated, replaced by a faster connector code-named Coyote, it is still a great learning tool.

A Tomcat connector is an independent module that can be plugged into a servlet container. There are already many connectors in existence. Examples include Coyote, mod_jk, mod_jk2, and mod_webapp. A Tomcat connector must meet the following requirements:

1. It must implement the *org.apache.catalina.Connectorinterface*.
2. It must create request objects whose class implements the *org.apache.catalina.Request interface*.
3. It must create response objects whose class implements the *org.apache.catalina.Response* interface.

Tomcat 4's default connector works similarly to the simple connector in Chapter 3. It waits for incoming HTTP requests, creates request and response objects, then passes the request and response objects to the container. A connector passes the request and response objects to the container by calling the org.apache.catalina.Container interface's invoke method, which has the following signature.

```
public void invoke( org.apache.catalina.Request request, org.apache.catalina.Response response);
```

Inside the invoke method, the container loads the servlet class, call its service method, manage sessions, log error messages, etc.The default connector also employs a few optimizations not used in Chapter 3's connector. The first is to provide a pool of various objects to avoid the expensive object creation. Secondly, in many places it uses char arrays instead of strings.

The application in this chapter is a simple container that will be associated with the default connector. However, the focus of this chapter is not this simple container but the default connector. Containers will be discussed in Chapter 5. Nevertheless, the simple container will be discussed in the section "The Simple Container Application" towards the end of this chapter, to show how to use the default connector.

Another point that needs attention is that the default connector implements all features new to HTTP 1.1 as well as able to serve HTTP 0.9 and HTTP 1.0 clients. To understand the new features in HTTP 1.1, you first need to understand these features, which we will explain in the first section of this chapter. Thereafter, we discuss the org.apache.catalina.Connector, interface and how to create the request and response objects. If you understand how the connector in Chapter 3 works, you should not find any problem understanding the default connector.

This chapter starts with three new features in HTTP 1.1. Understanding them is crucial to understanding the internal working of the default connector. Afterwards, it introduces org.apache.catalina.Connector, the interface that all connectors must implement. You then will find classes you have encountered in Chapter 3, such as HttpConnector, HttpProcessor, etc. This time, however, they are more advanced than the similar classes in Chapter 3.

HTTP 1.1 New Features

This section explains three new features of HTTP 1.1. Understanding them is crucial to understanding how the default connector processes HTTP requests.

Persistent Connections

Prior to HTTP 1.1, whenever a browser connected to a web server, the connection was closed by the server right after the requested resource was sent. However, an Internet page can contain other resources, such as image files, applets, etc. Therefore, when a page is requested, the browser also needs to download the resources referenced by the page. If the page and all resources it references are downloaded using different connections, the process will be very slow. That's why HTTP 1.1 introduced persistent connections. With a persistent connection, when a page is downloaded, the server does not close the connection straight away. Instead, it waits for the web client to request all resources referenced by the page. This way, the page and referenced resources can be downloaded using the same connection. This saves a lot of work and time for the web server, client, and the network, considering that establishing and tearing down HTTP connections are expensive operations.

The persistent connection is the default connection of HTTP 1.1. Also, to make it explicit, a browser can send the request header connection with the value keep-alive:

```
connection: keep-alive
```

Chunked Encoding

The consequence of establishing a persistent connection is that the server can send byte streams from multiple resources, and the client can send multiple requests using the same connection. As a result, the sender must send the content length header of each request or response so that the recipient would know how to interpret the bytes. However, often the case is that the sender does not know how many bytes it will send. For example, a servlet container can start sending the response when the first few bytes become available and not wait until all of them ready. This means, there must be a way to tell the recipient how to interpret the byte stream in the case that the content-length header cannot be known earlier.

Even without having to send multiple requests or many responses, a server or a client does not necessarily know how much data it will send. In HTTP 1.0, a server could just leave out the content-length header and keep writing to the connection. When it was finished, it would simply close the connection. In this case, the client would keep reading until it got a -1 as an indication that the end of file had been reached.

HTTP 1.1 employs a special header called transfer-encoding to indicate that the byte stream will be sent in chunks. For every chunk, the length (in hexadecimal) followed by CR/LF is sent prior to the data. A transaction is marked with a zero length chunk. Suppose you want to send the following 38 bytes in 2 chunks, the first with the length of 29 and the second 9.

I'm as helpless as a kitten up a tree.

You would send the following:

```
1D\r\n
I'm as helpless as a kitten u 9\r\n
p a tree.
0\r\n
```

1D, the hexadecimal of 29, indicates that the first chunk consists of 29 bytes. 0\r\n indicates the end of the transaction.

Use of the 100 (Continue) Status

HTTP 1.1 clients may send the Expect: 100-continue header to the server before sending the request body and wait for acknowledgement from the server. This normally happens if the client is going to send a long request body but is not sure that the server is willing to accept it. It would be a waste if the client sent the long body just to find out the server turned it down.

Upon receipt of the Expect: 100-continue header, the server responds with the following 100-continue header if it is willing to or can process the request, followed by two pairs of CRLF characters.

```
HTTP/1.1 100 Continue
```

The server should then continue reading the input stream.

The Connector Interface

A Tomcat connector must implement the org.apache.catalina.Connector interface. Of many methods in this interface, the most important are getContainer, setContainer, createRequest, and createResponse.

setContainer is used to associate the connector with a container. getContainer returns the associated container. createRequest constructs a request object for the incoming HTTP request and createResponse creates a response object.

The org.apache.catalina.connector.http.HttpConnector class is an implementation of the Connector interface and is discussed in the next section, "The HttpConnector Class". Now, take a close look at Figure 4.1 for the UML class diagram of the default connector. Note that the implementation of the Request and Response interfaces have been omitted to keep the diagram simple. The org.apache.catalina prefix has also been omitted from the type names, except for the SimpleContainer class.

Figure 4.1: The default connector class diagram

Therefore, Connector should be read org.apache.catalina.Connector,util.StringManager org.apache.catalina.util.StringManager, etc.

A Connector has one-to-one relationship with a Container. The navigability of the arrow representing the relationship reveals that the Connector knows about the Container but not the other way around. Also note that, unlike in Chapter 3, the relationship between HttpConnector and HttpProcessor is one-to-many.

The HttpConnector Class

You already know how this class works because the simplified version of org.apache.catalina.connector.http.HttpConnector was explained in Chapter 3. It implements org.apache.catalina.Connector (to make it eligible to work with Catalina), java.lang.Runnable (so that its instance can work in its own thread), and org.apache.catalina.Lifecycle. The Lifecycle interface is used to maintain the life cycle of every Catalina component that implements it.

Lifecycle is explained in Chapter 6 and for now you don't have to worry about it except to know this: by implementing Lifecycle, after you have created an instance of HttpConnector, you should call its initialize and start methods. Both methods must only called once during the life time of the component. We will now look at those aspects that are different from the HttpConnector class in Chapter 3: how HttpConnector creates a server socket, how it maintains a pool of HttpProcessor, and how it serves HTTP requests.

Creating a Server Socket

The initialize method of HttpConnector calls the open private method that returns an instance of java.net.ServerSocket and assigns it to serverSocket. However, instead of calling the java.net.ServerSocket constructor, the open method obtains an instance of ServerSocket from a server socket factory. If you want to know the details of this factory, read the ServerSocketFactory interface and the DefaultServerSocketFactory class in the org.apache.catalina.net package. They are easy to understand.

Maintaining HttpProcessor Instances

In Chapter 3, the HttpConnector instance had only one instance of HttpProcessor at a time, so it can only process one HTTP request at a time. In the default connector, the HttpConnector has a pool of HttpProcessor objects and each instance of HttpProcessor has a thread of its own. Therefore, the HttpConnector can serve multiple HTTP requests simultaneously.

The HttpConnector maintains a pool of HttpProcessor instances to avoid creating HttpProcessor objects all the time. The HttpProcessor instances are stored in a java.io.Stack called processors:

```
private Stack processors = new Stack();
```

In HttpConnector, the number of HttpProcessor instances created is determined by two variables: minProcessors and maxProcessors. By default, minProcessors is set to 5 and maxProcessors 20, but you can change their values through the setMinProcessors and setMaxProcessors methods.

```
protected int minProcessors = 5;
private int maxProcessors = 20;
```

Initially, the HttpConnector object creates minProcessors instances of HttpProcessor. If there are more requests than the HttpProcessor instances can serve at a time, the HttpConnector creates more HttpProcessor instances until the number of instances reaches maxProcessors. After this point is reached and there are still not enough HttpProcessor instances, the incoming HTTP requests will be ignored. If you want the HttpConnector to keep creating HttpProcessor instances, set maxProcessors to a negative number. In addition, the curProcessors variable keeps the current number of HttpProcessor instances.

Here is the code that creates an initial number of HttpProcessor instances in the HttpConnector class's start method:

```
while (curProcessors < minProcessors) {
   if ((maxProcessors > 0) && (curProcessors >= maxProcessors))
      break;
   HttpProcessor processor = newProcessor();
   recycle(processor);
}
```

The newProcessor method constructs a new HttpProcessor object and increments curProcessors. The recycle method pushes the HttpProcessor back to the stack.

Each HttpProcessor instance is responsible for parsing the HTTP request line and headers and populates a request object. Therefore, each instance is associated with a request object and a response object. The HttpProcessor class's constructor contains calls to the HttpConnector class's createRequest and createResponse methods.

Serving HTTP Requests

The HttpConnector class has its main logic in its run method, just like in Chapter 3. The run method contains a while loop where the server socket waits for an HTTP request until the HttpConnector is stopped.

```
while (!stopped) {
   Socket socket = null;
   try {
      socket = serverSocket.accept();
      ...
```

For each incoming HTTP request, it obtains an HttpProcessor instance by calling the createProcessor private method.

```
HttpProcessor processor = createProcessor();
```

However, most of the time the createProcessor method does not create a new HttpProcessor object. Instead, it gets one from the pool. If there is still an HttpProcessor instance in the stack, createProcessor pops one. If the stack is empty and the maximum number of HttpProcessor instances has not been exceeded, createProcessor creates one. However, if the maximum number has been reached, createProcessor returns null. If this happens, the socket is simply closed and the incoming HTTP request is not processed.

```
if (processor == null) {
   try {
      log(sm.getString("httpConnector.noProcessor"));
      socket.close();
      }
      ... continue;
```

If createProcessor does not return null, the client socket is passed to the HttpProcessor class's assign method:

```
processor.assign(socket);
```

It's now the HttpProcessor instance's job to read the socket's input stream and parse the HTTP request. An important note is this. The assign method must return straight away and not wait until the HttpProcessor finishes the parsing, so the next incoming HTTP request can be served. Since each HttpProcessor instance has a thread of its own for the parsing, this is not very hard to achieve. You will see how this is done in the next section, "The HttpProcessor Class".

The HttpProcessor Class

The HttpProcessor class in the default connector is the full version of the similarly named class in Chapter 3. You've learned how it worked and in this chapter we're most interested in knowing how the HttpProcessor class makes its assign method asynchronous so that the HttpConnector instance can serve many HTTP requests at the same time.

Note: Another important method of the HttpProcessor class is the private process method which parses the HTTP request and invoke the container's invoke method. We'll have a look at it in the section, "Processing Requests" later in this chapter.

In Chapter 3, the HttpConnector runs in its own thread. However, it has to wait for the currently processed HTTP request to finish before it can process the next: request. Here is part of the HttpConnector class's run method in Chapter 3:

```
public void run() {
   ...
   while (!stopped) {
      Socket socket = null;
      try {
         socket = serversocket.accept();
         } catch (Exception e) { continue;}
// Hand this socket off to an Httpprocessor
   }
}
```

The process method of the HttpProcessor class in Chapter 3 is synchronous. Therefore, its run method waits until the process method finishes before accepting another request.

In the default connector, however, the HttpProcessor class implements java.lang.Runnable and each instance of HttpProcessor runs in its own thread, which we call the "processor thread". For each HttpProcessor instance the HttpConnector creates, its start method is called, effectively starting the "processor thread" of the HttpProcessor instance. Listing 4.1 presents the run method in the HttpProcessor class in the default connector:

Listing 4.1: The HttpProcessor class's run method.

```
public void run() {
// Process requests until we receive a shutdown signal
    while (!stopped) {
        // Wait for the next socket to be assigned
        Socket socket = await();
        if (socket == null)
            continue;
            // Process the request from this socket
            try {
                process(socket);
            }catch (Throwable t) {
                log("process.invoke", t);
        }
        // Finish up this request
        connector.recycle(this);
    }
        // Tell threadStop() we have shut ourselves down successfully
        synchronized (threadSync) { threadSync.notifyAll();
} }
```

The while loop in the run method keeps going in this order: gets a socket, process it, calls the connector's recycle method to push the current HttpProcessor instance back to the stack. Here is the HttpConenctor class's recycle method:

```
void recycle(HttpProcessor processor) {
    processors.push(processor);
}
```

Notice that the while loop in the run method stops at the await method. The await method holds the control flow of the "processor thread" until it gets a new socket from the HttpConnector. In other words, until the HttpConnector calls the HttpProcessor instance's assign method. However, the await method runs on a different thread than the assign method. The assign method is called from the run method of the HttpConnector. We name the thread that the HttpConnector instance's run method runs on the "connector thread". How does the assign method tell the await method that it has been called? By using a boolean called available, and by using the wait and notifyAll methods of java.lang.Object.

Note The wait method causes the current thread to wait until another thread invokes the notify or the notifyAll method for this object.

Here is the HttpProcessor class's assign and await methods:

```
synchronized void assign(Socket socket) {
// Wait for the processor to get the previous socket
    while(available) {
        try {
            wait();
        }catch (InterruptedException e) { }
    }
    // Store the newly available Socket and notify our thread
    this.socket = socket;
    available = true;
    notifyAll();
    ...
}
```

```
private synchronized Socket await() {
    // Wait for the Connector to provide a new Socket
    while (!available) { try {
        wait();
    }catch (InterruptedException e) { } }
    // Notify the Connector that we have received this Socket
    Socket socket = this.socket;
    available = false;
    notifyAll();
    if ((debug >= 1) && (socket != null))
        log(" The incoming request has been awaited");
    return (socket);
}
```

The program flows of both methods are summarized in Table 4.1.

Table 4.1: Summary of the await and assign method

```
The processor thread (the await method) The connector thread (the assign method)
while (!available) { while (available) { wait(); wait(); }}
Socket socket = this.socket; available = false;
notifyAll();
return socket; // to the run // method
this.socket = socket; available = true; notifyAll();
...
```

Initially, when the "processor thread" has just been started, available is false, so the thread waits inside the while loop (see Column 1 of Table 4.1). It will wait until another thread calls notify or notifyAll. This is to say that calling the wait method causes the "processor thread" to pause until the "connector thread" invokes the notifyAll method for the HttpProcessor instance.

Now, look at Column 2. When a new socket is assigned, the "connector thread" calls the HttpProcessor's assign method. The value of available is false, so the while loop is skipped and the socket is assigned to the HttpProcessor instance's socket variable:

```
this.socket = socket;
```

The "connector thread" then sets available to true and calls notifyAll. This wakes up the processor thread and now the value of available is true so the program controls goes out of the while loop: assigning the instance's socket to a local variable, sets available to false, calls notifyAll, and returns the socket, which eventually causes the socket to be processed.

Why does the await method need to use a local variable (socket) and not return the instance's socket variable? So that the instance's socket variable can be assigned to the next incoming socket before the current socket gets processed completely.

Why does the await method need to call notifyAll? Just in case another socket arrives when the value of available is true. In this case, the "connector thread" will stop inside the assign method's while loop until the nofifyAll call from the "processor thread" is received.

Request Objects

The HTTP Request object in the default connector is represented by the org.apache.catalina.Request interface. This interface is directly implemented by the RequestBase class, which is the parent of HttpRequest. The ultimate implementation is HttpRequestImpl, which extends HttpRequest. Like in Chapter 3, there are facade classes: RequestFacade and HttpRequestFacade. The UML diagram for the Request interface and its implementation classes is given in Figure 4.2. Note that except for the types belonging to the javax.servlet and javax.servlet.http packages, the prefix org.apache.catalina has been omitted.

Figure 4.2: The Request interface and related types

If you understand about the request object in Chapter 3, you should not have problems understanding the diagram.

Response Objects

The UML diagram of the Response interface and its implementation classes is given in Figure 4.3.

Figure 4.3: The Response interface and its implementation classes

Processing Requests

At this point, you already understand about the request and response objects and how the HttpConnector object creates them. Now is the last bit of the process. In this section we focus on the process method of the HttpProcessor class, which is called by the HttpProcessor class's run method after a socket is assigned to it. The process method does the following:

- parse the connection
- parse the request
- parse headers

Each operation is discussed in the sub-sections of this section after the process method is explained.

The process method uses the boolean ok to indicate that there is no error during the process and the boolean finishResponse to indicate that the finishResponse method of the Response interface should be called.

```
boolean ok = true;
boolean finishResponse = true;
```

In addition, the process method also uses the instance boolean variables keepAlive, stopped, and http11.keepAlive indicates that the connection is persistent, stopped indicates that the HttpProcessor instance has been stopped by the connector so that the process method should also stop, and http11 indicates that the HTTP request is coming from a web client that supports HTTP 1.1.

Like in Chapter 3, a SocketInputStream instance is used to wrap the socket's input stream. Note that, the constructor of SocketInputStream is also passed the buffer size from the connector, not from a local variable in the HttpProcessor class. This is because HttpProcessor is not accessible by the user of the default connector. By putting the buffer size in the Connector interface, this allows anyone using the connector to set the buffer size.

```
SocketInputStream input = null;
OutputStream output = null;
// Construct and initialize the objects we will need
try {
    input = new SocketInputStream(socket.getInputstream(), connector.getBufferSize());
} catch (Exception e) {
    ok = false;
}
```

Then, there is a while loop which keeps reading the input stream until the HttpProcessor is stopped, an exception is thrown, or the connection is closed.

```
keepAlive = true;
while (!stopped && ok && keepAlive) {
... }
```

Inside the while loop, the process method starts by setting finishResponse to true and obtaining the output stream and performing some initialization to the request and response objects.

```
finishResponse = true; try {
request.setStream(input);
request.setResponse(response);
output = socket.getOutputStream();
response.setStream(output);
response.setRequest(request);
((HttpServletResponse) response.getResponse()).setHeader
("Server", SERVER_INFO);
} catch (Exception e) {
    log("process.create", e);
    //logging is discussed in Chapter 7
    ok = false;
}
```

Afterwards, the process method start parsing the incoming HTTP request by calling the parseConnection, parseRequest, and parseHeaders methods, all of which are discussed in the sub-sections in this section.

```
try {
    if (ok) {
        parseConnection(socket);
        parseRequest(input, output);
    if (!request.getRequest().getProtocol()
.startsWith("HTTP/0"))
        parseHeaders(input);
```

The parseConnection method obtains the value of the protocol, which can be HTTP 0.9, HTTP 1.0 or HTTP 1.1. If the protocol is HTTP 1.0, the keepAlive boolean is set to false because HTTP 1.0 does not support persistent connections. The parseHeaders method will set the sendAck boolean to true if an Expect: 100-continue header is found in the HTTP request.

If the protocol is HTTP 1.1, it will respond to the Expect: 100-continue header, if the web client sent this header, by calling the ackRequest method. It will also check if chunking is allowed.

```
if (http11) {
    // Sending a request acknowledge back to the client if
    // requested.
    ackRequest(output);
    // If the protocol is HTTP/1.1, chunking is allowed.
    if (connector.isChunkingAllowed())
        response.setAllowChunking(true);
}
```

The ackRequest method checks the value of sendAck and sends the following string if sendAck is true:

```
HTTP/1.1 100 Continue\r\n\r\n
```

During the parsing of the HTTP request, one of the many exceptions might be thrown. Any exception will set ok or finishResponse to false. After the parsing, the process method passes the request and response objects to the container's invoke method.

```
try {
    ((HttpServletResponse) response).setHeader
("Date", FastHttpDateFormat.getCurrentDate());
    if (ok) {
    connector.getContainer().invoke(request, response);
    }
}
```

Afterwards, if finishResponse is still true, the response object's finishResponse method and the request's object finishRequest methods are called, and the output is flushed.

```
if (finishResponse) {
   ...
   response.finishResponse();
   ...
   request.finishRequest();
   ...
   output.flush();
```

The last part of the while loop checks if the response's Connection header has been set to close from inside the servlet or if the protocol is HTTP 1.0. If this is the case, keepAlive is set to false. Also, the request and response objects are then recycled.

```
if ( "close".equals(response.getHeader("Connection")) ) {
   keepAlive = false;
}
   // End of request processing
   status = Constants.PROCESSOR_IDLE;
   // Recycling the request and the response objects
   request.recycle();
   response.recycle();
}
```

At this stage, the while loop will start from the beginning if keepAlive is true, there is no error during the previous parsing and from the container's invoke method, or the HttpProcessor instance has not been stopped. Otherwise, the shutdownInput method is called and the socket is closed.

```
try {
   shutdownInput(input);
   socket.close();
} ...
```

The shutdownInput method checks if there are any unread bytes. If there are, it skips those bytes.

Parsing the Connection

The parseConnection method obtains the Internet address from the socket and assigns it to the HttpRequestImpl object. It also checks if a proxy is used and assigns the socket to the request object. The parseConnection method is given in Listing 4.2.

Listing 4.2: The parseConnection method

```
private void parseConnection(Socket socket) throws IOException, ServletException {
   if (debug >= 2)
   log(" parseConnection: address=" + socket.getInetAddress() + ", port=" + connector.getPort());
   ((HttpRequestImpl) request).setInet(socket.getInetAddress());
   if (proxyPort != 0)
      request.setServerPort(proxyPort);
   else
      request.setServerPort(serverPort);
      request.setSocket(socket);
}
```

Parsing the Request

The parseRequest method is the full version of the similar method in Chapter 3. If you understand Chapter 3 well, you should be able to understand how this method works by reading the method.

Parsing Headers

The parseHeaders method in the default connector uses the HttpHeader and DefaultHeaders classes in the org.apache.catalina.connector.http package. The HttpHeader class represents an HTTP request header. Instead of working with strings like in Chapter 3, the HttpHeader class uses character arrays to avoid expensive string operations. The DefaultHeaders class is a final class containing the standard HTTP request headers in character arrays:

```
static final char[] AUTHORIZATION_NAME = "authorization".toCharArray();
static final char[] ACCEPT_LANGUAGE_NAME = "accept-language".toCharArray();
static final char[] COOKIE_NAME = "cookie".toCharArray();
...
```

The parseHeaders method contains a while loop that keeps reading the HTTP request until there is no more header to read. The while loop starts by calling the allocateHeader method of the request object to obtain an instance of empty HttpHeader. The instance is passed to the readHeader method of SocketInputStream.

```
HttpHeader header = request.allocateHeader();
// Read the next header
input.readHeader(header);
```

If all headers have been read, the readHeader method will assign no name to the HttpHeader instance, and this is time for the parseHeaders method to return.

```
if (header.nameEnd == 0) {
    if (header.valueEnd == 0) {
        return;
    } else {
        throw new ServletException
        (sm.getString("httpProcessor.parseHeaders.colon"));
    }
}
```

If there is a header name, there must also be a header value:

```
String value = new String(header.value, 0, header.valueEnd);
```

Next, like in Chapter 3, the parseHeaders method compares the header name with the standard names in DefaultHeaders. Note that comparison is performed between two character arrays, not between two strings.

```
if (header.equals(DefaultHeaders.AUTHORIZATION_NAME)) {
    request.setAuthorization(value);
}
else if (header.equals(DefaultHeaders.ACCEPT_LANGUAGE_NAME)) {
    parseAcceptLanguage(value);
}
else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    // parse cookie
}
else if (header.equals(DefaultHeaders.CONTENT_LENGTH_NAME)) {
    // get content length
}
else if (header.equals(DefaultHeaders.CONTENT_TYPE_NAME)) {
    request.setContentType(value);
}
else if (header.equals(DefaultHeaders.HOST_NAME)) {
    // get host name
}
else if (header.equals(DefaultHeaders.CONNECTION_NAME)) {
    if (header.valueEquals(DefaultHeaders.CONNECTION_CLOSE_VALUE)) {
        keepAlive = false;
        response.setHeader("Connection", "close");
        }

}
else if (header.equals(DefaultHeaders.EXPECT_NAME)) {
    if (header.valueEquals(DefaultHeaders.EXPECT_100_VALUE))
        sendAck = true;
    else
        throw new ServletException(sm.getstring
("httpProcessor.parseHeaders.unknownExpectation"));
}
else if (header.equals(DefaultHeaders.TRANSFER_ENCODING_NAME)) {
    //request.setTransferEncoding(header);
}

request.nextHeader();
```

The Simple Container Application

The main purpose of the application in this chapter is to show how to use the default connector. It consists of two classes: ex04.pyrmont.core.SimpleContainer and ex04 pyrmont.startup.Bootstrap. The SimpleContainer class implements org.apache.catalina.container so that it can be associated with the connector. The Bootstrap class is used to start the application, we have removed the connector module and the ServletProcessor and StaticResourceProcessor classes in the application accompanying Chapter 3, so you cannot request a static page.

The SimpleContainer class is presented in Listing 4.3.

Listing 4.3: The SimpleContainer class

```
package ex04.pyrmont.core;

import java.beans.PropertyChangeListener;
import java.net.URL;
import java.net.URLClassLoader;
import java.net.URLStreamHandler;
import java.io.File;
import java.io.IOException;
import javax.naming.directory.DirContext;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.catalina.Cluster;
import org.apache.catalina.Container;
import org.apache.catalina.ContainerListener;
import org.apache.catalina.Loader;
import org.apache.catalina.Logger;
import org.apache.catalina.Manager;
import org.apache.catalina.Mapper;
import org.apache.catalina.Realm;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
```

```
public class SimpleContainer implements Container {
  public static final String WEB_ROOT = System.getProperty("user.dir") + File.separator + "webroot";
  public SimpleContainer() { } public String getInfo() {
    return null;
}
  public Loader getLoader() {
    return null;
  }
  public void setLoader(Loader loader) { }
  public Logger getLogger() {
    return null;
  }
  public void setLogger(Logger logger) { }
  public Manager getManager() { return null; }
  public void setManager(Manager manager) { }
  public Cluster getCluster() { return null; }
  public void setCluster(Cluster cluster) { }
  public String getName() { return null; }
  public void setName(String name) { }
  public Container getParent() { return null; }
  public void setParent(Container container) { }
  public ClassLoader getParentClassLoader() { return null;}
  public void setParentClassLoader(ClassLoader parent) { }
  public Realm getRealm() { return null; }
  public void setRealm(Realm realm) { }
  public DirContext getResources() { return null; }
  public void setResources(DirContext resources) { }
  public void addChild(Container child) { }
  public void addContainerListener(ContainerListener listener) { }
  public void addMapper(Mapper mapper) { }
  public void addPropertyChangeListener(PropertyChangeListener listener) { }
  public Container findchild(String name) { return null; }
  public Container[] findChildren() { return null;}
  public ContainerListener[] findContainerListeners() { return null; }
  public Mapper findMapper(String protocol) { return null;}
  public Mapper[] findMappers() { return null; }
  public void invoke(Request request, Response response)throws IoException, ServletException {
    string servletName = ( (Httpservletrequest)
request).getRequestURI();
    servletName = servletName.substring(servletName.lastIndexof("/") +
1);
    URLClassLoader loader = null;
    try {
      URL[] urls = new URL[1];
      URLStreamHandler streamHandler = null;
      File classpath = new File(WEB_ROOT);
    }
    string repository = (new URL("file",null,
    classpath.getCanonicalpath() + File.separator)).toString();
    urls[0] = new URL(null, repository, streamHandler);
    loader = new URLClassLoader(urls);
  }catch (IOException e) {
      System.out.println(e.toString() );
      Class myClass = null;
  try {
      myClass = loader.loadclass(servletName);
    } catch (classNotFoundException e) {
      System.out.println(e.toString());
    }
    servlet servlet = null;
    try {
      servlet = (Servlet) myClass.newInstance();
      servlet.service((HttpServletRequest) request,
(HttpServletResponse) response);
    }catch (Exception e) {
      System.out.println(e.toString());
    }catch (Throwable e) {
      System.out.println(e.toString());
    }
  }
  public Container map(Request request, boolean update) { return null; }
  public void removeChild(Container child) { }
  public void removeContainerListener(ContainerListener listener) { }
  public void removeMapper(Mapper mapper) { }
  public void removoPropertyChangeListener(
PropertyChangeListener listener) { }
  }
```

I only provide the implementation of the invoke method in the SimpleContainer class because the default connector will call this method. The invoke method creates a class loader, loads the servlet class, and calls its service method. This method is very similar to the process method in the ServletProcessor class in Chapter 3.

The Bootstrap class is given in Listing 4.4.

Listing 4.4: The ex04.pyrmont.startup.Bootstrap class

```
package ex04.pyrmont.startup;

import ex04.pyrmont.core.simplecontainer;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap {
  public static void main(string[] args) {
    HttpConnector connector = new HttpConnector(); SimpleContainer
    container = new SimpleContainer();
    connector.setContainer(container);
    try {
      connector.initialize();
      connector.start();
    // make the application wait until we press any key.
    System in.read();
    }catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The main method of the Bootstrap class constructs an instance of org.apache.catalina.connector.http.HttpConnector and a SimpleContainer instance. It then associates the connector with the container by calling the connector's setContainer method, passing the container. Next, it calls the connector's initialize and start methods. This will make the connector ready for processing any HTTP request on port 8080.

You can terminate the application by pressing a key on the console.

Running the Application

To run the application in Windows, from the working directory, type the following:

```
java -classpath ./lib/servlet.jar;./ ex04.pyrmont.startup.Bootstrap
```

In Linux, you use a colon to separate two libraries.

```
java -classpath ./lib/servlet.jar:./ ex04.pyrmont.startup.Bootstrap
```

You can invoke PrimitiveServlet and ModernServlet the way you did in Chapter 3. Note that you cannot request the index.html file because there is no processor for static resources.

Summary

This chapter showed what it takes to build a Tomcat connector that can work with Catalina. It dissected the code of Tomcat 4's default connector and built a small application that used the connector. All applications in the upcoming chapters use the default connector.

OverView

A container is a module that processes the requests for a servlet and populates the response objects for web clients. A container is represented by the org.apache.catalina.Container interface and there are four types of containers: Engine, Host, Context, and Wrapper. This chapter covers Context and Wrapper and leaves Engine and Host to Chapter 13. This chapter starts with the discussion of the Container interface, followed by the pipelining mechanism in a container. It then looks at the Wrapper and Context interfaces. Two applications conclude this chapter by presenting a simple wrapper and a simple context respectively.

The Container Interface

A container must implement org.apache.catalina.Container. As you have seen in Chapter 4, you pass an instance of Container to the setContainer method of the connector, so that the connector can call the container's invoke method. Recall the following code from the Bootstrap class in the application in Chapter 4.

```
HttpConnector connector = new HttpConnector();
SimpleContainer container = new SimpleContainer();
connector.setContainer(container);
```

The first thing to note about containers in Catalina is that there are four types of containers at different conceptual levels:

- Engine. Represents the entire Catalina servlet engine.
- Host. Represents a virtual host with a number of contexts.
- Context. Represents a web application. A context contains one or more wrappers.
- Wrapper. Represents an individual servlet.

Each conceptual level above is represented by an interface in the org.apache.catalina package. These interfaces are Engine, Host, Context, and Wrapper. All the four extends the Container interface. Standard implementations of the four containers are StandardEngine, StandardHost, StandardContext, and StandardWrapper, respectively, all of which are part of the org.apache.catalina.core package.

Figure 5.1 shows the class diagram of the Container interface and its sub-interfaces and implementations. Note that all interfaces are part of the org.apache.catalina package and all classes are part of the org.apache.catalina.core package.

Figure 5.1: The class diagram of Container and its related types Note All implementation classes derive from the abstract class ContainerBase.

A functional Catalina deployment does not need all the four types of containers. For example, the container module in this chapter's first application consists of only a wrapper. The second application is a container module with a context and a wrapper. Neither host nor engine is needed in the applications accompanying this chapter.

A container can have zero or more child containers of the lower level. For instance, a context normally has one or more wrappers and a host can have zero or more contexts. However, a wrapper, being the lowest in the 'hierarchy', cannot contain a child container. To add a child container to a container, you use the Container interface's addChild method whose signature is as follows.

```
public void addChild(Container child);
```

To remove a child container from a container, call the Container interface's removeChild method. The remove method's signature is as follows.

```
public void removeChild(Container child);
```

In addition, the Container interface supports the finding of a child container or a collection of all child containers through the findChild and findChildren methods. The signatures of both methods are the following.

```
public Container findChild(String name);
public Container[] findChildren();
```

A container can also contain a number of support components such as Loader, Logger, Manager, Realm, and Resources. We will discuss these components in later chapters. One thing worth noting here is that the Container interface provides the get and set methods for associating itself with those components. These methods include getLoader and setLoader, getLogger and setLogger, getManager and setManager, getRealm and setRealm, and getResources and setResources.

More interestingly, the Container interface has been designed in such a way that at the time of deployment a Tomcat administrator can determine what a container performs by editing the configuration file (server.xml). This is achieved by introducing a pipeline and a set of valves in a container, which we will discuss in the next section, "Pipelining Tasks".

Note: The Container interface in Tomcat 4 is slightly different from that in Tomcat 5. For example, in Tomcat 4 this interface has a map method, which no longer exists in the Container interface in Tomcat 5.

Pipelining Tasks

This section explains what happens when a container's invoke method is called by the connector. This section then discusses in the sub-sections the four related interfaces in the org.apache.catalina package: Pipeline, Valve, ValveContext, and Contained.

A pipeline contains tasks that the container will invoke. A valve represents a specific task. There is one basic valve in a container's pipeline, but you can add as many valves as you want. The number of valves is defined to be the number of additional valves, i.e. not including the basic valve. Interestingly, valves can be added dynamically by editing Tomcat's configuration file (server.xml). Figure 5.2 shows a pipeline and its valves.

Figure 5.2: Pipeline and valves

If you understand servlet filters, it is not hard to imagine how a pipeline and its valve work. A pipeline is like a filter chain and each valve is a filter. Like a filter, a valve can manipulate the request and response objects passed to it. After a valve finishes processing, it calls the next valve in the pipeline. The basic valve is always called the last.

A container can have one pipeline. When a container's invoke method is called, the container passes processing to its pipeline and the pipeline invokes the first valve in it, which will then invoke the next valve, and so on, until there is no more valve in the pipeline. You might imagine that you could have the following pseudo code inside the pipeline's invoke method:

```
// invoke each valve added to the pipeline
for (int n=0; n < valves.length; n++) {
  valve[n].invoke( ... );
}
// then, invoke the basic valve
basicValve.invoke( ... );
```

However, the Tomcat designer chose a different approach by introducing the org.apache.catalina.ValveContext interface. Here is how it works.

A container does not hard code what it is supposed to do when its invoke method is called by the connector. Instead, the container calls its pipeline's invoke method. The Pipeline interface's invoke method has the following signature, which is exactly the same as the invoke method of the Container interface.

```
public void invoke(Request request, Response response) throws IOException, ServletException;
```

Here is the implementation of the Container interface's invoke method in the *org.apache.catalina.core.ContainerBase class.*

```
public void invoke(Request request, Response response) throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

where pipeline is an instance of the Pipeline interface inside the container.

Now, the pipeline has to make sure that all the valves added to it as well as its basic valve must be invoked once. The pipeline does this by creating an instance of the ValveContext interface. The ValveContext is implemented as an inner class of the pipeline so that the ValveContext has access to all members of the pipeline. The most important method of the ValveContext interface is invokeNext:

```
public void invokeNext(Request request, Response response) throws IOException, ServletException
```

After creating an instance of ValveContext, the pipeline calls the invokeNext method of the ValveContext. The ValveContext will first invoke the first valve in the pipeline and the first valve will invoke the next valve before the first valve does its task. The ValveContext passes itself to each valve so that the valve can call the invokeNext method of the ValveContext. Here is the signature of the invoke method of the Valve interface.

```
public void invoke(Request request, Response response, ValveContext ValveContext) throws IOException, ServletException
```

An implementation of a valve's invoke method will be something like the following.

```
public void invoke(Request request, Response response,
ValveContext valveContext) throws IOException, ServletException {
  // Pass the request and response on to the next valve in our pipeline   valveContext.invokeNext(request, response);
  // now perform what this valve is supposed to do
  ...
}
```

The org.apache.catalina.core.StandardPipeline class is the implementation of Pipeline in all containers. In Tomcat 4, this class has an inner class called StandardPipelineValveContext that implements the ValveContext interface. Listing 5.1 presents the StandardPipelineValveContext class.

Listing 5.1: The StandardPipelineValveContext class in Tomcat 4

```
protected class StandardPipelineValveContext implements ValveContext {
  protected int stage = 0;
  public String getInfo() {
    return info;
  }
  public void invokeNext(Request request, Response response) throws IOException, ServletException {
    int subscript = stage;
    stage = stage + 1;
    // Invoke the requested Valve for the current request thread
    if (subscript < valves.length) {
      valves[subscript].invoke(request, response, this);
    }else if ((subscript == valves.length) && (basic != null)) {
      basic.invoke(request, response, this);
    }else {
      throw new ServletException (sm.getString("standardPipeline.noValve"));
  } }
}
```

The invokeNext method uses subscript and stage to remember which valve is being invoked. When first invoked from the pipeline's invoke method, the value of subscript is 0 and the value of stage is 1. Therefore, the first valve (array index 0) is invoked. The first valve in the pipeline receives the ValveContext instance and invokes its invokeNext method. This time, the value of subscript is 1 so that the second valve is invoked, and so on.

When the invokeNext method is called from the last valve, the value of subscript is equal to the number of valves. As a result, the basic valve is invoked.

Tomcat 5 removes the StandardPipelineValveContext class from StandardPipeline and instead relies on the org.apache.catalina.core.StandardValveContext class, which is presented in Listing 5.2.

Listing 5.2: The StandardValveContext class in Tomcat 5

```
package org.apache.catalina.core;

import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.util.StringManager;

public final class StandardValveContext implements ValveContext {
  protected static StringManager sm =
StringManager.getManager(Constants.Package);
  protected String info =
"org.apache.catalina.core.StandardValveContext/1.0";
  protected int stage = 0;
  protected Valve basic = null;
  protected Valve valves[] = null;
  public String getInfo() {
    return info;
  }
  public final void invokeNext(Request request, Response response) throws IOException, ServletException {
    int subscript = stage;
    stage = stage + 1;
    // Invoke the requested Valve for the current request thread
    if (subscript < valves.length) {
      valves[subscript].invoke(request, response, this); }
    else if ((subscript == valves.length) && (basic != null)) {
      basic.invoke(request, response, this);
    }else {
      throw new ServletException (sm.getString("standardPipeline.noValve"));
    } }
    void set(Valve basic, Valve valves[]) {
      stage = 0;
      this.basic = basic;
      this.valves = valves;
  }
}
```

Can you see the similarities between the StandardPipelineValveContext class in Tomcat 4 and the StandardValveContext class in Tomcat 5?

We will now explain the Pipeline, Valve, and ValveContext interfaces in more detail. Also discussed is the org.apache.catalina.Contained interface that a valve class normally implements.

The Pipeline Interface

The first method of the Pipeline interface that we mentioned was the invoke method, which a container calls to start invoking the valves in the pipeline and the basic valve. The Pipeline interface allows you to add a new valve through its addValve method and remove a valve by calling its removeValve method. Finally, you use its setBasic method to assign a basic valve to a pipeline and its getBasic method to obtain the basic valve. The basic valve, which is invoked last, is responsible for processing the request and the corresponding response. The Pipeline interface is given in Listing 5.3.

Listing 5.3: The Pipeline interface

```
package org.apache.catalina; import java.io.IOException;
import javax.servlet.ServletException;

public interface Pipeline {
  public Valve getBasic();
  public void setBasic(Valve valve);
  public void addValve(Valve valve);
  public Valve[] getValves();
  public void invoke(Request request, Response response)
throws IOException, ServletException;
  public void removeValve(Valve valve);
```

The Valve Interface

The Valve interface represents a valve, the component responsible for processing a request. This interface has two methods: invoke and getInfo. The invoke method has been discussed above. The getInfo method returns information about the valve implementation. The Valve interface is given in Listing 5.4.

Listing 5.4: The Valve interface

```
package org.apache.catalina;

import java.io.IOException;
import javax.servlet.ServletException;

public interface Valve {
  public String getInfo();
  public void invoke(Request request, Response response,
ValveContext context) throws IOException, ServletException;
}
```

The ValveContext Interface

This interface has two methods: the invokeNext method, which has been discussed above, and the getInfo method, which returns information about the ValveContext implementation. The ValveContext interface is given in Listing 5.5.

Listing 5.5: The ValveContext interface

```
package org.apache.catalina;

import java.io.IOException;
import javax.servlet.ServletException;

public interface ValveContext {
  public String getInfo();
  public void invokeNext(Request request, Response response)
throws IOException, ServletException;
}
```

The Contained Interface

A valve class can optionally implement the org.apache.catalina.Contained interface. This interface specifies that the implementing class is associated with at most one container instance. The Contained interface is given in Listing 5.6.

Listing 5.6: The Contained interface

```
package org.apache.catalina;
public interface Contained {
public Container getContainer();
public void setContainer(Container container);
```

he Wrapper Interface

The org.apache.catalina.Wrapper interface represents a wrapper. A wrapper is a container representing an individual servlet definition. The Wrapper interface extends Container and adds a number of methods. Implementations of Wrapper are responsible for managing the servlet life cycle for their underlying servlet class, i.e. calling the init, service, and destroy methods of the servlet. Since a wrapper is the lowest level of container, you must not add a child to it. A wrapper throws an IllegalArgumantException if its addChild method is called.

Important methods in the Wrapper interface include allocate and load. The allocate method allocates an initialed instance of the servlet the wrapper represents. The allocate method must also take into account whether or not the servlet implements the javax.servlet.SingleThreadModel interface, but we will discuss this later in Chapter 11. The load method loads and initializes an instance of the servlet the wrapper represents. The signatures of the allocate and load methods are as follows.

```
public javax.servlet.Servlet allocate() throws
javax.servlet.ServletException;
public void load() throws javax.servlet.ServletException;
```

The other methods will be covered in Chapter 11 when we discuss the org.apache.catalina.core.StandardWrapper class.

The Context Interface

A context is a container that represents a web application. A context usually has one or more wrappers as its child containers.

Important methods include addWrapper, createWrapper, etc. This interface will be covered in more detail in Chapter 12.

The Wrapper Application

This application demonstrates how to write a minimal container module. The core class of this application is ex05.pyrmont.core.SimpleWrapper, an implementation of the Wrapper interface. The SimpleWrapper class contains a Pipeline (implemented by the ex05.pyrmont.core.SimplePipeline class) and uses a Loader (implemented by the ex05.pyrmont.core.SimpeLoader) to load the servlet. The Pipeline contains a basic valve (ex05.pyrmont.core.SimpleWrapperValve) and two additional valves (ex05.pyrmont.core.ClientIPLoggerValve and ex05.pyrmont.core.HeaderLoggerValve). The class diagram of the application is given in Figure 5.3.

Figure 5.3: The Class Diagram of the Wrapper Application Note The container uses Tomcat 4's default connector.

The wrapper wraps the ModernServlet that you have used in the previous chapters. This application proves that you can have a servlet container consisting only of one wrapper. All classes are not fully developed, implementing only methods that must be present in the class. Let's now look at the classes in detail.

ex05.pyrmont.core.SimpleLoader

The task of loading servlet classes in a container is assigned to a Loader implementation. In this application, the SimpleLoader class is that implementation. It knows the location of the servlet class and its getClassLoader method returns a java.lang.ClassLoader instance that searches the servlet class location. The SimpleLoader class declares three variables. The first is WEB_ROOT, which points to the directory where the servlet class is to be found.

```
public static final String WEB_ROOT = System.getProperty("user.dir") + File.separator + "webroot";
```

The other two variables are object references of type ClassLoader and Container:

```
ClassLoader classLoader = null;
Container container = null;
```

The SimpleLoader class's constructor initializes the class loader so that it is ready to be returned to the SimpleWrapper instance.

```
public SimpleLoader() {
  try {
      URL[] urls = new URL[l];
      URLStreamHandler streamHandler = null;
      File classPath = new File(WEB_ROOT);
      String repository = (new URL("file", null,
classPath.getCanonicalPath() + File.separator)).toString() ;
      urls[0] = new URL(null, repository, streamHandler);
      classLoader = new URLClassLoader(urls);
    }catch (IOException e) {
      System.out.println(e.toString() ); }
}
```

The code in the constructor has been used to initialize class loaders in the applications in previous chapters and won't be explained again.

The container variable represents the container associated with this loader.

Note Loaders will be discussed in detail in Chapter 8.

ex05.pyrmont.core.SimplePipeline

The SimplePipeline class implements the org.apache.catalina.Pipeline interface. The most important method in this class is the invoke method, which contains an inner class called SimplePipelineValveContext. The SimplePipelineValveContext implements the org.apache.catalina.ValveContext interface and has been explained in the section "Pipelining Tasks" above.

ex05.pyrmont.core.SimpleWrapper

This class implements the org.apache.catalina.Wrapper interface and provides implementation for the allocate and load methods. Among others, this class declares the following variables:

```
private Loader loader;
protected Container parent = null;
```

The loader variable is a Loader that is used to load the servlet class. The parent variable represents a parent container for this wrapper. This means that this wrapper can be a child container of another container, such as a Context.

Pay special attention to its getLoader method, which is given in Listing 5.7.

Listing 5.7: The SimpleWrapper class's getLoader method

```
public Loader getLoader() {
  if (loader != null)
    return (loader);
  if (parent != null)
    return (parent.getLoader());

  return (null);
}
```

The getLoader method returns a Loader that is used to load a servlet class. If the wrapper is associated with a Loader, this Loader will be returned. If not, it will return the Loader of the parent container. If no parent is available, the getLoader method returns null.

The SimpleWrapper class has a pipeline and sets a basic valve for the pipeline. You do this in the SimpleWrapper class's constructor, given in Listing 5.8. **Listing 5.8: The SimpleWrapper class's constructor**

```
public SimpleWrapper() {
  pipeline.setBasic(new SimpleWrapperValve());
}
```

Here, pipeline is an instance of SimplePipeline as declared in the class:

```
private SimplePipeline pipeline = new SimplePipeline(this);
```

ex05.pyrmont.core.SimpleWrapperValve

The SimpleWrapperValve class is the basic valve that is dedicated to processing the request for the SimpleWrapper class. It implements the org.apache.catalina.Valve interface and the org.apache.catalina.Contained interface. The most important method in the SimpleWrapperValve is the invoke method, given in Listing 5.9.

Listing 5.9: The SimpleWrapperValve class's invoke method

```
public void invoke(Request request, Response response, ValveContext valveContext)
throws IOException, ServletException {
  SimpleWrapper wrapper = (SimpleWrapper) getContainer(); ServletRequest sreq = request.getRequest();
  ServletResponse sres = response.getResponse();
  Servlet servlet = null;
  HttpServletRequest hreq = null;
  if (sreq instanceof HttpServletRequest)
    hreq = (HttpServletRequest) sreq;
  HttpServletResponse hres = null;
  if (sres instanceof HttpServletResponse)
    hres = (HttpServletResponse) sres;
  // Allocate a servlet instance to process this request try {
  servlet = wrapper.allocate();
  if (hres!=null && hreq!=null) {
    servlet.service(hreq, hres);
  }else {
    servlet.service(sreq, sres);
  } }catch (ServletException e) {
} }
```

Because SimpleWrapperValve is used as a basic valve, its invoke method does not need to call the invokeNext method of the ValveContext passed to it. The invoke method calls the allocate method of the SimpleWrapper class to obtain an instance of the servlet the wrapper represents. It then calls the servlet's service method. Notice that the basic valve of the wrapper's pipeline invokes the servlet's service method, not the wrapper itself.

ex05.pyrmont.valves.ClientIPLoggerValve

The ClientIPLoggerValve class is a valve that prints the client's IP address to the console. This class is given in Listing 5.10.

Listing 5.10: The ClientIPLoggerValve class

```
package ex05.pyrmont.valves;

import java.io.IOException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class ClientIPLoggerValve implements Valve, Contained {
  protected Container container;
  public void invoke(Request request, Response response,
ValveContext valveContext) throws IOException, ServletException {
    // Pass this request on to the next valve in our pipeline
    valveContext.invokeNext(request, response);
    System.out.println("Client IP Logger Valve");
    ServletRequest sreq = request.getRequest();
    System.out.println(sreq.getRemoteAddr());

    System, out.println("-----------------------------------");
    }
  public String getInfo() { return null; }
  public Container getContainer() { return container; }
  public void setContainer(Container container) {
    this.container = container;
  }
}
```

Pay attention to the invoke method. The first thing the invoke method does is call the invokeNext method of the valve context to invoke the next valve in the pipeline, if any. It then prints a few lines of string including the output of the getRemoteAddr method of the request object.

ex05.pyrmont.valves.HeaderLoggerValve

This class is very similar to the ClientIPLoggerValve class. The HeaderLoggerValve class is a valve that prints the request header to the console. This class is given in Listing 5.11.

Listing 5.11: The HeaderLoggerValve class

```
package ex05.pyrmont.valves;

import java.io.IOException;
import java.util.Enumeration;
import javax.servlet.ServletRequest;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class HeaderLoggerValve implements Valve, Contained {
  protected Container container;
  public void invoke(Request request, Response response,
ValveContext valveContext) throws IOException, ServletException {
    // Pass this request on to the next valve in our pipeline
    valveContext.invokeNext(request, response);
    System.out.println("Header Logger Valve");
    ServletRequest sreq = request.getRequest();
    if (sreq instanceof HttpServletRequest) {
      HttpServletRequest hreq = (HttpServletRequest) sreq;
      Enumeration headerNames = hreq.getHeaderNames();
      while (headerNames.hasMoreElements()) {
        String headerName = headerNames.nextElement().toString();
        String headerValue = hreq.getHeader(headerName);
        System.out.println(headerName + ":" + headerValue);
    }
    } else
        System.out.println("Not an HTTP Request");
    System.out.println ("-----------------------------------"); }
  public String getInfo() { return null; }
  public Container getContainer() { return container; }
  public void setContainer(Container container) {
    this.container = container;
  }
}
```

Again, pay special attention to the invoke method. The first thing the invoke method does is call the invokeNext method of the valve context to invoke the next valve in the pipeline, if any. It then prints the values of some headers.

ex05.pyrmont.startup.Bootstrap1

The Bootstrap1 class is used to start the application. It is given in Listing 5.12.

Listing 5.12: The Bootstrap1 class

```
package ex05.pyrmont.startup;

import ex05.pyrmont.core.SimpleLoader;
import ex05.pyrmont.core.SimpleWrapper;
import ex05.pyrmont.valves.ClientIPLoggerValve;
import ex05.pyrmont.valves.HeaderLoggerValve;
import org.apache.catalina.Loader;
import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap1 {
  public static void main(String[] args) {
    HttpConnector connector = new HttpConnector();
    Wrapper wrapper = new SimpleWrapper();
    wrapper.setServletClass("ModernServlet");
    Loader loader = new SimpleLoader();
    Valve valve1 = new HeaderLoggerValve(); Valve valve2 = new ClientIPLoggerValve();
wrapper.setLoader(loader);
((Pipeline) wrapper).addValve(valve1); ((Pipeline) wrapper).addValve(valve2);
connector.setContainer(wrapper);
try { connector.initialize(); connector.start();
// make the application wait until we press a key.
System.in.read(); }
catch (Exception e) { e.printStackTrace();
} }

}
```