Java Server Side Programming

The Conceptual Foundation

Mukesh Prasad

Java Server Side Programming

The Conceptual Foundation

Mukesh Prasad

Contents

Introduction

Server-side programming is frequently learned as a collection of disconnected information. Students and programmers learns various features provided by Java servlets and Java Server pages etc. one piece at a time. Very slowly the full picture is acquired, as puzzle pieces fitting into place. Until then, the learning method is very much like rote memorization. And even then, the full picture may be incorectly understood because it was learned so haphazardly.

This book aims to make the understanding of the main conceptual framework behind server technologies, the first step in learning server-side programming.

The learning philosophy is “learning by doing”. Instead of learning details of various pieces, the reader is guided into writing a Java web server. This is a surprisingly simple task in the modern Java environment, and the task of writing a well working static web server is completed in merely the first two chapters!

Working on the understanding built from this, the reader then implements a mini-version of Java servlets, and then progresses to implementing a mini version of the JSP technology.

The servlet and JSP subsets are chosen to minimize configuration and string-manipulation, and to focus on the core issues. The JSP subset invariably requires some complex string-manipulation. Therefore string-manipulation routines are provided in the accompanying software, so as not to distract from the core understanding being built up.

The Java server, including the servlet and JSP subsets, can be implemented in under 1000 lines of Java code! This includes cookies, sessions and session expiration, as well as automatic detection, recompilation, rebuilding, and redeployment of server pages upon modification.

Once the reader has a fundamental conceptual grasp of what is going on, the full servlet and JSP technologies are introduced, with reference to what the reader has already built.

Basic knowledge of Java is a pre-requisite, including knowledge of Threads and I/O. A small amount of HTML familiarity is also required.

The book starts from the level of Java sockets. Knowledge of Java sockets is not assumed or required.

The web-server built in the book is known as MVOWS, for My Very Own Web Server. The discussion frequently refers back to MVOWS.

The web-server is the core technology for server-side programming, and fully enables and constrains what can and cannot be done, and the various tradeoffs thereof. Therefore a conceptual fundamental understanding at this level will not only make the current-day server-side technologies simpler to acquire, but will also go a long way towards easy acquisition of any future server-side technologies that may emerge.

Using this book

This book is organized as follows.

First of all, there is a chapter introducing sockets. In this chapter, the reader builds a small server program and a small client program, capable of writing and reading lines of text to and from each other.

The next chapter involves writing a static web server in Java, called MVOWS (My Very Own Web Server.)

Then something similar to Java servlets, but a stripped down version, is added to that web server.

This is followed by adding something similar to Java Server pages (JSP), also a stripped down version, to that web server.

This prepares the student for full-fledged Java servlets and JSP, which are then introduced. Their explanations are based upon the stripped down versions already familiar to the reader.

The intent is that the reader will see the technologies from the bottom up, and will have full hands-on experience with them. The reader should actually write and compile all of the socket servers, MVOWS, and the stripped down versions of servlets and JSPs. The reader should also get all of these to work. This will provide a visceral understanding, and a mental framework for server side technologies.

The terminology used in the book for Java applications like Tomcat that provide servlet and JSP technologies, is "Java server". Sometimes they are called "containers", but they all can serve HTML and other files, therefore the "Java server" term has been preferred in the book.

Code Download

Most chapters in the book have code listings.

Accompanying the book is a ZIP file available for download.

Download information is provided on the last page of the book.

The ZIP file contains three top level folders, "listings", "solutions" and "support". The "listings" folder contains code for all the listings in the book as text files, so they can be easily copied and pasted. The "solutions" folder contains full solutions to the problems presented. Some of the files from the "solutions" folder are copied to the "support" folder. The difference between "support" vs. "solutions" folder is that you are encouraged to use items from the "support" folder, as they are meant to keep your focus on the Java server concepts, instead of getting lost in string manipulation etc.

How to use this book

It is strongly recommended that the reader should get all the examples to actually work.

In case you are not familiar with sockets, the first chapter will introduce you to them. If you are familiar with sockets already, use the solution provided in the first chapter, or write a minimum framework as per the listings in the first chapter. This is important, because the next chapter builds upon the work of the first chapter.

The contents of the book can be used at various different levels of learning, depending upon the time and effort the reader wishes to invest.

The code provided in the `support` folder should always be used to help - this code is meant to keep your focus on server-side concepts, instead of getting you bogged down in string manipulation etc. Of course, if you are excellent at string processing, feel free to write your own, but be aware that is not the focus of the teaching in this book.

If sufficient time and effort can be spared, the reader should not use copy-and-paste of the code listings at all. Instead, the reader should get MVOWS to work by reviewing the code listings, and then writing them on his/her own. (When you get stuck, get back to the book and review it again.) The code listings and full solutions are provided for studying and comparing your solutions to the provided solutions.

The next best approach is to copy/paste the code listings into classes written by the reader, and to make it all work.

Finally, some readers could just use the full solutions provided, and just review the code and see how it works. For learning purposes, it is best in this cases to make modifications to the provided solutions, to make it do alternative and enhanced tasks. (This is also a suggested approach for educational classes.)

The level of detail

The book provides depth, and eschews too much width.

It doesn't cover all the various nitty gritty details. The philosophy of the book is to teach the important concepts in depth, therefore not a lot of user attention is used up in details. The details can be easily found using a web search engine, at the time when they are really needed. At that time the reader will be really focused on those details, whereas if provided in the book, the reader will only vaguely grasp the details.

Instead of a lot of details, most relevant technologies are very lightly discussed so that the reader will know that those technologies exist, and will be ready to search for them when needed. There are no "cookbook recipes" in here. The belief is that the reader can easily search for them or invent them, as long as the in-depth understanding has been provided.

Important note about non-Windows systems

Because the focus in this book is on the concepts of a web-server rather than the detailed mechanisms and efficiencies, the highly convenient `println` method available in Java is frequently used.

On some non-Windows systems however, `println` does not do what is expected in a web-protocol environment. If you are working on a non-Windows system, make sure to add the following line at the start of your `main` method.

```
System.setProperty("line.separator", "\r\n");
```

This will take care of the `println` behavior problem.

Chapter 1
Programming with Sockets

In Java, “sockets” are how you connect two programs to each other using the internet.

The term used in Java, “socket”, is reminiscent of electrical sockets in everybody’s houses. We can use the analogy a little bit, though it won’t go very far.

The basic concept is similar, in that you use an electrical socket to connect an appliance to a power-service. Using Java sockets, you connect two programs together.

But there are many differences between the electrical sockets in your house, and the Java “socket”s. The first difference is that all sockets in your house are served by the same electric supplier (well, unless you are an alternative energy pioneer of some kind!) In the programming world, a supplier is known as a “server”, and there can be many servers available. Each server can make a large number of “sockets” available.

Into electrical sockets, you can plug in appliances, e.g. a toaster. In Java sockets terminology, a toaster would be a “client”. An electrical toaster receives electrical power from its socket. The client program can receive data from its socket. But unlike an electrical toaster, the “client” program can also send data to the server. The transfer is bi-directional.

Just like your electric supplier can serve a large number of appliances, in the programming world, a “server” can serve a large number of “client”s simultaneously. Most server programs are designed with this in mind. Unlike your household appliances, a client program can plug into more than one servers using a separate socket for each server (though this is not often necessary.)

A web browser like Internet Explorer or Firefox or Chrome or Safari, is an example of a client. Typically, your browser is plugged into a server, say at Google.com or at MSN.com. The servers at Google.com and MSN.com are written so as to be able to handle a large number of clients. If you are using tabbed pages, your browser is also plugging into a number of different servers.

The port number

Servers and clients communicate using a “port number”. A “port number” is basically just some number some programmer somewhere effectively pulled out of a hat! The number has to be higher than zero and lower than 65536 (so it can fit in 16 bits.) To communicate with a server, the client program has to know its “port number”, and also the machine where the server is running.

So let’s say, you picked the number 1234 as your port number, and you write a server that uses that port number. In network programming terminology, your server “listens to port 1234”. Then in order to connect to your server, any client program has to know that the port number that your server is listening to, is 1234.

This mechanism is not very organized, but it works. There are a number of standard port numbers, that somebody picked out a long time ago somewhere, and now they have become a part of the world. For example, the port number 80 is used for browsers and their servers. Port number 25 is used for email. Port number 21 is used for file transfer and so on. So if you want to connect to the File Transfer server for organization BigCo.com, all you have to do is plug into port number 21 at their File Transfer server’s hosting machine.

If you write your own server and your own client for your own special purposes, you will have to pick a port number too. Obviously, picking up port number 80 is not a good idea, unless you want to serve up web pages. Port number 25 and 21 aren’t too good either… In general, programmers are expected to use port numbers higher than 1024, so the basic services have that range to themselves.

Even if you use a port number higher than 1024, there is some chance that you may use a port number that some other programmer at your organization is already using for a different type of server. In practice, this is not a huge problem, because when you discover this, you just change your port number!

The advantage of standardized port numbers is that anybody can write their own servers for a port like port 80, that serves up web pages. Similarly, anybody can write their own client that plugs into port 80 and reads web pages from various servers.

The Connection Setup

As mentioned earlier, the electrical socket metaphor breaks down rather quickly as we get into the programming world. The physical electrical sockets are very simple, you plug something in, and you have a circuit. In the programming world, things are a bit more sophisticated.

First of all, the server sets up a “Server Socket”, and uses it to listen to the “port number”. Each time a client tries to connect to that “port number”, a new “connection” is created between the client and the server. This new “connection” is only a little like our electrical socket. It’s as if each time you tried to plug your toaster into a socket, a brand new socket was created just for that toaster and just for that one attempt to plug it in!

Both server and client get one end each of the socket. In programming terms, they get a “socket” object apiece (apart from the original “Server Socket”), and they can use these “socket”s to communicate with each other.

This might not be very clear. That’s all right, it should become very clear once we start clarifying the concepts using the more precise language we have in common, Java!

Writing a Socket Server and a Socket Client

The examples in this book assume you have a way to run two or more Java programs at the same time on your computer. Because there are many different Java IDEs available and the details vary, in this book we assume that you are comfortable with starting a Java program in the Command Prompt, and you know that a Control-C can be used to stop such a Java program, and you know that you can start up several Command Prompts and run a separate Java program in each one.

You don't have to use the Command Prompt method, if you are familiar with your IDE and how you can run two or more different programs (a server and one or more clients) at the same time using your IDE. (However, if your IDE is giving you trouble, it might be worth it to spend a few minutes learning the Command Prompt method if you don't already know it!)

In this book, we also assume that you are reasonably familiar with Java, and can write classes with main programs and handle exceptions with no trouble. Many code listings in the book only supply the inner details. To make them work, you will need to put them in classes, supply a main program, handle exceptions and so on.

Ok, so let us write our first socket server. There is a Java class to help you write servers. This class is `java.net.ServerSocket`. To write a server, all you have to do is instantiate an instance of `java.net.ServerSocket` specifying the port number you want, and call its `accept` method. The `accept` method will block (it will not return) until some client tries to plug into your server, using the port number you have chosen. Once a client has plugged in, the `accept` method will return a `java.net.Socket` object. The `Socket` object contains both ends of a data sending/receiving mechanism.

Reading from and writing to the socket is done using the java.io package, that you should already be familiar with. You can get an input stream as well as an output stream from the Socket object. You can use these to read messages sent by the client, and to write messages to the client.

Similarly, the client will have its own Socket object that is the other end of the server's Socket object. The client can also get an input stream as well as an output stream from its socket object. What the server writes as output, the client will see as its input. What the client writes as output, the server will see as its input.

Let us write a very little program and get it to work. Here is a basic code fragment, just put it in a main program, and get it to work.

Note that in all the programming in this book, it will be convenient to import `java.net.*` and `java.io.*` packages.

```
ServerSocket server =
      new ServerSocket( 1234 );

System.out.println(
     "Server created, waiting for client");
Socket socket = server.accept();
System.out.println("Client has connected");
```

Listing 1.1

Write this and make it work. If you see a "bind error", then something on your machine is using the port 1234, try another port number.

Once the server has successfully started, you should see the output "Server created, waiting for client", but nothing further. That's because your server is waiting for a client. So let us now give it a client. But first, note that we haven't given the server an exit mechanism, so if we want to stop it, we have to interrupt it using Control C. If you give that a try, and then try to restart the server immediately, sometimes you may also see a "bind error". This time, we know there is no other program on your machine that's using the same port! The problem here is that your new instance of the server is conflicting with the previous instance of the server! Even if you control-C'd out of the previous instance, occasionally the operating system may take a few seconds to tear down the circuitry. So if you see the "bind error" of this kind, just wait a few seconds and then start up the server.

Ok, with that out of the way, here is the basic code fragment for a client:

```
Socket client = new Socket("localhost", 1234 );
```

That’s it for now - put this in its own main program. Then run the server program in one Command Prompt (or Shell) window, and while the server program is waiting for a client, run your client program in another window. As soon as you run your client, the server program should print “Client has connected”. If you change the port numbers so the server/client port numbers do not match, you will not see this message, the server will keep waiting for a client on its port.

Both the server and client programs should exit after this. So you have two programs communicating between two windows on your machine. The string “localhost” just says that the client should look for a server on the same machine as itself. But the same mechanism works across machines. As long as a client knows what machine to talk to, it can do so by specifying the name or address of that machine instead of “localhost”. Instead of the name "localhost", you can also use the "IP Address", which is a numeric address of a machine. The "localhost" corresponds to the "127.0.0.1" IP address, and you can use that instead of "localhost".

Once the server has exited, if you try to run the client program again, you should see a “Connection refused” exception. This exception means that there is no server listening to the port you have specified, on the machine you have specified (in this case, port 1234 on “localhost.)

For the next step, let us send a message from the client, and have the server print it out.

On server, add

```
InputStream inputStream =
            socket.getInputStream();

InputStreamReader isReader =
     new InputStreamReader( inputStream );

BufferedReader inputReader =
  new BufferedReader( isReader );

System.out.println("Client wrote: " +
  inputReader.readLine());
```

Listing 1.2

Here, we are asking the `socket` for an input stream. This is how you communicate via the socket. It is very much like reading data from a file, except in this case the data is coming from the other end of the socket (the "client" program.)

We are turning the input stream into a `BufferedReader`, because that makes it easiler to read data one line at a time. Because there is no direct constructor of `BufferedReader` that accepts an input stream, we need to go via an intermediate `InputStreamReader`, but otherwise we don't need that `InputStreamReader` as such.

Once we have the `BufferedReader`, we then try to read a line of data and print it on the console.

But if you run this code fragment on the server side now, and if you run your client program after the server has started, you will see the server receives a “Connection reset” exception. That’s because the client hasn’t written anything, and the `inputReader.readLine()` is waiting until it sees a full line of text. But the client exits rather than writing a line of text. The operating system closes the underlying connection, therefore the server receives an exception notifying it that the connection was closed.

Let us modify the client to write a line now. On the client, after the

```
Socket client =
     new Socket("localhost", 1234 );
```

add

```
OutputStream outputStream =
   client.getOutputStream();

OutputStreamWriter oswriter =
   new OutputStreamWriter( outputStream );

PrintWriter outputWriter =
   new PrintWriter( oswriter );

outputWriter.println("Hello from client");

outputWriter.flush();
```

Listing 1.3

Here we are doing the opposite of what we did on the server. We are asking the `socket` for an ouptut stream. We are going to be writing to this output stream. But again, we would prefer to use a `PrintWriter` for convenience of writing a line at a time. So we build a `PrintWriter` via an `OutputStreamWriter` from the output stream.

Once we have the `PrintWriter`, we write a line of text to it. Again, this is just like writing to a file, except the data is being sent over to the other program (the server) in this case.

Now run the server in one window, and the client in another window, and you should see the “Hello from client” being output by the server. The `outputWriter.flush()` is important, because when we use `java.io` package, the Java I/O routines often buffer the output for improved performance. Unless the buffers are flushed, the output will not always get sent out. The output will get sent out once the buffers are full, or once the stream is closed. But our client is just falling off the end without closing the stream and all that good stuff… so if we don’t flush the output, it won’t get flushed and the output will be sitting in the Java internal buffers and the server will never see it! This is an important thing to become aware of early in network programming. It is not unusual for a bug to be caused simply due to an output buffer being left unflushed.

Ok, to finish up this basic introduction, let us send some bytes in the other direction as well. Unlike files, with a socket, you can do both reading and writing. What one side writes, the other side reads. What the other side writes, the first side reads!

At the end of the client code, add

```
InputStream inputStream =
   client.getInputStream();
  BufferedReader inputReader =
    new BufferedReader(
      new InputStreamReader( inputStream ));
  System.out.println("Server wrote: " +
      inputReader.readLine());
```

Listing 1.4

and at the end of the server code, add

```
OutputStream outputStream =
     socket.getOutputStream();
PrintWriter outputWriter =
  new PrintWriter(
    new OutputStreamWriter( outputStream ));
outputWriter.println("Hello from server");
outputWriter.flush();
```

Listing 1.5

You will notice that this is just a mirror reversal of what we did earlier to send bytes from the client to the server (though the code is a little bit compact, because we don't really need to retain the intermediate `InputStreamReader` and `OutputStreamWriter`, and the flow of object creation should be clear by now.)

Once the connection is established, both sides act identically.

When you run this latest version of the server and then the client, you should see that both sides have received the text sent by the other side.

Solutions folder: 1.1

Handling multiple clients

We have seen the very basics of communication between clients and servers. But as was mentioned earlier, a server usually has to be able to handle multiple clients. Nor can it exit every time a client finishes talking to it. Imagine the web server shutting down happy about a job well done, every time it successfully serves up a web page!

A very common way to handle all this in Java is to use Java Threads. As soon as the server accepts a new connection, it fires off a new thread that is dedicated to that connection. That thread does all the communication necessary with the client, and when done, that thread exits, not the main server program. In the meanwhile, the main server program is free to accept more connections, and to fire off more threads for each communication. (For performance and other reasons, all this could get very complicated, e.g. threads could be reused, or some processing could be handled “asynchronously”, or threads may poll for connections etc… We don’t need to get involved with all of that, since our purpose is learning the concepts here.)

So let us modify the server class, and instead of directly processing the socket it has received from accept, let us have it fire up a separate thread.

```
ServerSocket server =
        new ServerSocket( 1234 );

for (;;)
{
  Socket socket = server.accept();
  new ServerInstance( socket ).start();
}
```

Listing 1.6

Once the server establishes its main “socket server”, every time it accepts a new connection, it fires off a new instance of `ServerInstance`, and goes back to accepting the next connection.

But what is `ServerInstance`? Well, we have deferred the writing of it - it is a class we will be using to run a separate thread.

As you may have guessed if you are familiar with thread programming (and if you are not, you should read up on and understand Java multi-threading before proceeding), the `ServerInstance` class extends the `Thread` class. That way, the server’s main loop can accept multiple clients without having to wait for any client to finish its business.

Every instance of the `ServerInstance` class is responsible for dealing with only one connection with only one client. It reads/writes data from/to the client, and takes any necessary actions. It also makes sure any exceptions will not affect the server’s main loop.

Before we can compile our new server, we have to provide the ServerInstance class. The ServerInstance class must extend the `Thread` class, and must have a constructor that accepts a `Socket` object as parameter, and then saves that in an instance variable:

```
Socket socket;

public ServerInstance(Socket s)
{
  socket = s;
}
```

The ServerInstance class must also define a run method, which does all the work for that server instance.

```
public void run()
{
  try {
    InputStream inputStream =
      socket.getInputStream();
    BufferedReader inputReader =
      new BufferedReader(
        new InputStreamReader( inputStream ));
    System.out.println("Client wrote: " +
      inputReader.readLine());

    OutputStream outputStream =
      socket.getOutputStream();
    PrintWriter outputWriter =
      new PrintWriter(
        new OutputStreamWriter(
                     outputStream ));
    outputWriter.println("Hello from server");
    outputWriter.flush();
  } catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Listing 1.7

Note that the task of “run” is identical to what the server was doing earlier, but now all of that runs in a separate thread.

Once you make all this work, run the new server in a Command Prompt window. From a separate Command Prompt run the client. The server and client should print out the data sent by each other.

The server no longer exits after each run. You can re-run the client any number of times, and the server will respond.

Let us now make the server be able to handle multiple data messages. It will respond to each message by echoing the message back to the client, unless the message is “exit” in which case it will terminate the connection. To do this, we need to change the run method in the ServerInstance class, to contain a loop.

```
public void run()
{
  try {
    InputStream inputStream =
           socket.getInputStream();
    BufferedReader inputReader =
      new BufferedReader(
        new InputStreamReader( inputStream ));
    OutputStream outputStream =
      socket.getOutputStream();
    PrintWriter outputWriter =
      new PrintWriter(
        new OutputStreamWriter(
              outputStream ));

    while (! terminate)
    {
      process( inputReader, outputWriter );
    }

    socket.close(); // Graceful termination

  } catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Listing 1.8

Here, we have added a `while` loop which simply keeps calling the method `process` until the class variable `terminate` becomes true.

Obviously, a class variable

```
boolean terminate = false;
```

needs to be added in the class to make the above code work.

Also, we have deferred the processing to a method named `process`, so that method will need to be written. Let us just make it echo the line back, so the processing will be simple enough:

```
void process( BufferedReader input,
              PrintWriter output )
           throws IOException
{
  String line = input.readLine();
  // Echo the line back
  output.println("Echo: " + line);
  output.flush();
  if ( line.equalsIgnoreCase( "exit" ))
  {
    terminate = true;
    System.out.println(
           "Connection terminated" );
  }
}
```

Listing 1.9

The `process` method simply reads a line, and echos it back. If the line happens to be the string `exit`, it sets `terminate` to true.

Get this to work, and you will be quite close to writing your very own Java web server!

When you test this, the client we have been using so far, will work. But the connection termination still happens via an exception, rather un-gracefully! The client is not aware that it should send an “exit” to terminate the connection.

We don’t need to make the client too fancy, simply have it read lines from `System.in`, and send those over, until it reads an “exit” at which point it sends the “exit” and closes its side of the socket.

```
Socket client =
        new Socket("localhost", 1234 );

OutputStream outputStream =
  client.getOutputStream();
PrintWriter outputWriter =
  new PrintWriter(
   new OutputStreamWriter( outputStream ));

InputStream inputStream =
   client.getInputStream();
BufferedReader inputReader =
   new BufferedReader(
     new InputStreamReader( inputStream ));

BufferedReader cmdReader =
   new BufferedReader(
       new InputStreamReader( System.in ));
for (;;)
{
  System.out.print( "Command> " );
  String line = cmdReader.readLine();
  outputWriter.println( line );
  outputWriter.flush();
  if ( line.equalsIgnoreCase( "exit" ))
  {
    break;
  }

  line = inputReader.readLine();
  System.out.println("Server wrote: " + line);
}

client.close();
```

Listing 1.10

The client is now reading lines from `System.in` via a `BufferedReader`, and is writing those lines out to the socket via `PrintWriter`.

After every write, the client reads a line from the socket via another `BufferedReader`, and prints that line out out `System.out` before looping back for the next line from the user.

Using this version of the client, you can now connect multiple clients to your server (try running multiple connections by opening several clients in several Command Prompt windows), and can gracefully close the connections.

Solutions folder: 1.2

What is a Protocol?

In the server/client we have written above.

- Every time the client sends a line of text, the server responds with a line of text.
- The client can send any text, as long as it’s not “exit”.
- If the client sends “exit”, the server assumes client is going to close the connection, and the server closes its own side of the connection.

Those 3 rules above, define what is known as a “protocol” in network programming.

A “protocol” is a word often used in diplomacy, and it defines expected and agreed-upon rules of behavior.

In computers, the “protocol” is also agreed-upon rules of behavior. If you are writing both a server and a client, you get to make up the protocol. (Like the 3 rules we made up above.)

If you are writing a server and somebody else is writing a client, both of you must make up the protocol together, or have a third party make up the protocol. The important part is — both the client and server must agree upon the same protocol, and the protocol must have enough detail so that both the programmers writing the client and the server should be able to do their job, and the protocol must have enough power in it to be able to get the task done.

With all of the above, now you know enough to start writing MVOWS - My Very Own Web Server.

Chapter 2
Your Very Own Web Server

You know enough at this point to start writing your very own web server.

Getting Started

Let us start with a few small changes to the server program you have already written.

Let us add a package name `mvows` (for "My Very Own Web Server") to both the server .java files to help organize the code, as this code is going to get a little bit more complex.

Then, change the port number to 80. Port number 80 is the standard port number for web servers. After changing the port number to 80, run your server again. If your server fails to start now, that may mean you have a web server running on your machine. You could try stopping that other server. Otherwise, you could run with the port number you already have (e.g. 1234.) But in that case, wherever the book refers to using

```
   http://localhost/
```

you must change that to

```
   http://localhost:<your-port-number>/
```

e.g.

```
   http://localhost:1234/
```

instead.

The second change is to `ServerInstance` — remove all the “echo” and “exit” logic, and simply read lines and write them out to `System.out.println`. Don’t worry about a graceful exit, we will go back to our ungraceful Control-C to shut down the server. At this point, we also don’t need to bother with sending any output. So the main loop in the `run()` method is very simplified now. Replace the

```
 while (! terminate)
 {
   process( inputReader, outputWriter );
 }
```

loop to this one below, which simply prints out the lines. Right now, we just want to see what we are dealing with.

```
   for (;;)
   {
       String line = inputReader.readLine();
       System.out.println( line );
   }
```

You will also have to take out the unreachable code after this loop. You can also remove the `process` method and the `terminate` field.

If you are running on a non-Windows machine (e.g. Linux or Mac), make sure to add in your main program

```
   System.setProperty("line.separator", "\r\n");
```

before any of the processing starts. (On Windows, this step is not necessary because this is the default.)

That’s it.

Fire up your server, and after you see the “Waiting for client connection” message, start a browser (e.g. Internet Explorer, FireFox, Chrome etc), and point it to the URL

```
http://localhost/
```

The “localhost”, once again, means “this machine I am running on”. Since your server is on the same machine as your browser, you can use “localhost” instead of needing a name or address for your machine.

Solutions folder: 2.1

The Protocol

If you have done everything correctly, you should see your server print out a bunch of lines. Your browser meanwhile, is waiting for some kind of response. Control-C out of your server, at which point your browser will report that it failed to connect to localhost.

The lines that your server printed out should look something like this:

```
GET / HTTP/1.1
Host: localhost
Connection: keep-alive
<<and a lot more gobbledygook…>>
```

What you are looking at, is the **protocol** that has been defined for web servers and clients. It is known as HTTP (HyperText Transfer Protocol), and is the basic protocol used by all web servers and browsers.

The first line is the most important

```
GET / HTTP/1.1
```

This line contains something known as the “method”, which is usually GET or POST. In this case, it is GET. After the GET, the next thing in the line is the / character, which tells the server the file (or more accurately the “resource”) that the browser wants. Run the server again, and make your browser try to retrieve

```
   http://localhost/SomeResource.html
```

this time. You should see the first line change to:

```
GET /SomeResource.html HTTP/1.1
```

The last part of the first line is HTTP/1.1, which specifies the type and version of the protocol, i.e. HTTP version 1.1. Currently, HTTP/1.1 is the most commonly used protocol by web clients.

The Empty Line

The first line - known as the request line - is followed by a number of other lines, known collectively as the request headers.

Following the request headers is an empty line.

The empty line is very important in the HTTP protocol. It marks the end of the request headers, which usually is also the end of the full request. The empty line is how the server knows when to stop reading from the client, and to start sending back stuff to the client.

The server's response similarly uses an empty line to differentiate between the response headers, and the response data.

The Server Responds Back

Now let us modify the server to send some text back. Here we need a bit of information about the protocol in the other direction. As we saw, the protocol for GET requests is that the browser (the client) will send several lines of text, followed by an empty line. The empty line is the signal to the server that the browser is done sending lines. At this point, the server is expected to reply.

The server's reply has to start with one or more protocol lines, followed by an empty line, followed by the contents of the file being asked for (or any other "resource".) The minimum we can get away with, is a line as follows

```
HTTP/1.0 200 OK
```

followed by an empty line.

In addition, we should tell what is the "type" of the content in the headers. Though even without it, the browsers are good at making guesses, it is easy enough to add a line like

```
Content-Type: text/html
```

before the empty line. This tells the browser that we are sending a text file, that contains HTML marked up text.

So we need to modify the server as follows — read lines until an empty line comes in. Then exit the read loop, and send the response line and a content-type line as shown above, then send an empty line, then send the HTML. For now, instead of figuring out what resource or file to send, we can just send some friendly text, e.g.

```
InputStream inputStream =
        socket.getInputStream();
BufferedReader inputReader =
  new BufferedReader(
     new InputStreamReader( inputStream ));
OutputStream outputStream =
  socket.getOutputStream();
PrintWriter outputWriter =
  new PrintWriter(
      new OutputStreamWriter( outputStream ));
for (;;)
{
  String line = inputReader.readLine();
  System.out.println( line );
  if ( line.equals( "" ))
      break;
}
// Now send the response
outputWriter.println("HTTP/1.0 200 OK");
outputWriter.println("Content-Type: text/html");
outputWriter.println(); // The empty line
outputWriter.println("<HTML>");
outputWriter.println("<BODY>");
outputWriter.println("Hello from ");
outputWriter.println("My Very Own ");
outputWriter.println("Web Server");
outputWriter.println("</BODY>");
outputWriter.println("</HTML>");
outputWriter.close();
inputStream.close();
socket.close();
```

Listing 2.1

The above code is sending the HTTP and the Content-Type lines, then an empty line to mark the end of the headers, and then it is sending some actual HTML text.

Note that we don’t have to flush the output this time, because we are able to actually finish the protocol, and can gracefully shut down the connection, including closing `outputWriter` and `inputStream`. Closing `outputWriter` automatically flushes any pending output.

When you run a browser against this modified version of the server, you should see the “Hello from My Very Own Web Server” in the browser. You may want to experiment with changing the text you send, and you should be able to see your changes in the browser.

Though browsers can sometimes "cache" the data, i.e. they may not bother to run the request again and just show the data from the previous request. If you don't see the output change in response to changes you made to what the server is sending, that's what is happening. If this happens, just request another resource, e.g.

```
http://localhost/test2
```

Now the browser will realize you are asking for something different, and will no longer use the previous data from the cache.

Solutions folder: 2.2

The favicon.ico request

Once you start sending data back, you may also start seeing requests for something called "/favicon.ico". This is just the browser asking for a little icon it can show for the HTML page. You can just ignore those requests, and the server will just send the same HTML for those requests as well. For our purposes it is better to disable these requests by adding something to the HTML we are sending back as follows

```
<head>
<link rel="shortcut icon" href="about:blank" />
</head>
```

because we will be studying the input that comes in from the browser, and it's best to minimize clutter. Some browsers (specially Mac related) may send some other requests, and some of it may just have to be ignored.

(If you are not very familiar with HTML, the above HTML text goes between the <HTML> and the <BODY> tags.)

Solutions folder: 2.3

What If Everything Is Not OK?

The first line you are sending is

```
HTTP/1.0 200 OK
```

This says the server’s protocol is HTTP version 1.0, and the “response code” is “200”. The response code of 200 means everything is OK. The following “OK” is a more human-readable version of the “200”, and is known as a “Reason Phrase”.

What if everything is not ok, and the server wants to send an error message instead?

There are a number of response codes defined for various situations. For example, “401 Not Authorized” means the client needs to authorize itself, usually by having the user log-in and provide a username/password. The response “404 Not Found” means the file/resource that the browser was asking for, was not found on the server.

You can try changing the server to send a

```
HTTP/1.0 404 Not Found
```

response instead, and remove all the text after the empty line the server sends back, and see how your browser responds to it. Note that some browsers will show you the response they have “cached” from an earlier transaction, and will not display an error message. Again, to get around this, just change the resource you are requesting, e.g. switch to the URL

```
http://localhost/another.resource
```

More About The Client Request

What about all the lines we dismissed earlier as

```
<<and a lot more gobbledygook…>> ?
```

These are called the “header” lines of the “request”. For the GET requests we have been seeing, the header lines followed by an empty line makes up the total request.

The header line can contain a wide variety of information. One of these is a “User Agent” line, which describes the browser (Internet Explorer vs. Chrome vs. Firefox etc.)

Another header line describes the language of the user (e.g. “en-US”, which stands for “English, the United States variety”.) Some web sites can use this information to send content in the user’s language. Related to the language, are header lines for “Encoding” and “Charset” — these concern the representations for various languages around the world.

The header lines can also contain login (authorization) information, and various other useful items.

Cookies

One of the more interesting bits of information that may be contained in the header lines sent in the client’s request, is “cookies”.

A “cookie” is just some random string of character the server makes up, and gives to the browser. The browser saves that string, and whenever in the future that particular browser visits that particular server, it will tell the server “here is the cookie you gave me”.

What could be the advantage of such an arrangement? Well, suppose the server is handling requests from 10 clients at a given time. It could give the string “001” to the first client “002” to the second client, and so on.

Now, every time the server sees the string “005”, it knows that the request is coming from the fifth client. This allows the server to maintain context. For example, it may keep track of the fact that the fifth client has a user who has reported his name as “John Smith”, and it may insert “Hello, John Smith” at the top of each page it sends back. This type of context information is very useful in processing web forms.

The cookies are of two types, temporary and permanent. A temporary cookie is lost when the user closes the browser. A permanent cookie is saved by the browser, and if the user returns to the same web site, say, a few days later, the browser will still report the same cookie. Permanent cookies have expiration dates.

All cookies have a name, so you can give the same browser more than one cookies. E.g. for a cookie named "browserNumber", you may give the value "005", but for a cookie named "username", you may give out the value "Mr. Smith".

Let us try this out. In the server, while we are reading the lines of the client request and printing them out on `System.out`, let us also check if any of the line starts with the string “ `Cookie:`”, and if not, send a cookie out. You send the cookie by sending a line in the response, after the “ `HTTP 200 OK`” but before the empty line. You send the cookie name and the value. Let’s say you decide upon the name “MyVeryOwnCookie1”.

You also have to pick a value for the cookie, so you can give out different values to different browsers. The line you have to send is of the form:

```
Set-Cookie: name=value
```

e.g.

```
Set-Cookie: MyVeryOwnCookie1: cookie23
```

Here is the modified version of the code to send out cookies coming in from the browser.

Normally, cookies are random unique strings, but for simplicity here we are just using a static integer that we keep on incrementing. We define this static integer in the `ServerInstance` class.

```
 static int cookieNumber = 1;
```

Then we modify the run method

```
 // Get the input/output as usual
 InputStream inputStream =
         socket.getInputStream();
 BufferedReader inputReader =
   new BufferedReader(
      new InputStreamReader( inputStream ));
 OutputStream outputStream =
   socket.getOutputStream();
```

```
PrintWriter outputWriter =
  new PrintWriter(
      new OutputStreamWriter( outputStream ));

boolean haveCookie = false;
// Did browser send a cookie?

for (;;)
{
  // Read and print lines until empty line
  String line = inputReader.readLine();
  if ( line.equals( "" ))
      break;
  if ( line.startsWith( "Cookie: " ))
  {
    System.out.println("** Cookie Line **");
    haveCookie = true;
    System.out.println( line );
  }
  // System.out.println( line );
  // Avoid printing all lines so it is
  // easier to see only the cookie line
}
// Now send the response
outputWriter.println("HTTP/1.0 200 OK");
outputWriter.println("Content-Type: text/html");
if ( ! haveCookie )
{
  String cookieValue = "cookie_" +
               cookieNumber++;
  String cookieName = "MyVeryOwnCookie1";
  String cline = "Set-Cookie: " +
       cookieName + ":" + cookieValue;
  outputWriter.println( cline );
  System.out.println("Sending cookie: "
        + cline );
}
outputWriter.println(); // The empty line

// Our usual HTML
outputWriter.println("<HTML>");
outputWriter.println("<head>");
outputWriter.println(
  "<link rel=\"shortcut icon\"" +
  " href=\"about:blank\" />");
outputWriter.println("</head>");
outputWriter.println("<BODY>");
outputWriter.println("Hello from ");
outputWriter.println("My Very Own Web Server");
outputWriter.println("</BODY>");
outputWriter.println("</HTML>");
outputWriter.close();
inputStream.close();
socket.close();
```

Listing 2.2

Try it out. The first time, there will be no cookie coming from the browser. But once you have set the cookie, the browser will be sending you back the same cookie every time you refresh the page or return to it, and you should see the output on the console verifying it. Open a different browser (e.g. if you have been using Internet Explorer, use Opera or Chrome), and you should see the new instance getting and returning a different cookie.

Setting permanent cookies is very similar, but you also have to send out an expiry-date. Permanent cookies are saved until the expiry-date.

Solutions folder: 2.4

Serving Files

So far, no matter what the request, we always send “Hello from My Very Own Web Server”. We need to get a little more sophisticated about this now!

Let’s define a place on the hard drive where our HTML (and related) files will be kept. Let’s say, C:\MyOwnServerFiles. Put some HTML files in there, e.g. a file called hello.html which has the “My Very Own Web Server” HTML, plus some other files. You can put some text files and some graphic (.gif, .jpg, .png) files, and some audio (.wav) files in there. **Make sure to use simple filenames, with no spaces or unusual characters in there.** Just letters and/or digits, a period, and a file extension. Also put in an `index.html` file.

How do we send the appropriate file back?

When the GET request comes in, we need to break apart the request line (the first line of the request) on spaces, i.e. tokenize it. We can do this using the `string.Split` method. The first token is the GET keyword, which we will ignore. The second token is the name of the resource (known as the URI, or the Uniform Resource Identifier.) The third token is the HTTP/1.1. We will also ignore that. We are interested only in the second token.

```
// Get the request line separately from
// the rest of the headers

String requestLine = inputReader.readLine();

// Now read until the empty line,
// discard the header lines

for (;;)
{
  String line = inputReader.readLine();
  if ( line.equals( "" ))
      break;
  // System.out.println(line);
}
```

Listing 2.3

At this point in the code, `requestLine` holds a string of the form

```
GET /Resource HTTP/1.0
```

We can just split it on spaces. We don't need to worry about the "Resource" containing spaces. Even though actual resource names can contain spaces, they get encoded, so the request line only contains the three tokens.

The second token (the URI) is of the form “/” or “/SomeFile.html” etc.

```
String[] tokens = requestLine.split(' ');
String resource = tokens[1];
```

We need to locate this file, read its contents, and send the contents back to the browser. Since we keep the files in C:\MyOwnServerFiles, we need to look in there for the filename supplied.

As it happens, Java makes this very easy using java.io package. If you just use

```
File dir = new File("C:\\MyOwnServerFiles" );
File file  = new File( dir, resource );
```

Java will take care of all the / and \ characters and give you back a File object pointing at the correct location. This includes folders etc, so if the browser requests /abc/def.txt, the above code fragment will correctly point to C:\MyOwnServerFiles\abc\def.txt file.

If the second token is / or /abc where abc is a folder, the File object that you get will refer to a directory.

In that case, a standard convention is simply to append “index.html”. I.e.

```
 if ( file.exists() && file.isDirectory())
   file = new File( file, "index.html");
```

Once you have the final `File` object (referenced by “file” in the above fragments) you just need to read the file and send its contents over, instead of the HTML code we had hard-coded so far. Just read all the file contents using any mechanism you are comfortable with, and send it all out to the client before closing the output stream and the connection.

Note that `outputWriter` is a `PrintWriter`, so you can’t write byte arrays to it (though if you read byte arrays, you can turn them into a String and then write to the outputWriter.) You can write single characters to it, though.

This is also a good time to take out the cookie code we put in earlier. You may want to leave the `System.out.println` that prints out header lines, commented for now, except that we want to print out the very first line, which is treated differently.

There are two more things to take care of:

If the file doesn’t exist, send

```
HTTP/1.0 404 File Not Found
```

instead of

```
HTTP/1.0 200 OK
```

and do not send anything after the empty line in this case.

Here is some code to do all of that.

```
 if ( ! file.exists())
 {
   System.out.println("File " +
       file.getAbsolutePath() + " does not exist");
   // Send error
   outputWriter.println("HTTP/1.0 404 Not Found");
   outputWriter.println(); // The empty line
 } else
 {
   System.out.println("Sending file "
        + file.getAbsolutePath());

   // Send the success header

   outputWriter.println("HTTP/1.0 200 OK");
   outputWriter.println(
           "Content-Type: text/html");
   outputWriter.println(); // The empty line


   // Send out the file
   FileInputStream in =
             new FileInputStream( file );
   int c;
   while (( c = in.read()) >= 0 )
     outputWriter.write( c );
   in.close();
 }
 // Close input, output, socket as usual
```

Listing 2.4

We are sending out the bytes of the file if it exists, otherwise we are sending a "404" error code.

Solutions folder: 2.5

Finally, we should not be sending out `text/html` all the time, specially if we want to be able to handle JPEG and other files such as audio and video.

We need to check the file extension, and send a content-type matching the extension. Below is a list of some common extensions and the content-type for those extensions. You should do a case-insensitive test, e.g. by lowercasing the name and using `endsWith()` to compare against lowercased extension.

Content types for some commonly used extensions:

Extension	Content Type
.htm .html	text/html
.txt .text	text/plain
.css	text/css
.xml	text/xml
.gif	image/gif
.png	image/png
.jpg .jpeg	image/jpeg
.wav	audio/wav
.mpg .mpeg	video/mpeg

Sending non-text files requires some extra precautions, so we have to change our code a little to use the `outputStream` directly.

```
// Send out the file
outputWriter.flush();
FileInputStream in = new FileInputStream( file );
int c;

while (( c = in.read()) >= 0 )
   outputStream.write( c );
in.close();
```

Listing 2.5

Note that in sending out the file, we are not using the `outputWriter` any more, but are flushing it out and using the underlying `outputStream`.

The reason is that the `outputWriter` does special processing of end of line characters, known as CR (Carriage Return) and LF (Line Feed). CR and LF have values of 13 and 10, and the special processing makes sure that a single LF (10) gets turned into the CR-LF pair (13 followed by 10), which makes it better behaved on Windows and also the internet protocols.

But when sending non-text file contents (e.g. an image or video or audio files), we don't want any special processing of these characters. E.g. a video file may have a 10 in it which just refers to a pixel or a sound and doesn't mean an end-of-line (because videos, images, audios etc don't even have end of lines.)

The `PrintWriter` will interpret the 10 as LF, and it will helpfully send out an additional 13 as CR. This, unfortunately, will mess up the non-text data-stream, because the CR will be interpreted as an additional, incorrect, pixel or sound! So we flush out the `outputWriter` to make sure the header is sent out before the file. Then we read and send the bytes of the file using the `outputStream` directly.

This is an important issue in working with "PrintWriter"s vs raw "outputStream"s. We will come across it again while learning about Java servlets. (Note that on some operating systems this is not an issue, still it is good practice to be aware of the issue and to write code that avoids trouble.)

Finally, what if the file name has an unusual character such as a space?

Give it a try. You will see that the browser turns a space into a + character or %20, so that if you type in the browser http://localhost/My File.html

the server actually sees

GET /My%20File.html HTTP/1.1

Naturally, the server can't find a file by this name, even if you do have a file named "My File.html". You have to turn the %20 back into a space before the server can find the file.

This is a standard encoding mechanism, known as URL encoding. This is just basic string manipulation, so we don't need to spend much time on it here. As it happens Java provides a nice utility for this. Given a String "str", just call

```
URLDecoder.decode( str, "UTF-8" )
```

To get the unencoded version. Making this call on `"/My%20File.html"` will return `"/My File.html"`, which can then be used to locate the file.

You don't even need to check if the resource name contains any special character. Just always call

```
 resource = URLDecoder.decode( resource, "UTF-8" );
```

on all incoming requests.

Once you have implemented all this, your very own web server will be able to serve up HTML files with links, images, style sheets, audios, videos, the whole nine yards. At best, you may need to add additional content-types for any new kind of files, but your web server is fully capable of serving out any and all static websites. Not bad for less than 150 lines of code!

Solutions folder: 2.6

Your own web server is quite a regular web server now, the only big problem being the word static.

To have a web server be truly useful, programmers need to be able to get at the content delivery process. You need to give other Java programmers some hooks so they can change the way forms are processed, and modify the way a page is displayed. Even a simple page displaying today's date, requires programmatic access to the page contents.

We will proceed, in the next few chapters, to add Java programmability to MVOWS, so that some programmer who doesn't know anything about MVOWS will be able to add dynamic web pages to it.

This will be the same kind of programmability that standard full-fledged Java servers have, except that since our purpose is learning the technology, we will focus on the core technology alone, and not any of the numerous bells and whistles or efficiency issues.

There is one more thing to understand first — how FORM tags are handled in HTTP, our web protocol. In dynamic website programming, form processing tends to be quite central.

Form Processing, Query Strings and POST

Since MVOWS now has the ability to handle any kind of HTML file, let us build a very small HTML file with a FORM tag and a couple of INPUT fields. Let us call it MyForm.html.

```
<HTML>
<BODY>
<FORM METHOD=GET ACTION=MyForm.html>
Field 1
<INPUT NAME="field1" TYPE=TEXT SIZE=10>
<P>
Field 2
<INPUT NAME="field2" TYPE=TEXT SIZE=10>
<P>
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Listing 2.6

That’s it. Load this file via your web server, put something in the two text fields, and submit the file. Watch the output of the server (make sure your server is still printing out the protocol lines, at least the very first line!)

You should see that the resource being requested now is something like

```
/MyForm.html?field1=data1&field2=data2
```

Your server should be trying to find a file called `MyForm.html?field1=data1&field2=data2`, and since there is no such file, it should be returning a 404 error code.

Which brings us into query strings.

The resource being requested can include the ? character, followed by some text known as a “Query String”. The ? character and the query string are not part of the file name.

Therefore the web-server needs to see if the resource name includes a ? character, and if yes, it needs to use the characters before the ? in the filename, and save everything after the ? as the “Query String”.

Note the form of the query string in this case.

```
field1=data1&field2=data2
```

Here `field1` and `field2` are the name of the INPUT fields. The data for the fields is whatever the user typed into those INPUT fields.

To split the query string, simply split the resource at the ? character.

```
   String queryString = null;
   int qIndex = resource.indexOf( '?' );
   if ( qIndex > 0 )
   {
       queryString =
               resource.substring( qIndex+1 );
       System.out.println( "Query String=" +
             queryString );
       resource = resource.substring( 0, qIndex );
   }
```

Listing 2.7

This must be done before the URLDecode, and you must do `URLDecode` on the resource name alone. The parts of the query string also need to be URL-decoded, but separately.

Now when you press the Submit button on the form, the results should be correctly processed - you should see the form again (as that was the target of the Form), and the data should be correctly printed out on `System.out` as the value of `queryString`.

Solutions folder: 2.7

How does one process the information that the user put into the fields `field1` and `field2`?

That in fact covers a large part of server-side programming, and we will begin to deal with this in the next chapter.

For now, change the METHOD to POST in the FORM and give it a try, see what happens.

You will notice that the query-string no longer follows the resource name.

Actually, it follows the empty-line that terminates the headers. But since our server is only reading lines, it's not being able to see the query string from the POST. To be able to see this query string, we have to read everything after the empty line from the browser. To decide how much to read after the empty line, the header field `Content-Length` is used.

A typical POST request may look like

```
GET /MyForm.html HTTP/1.0
Host: localhost
Connection: keep-alive

field1=data1&field2=data2
```

(Some headers have been omitted above for clarity.)

Note that the empty line is followed by the query string. This is how the posted data is sent, instead of as a part of the request line.

We will not be handling POST methods in MVOWS, as it is very similar to GET for our purposes. The POST method is useful for handling large amounts of data, for uploading files, or if you don't want the user to see the query string in the browser's URL window.

Some Miscellaneous Information

Here is some more interesting information about web-servers. It is useful to know, though we don't need to implement this for our task of gaining a conceptual insight into server-side programming.

The header line

```
Host: localhost
```

tells the server the actual host-name that the client is trying to reach. That may seem redundant. Doesn't the server know its own name?

Well, this information is useful for implementing something called "virtual servers". With the modern-day very high powered hardware, it is not necessary to dedicate a full machine for low-traffic websites. Therefore the browsers tell the server the name that the client is trying to access, so a single server can provide services for multiple websites, each with its own unique name.

The header line

```
Connection: keep-alive
```

represents a very major change in HTTP.

In the earlier version of HTTP - HTTP/1.0 - all data transfer was done the way we are doing it in MVOWS. After sending a page worth of information, the socket was closed. This is simple to implement, yet this was causing a lot of overhead in the web, as setting up and tearing down the socket connections are relatively expensive operations. Usually people get a lot of data from the same server (even if end-users request only a single page, beneath the hood there are a lot of separate requests, for image files, style sheet files etc.) Therefore there were many connections being set up and torn down, and the user experience was much slower than it had to be.

Slowly this method gave way to "keep-alive" connections, where the browser maintains an open connection to the server. Instead of closing the socket after serving a file, the server reads the next request on the same socket, as long as the browser keeps sending requests on that socket.

In the keep-alive connections, servers need to know when one request ends and when the next one starts. For GET requests, it is easy, just read until an empty line comes in. But for POST requests, the server needs to know how much to read after the empty line. This information is retrieved from the header field "Content-Length".

Once the server has completely read the GET or POST request, it can read the next GET or POST request on the same socket connection. We will not be doing this in MVOWS as we are not terribly concerned with efficiency, but it is useful to be aware that this is what is usually going on in typical servers.

This keep-alive technique is a standard part of HTTP/1.1.

There is also a header `If-Modified-Since` which lets the browser ask "Has this resource been modified since this date-time?"

If the resource has not been modified, the server sends back a header saying it's not been modified. The browser then uses its old copy. If the resource has in fact been modified, the server sends back the resource as usual. This allows the browser to avoid some file transfers, thus further improving the response times and further reducing the network utilization.

Your Very Own Little Browser

Before we wrap up this chapter - at this time, incidentally, we also know enough to write a simple web client that acts like a browser. In fact, being able to write your client is a useful exercise, simply so you can see what the big folks do when faced with various protocol issues. In this book we are not going to delve deeply into HTTP, and for actual questions about HTTP, there are various references. But for our learning purposes, it is enough to be able to see the common responses from around the web. Moreover, such a client can come in handy during debugging your server, and later on when you are using server-side technologies.

In particular, we would often want to see the headers, which are not shown in a standard browser.

Modify your client and have it connect to some standard web server, e.g. your company’s web server, or some common web server, on port 80. So your client may have a line like:

```
Socket client =
 new Socket("www.bigfamoushost.com", 80 );
```

Once the connection is made, have your client send out

```
GET / HTTP/1.0
```

Followed by an empty line, and then have it read all the bytes coming from the server and print them out, and maybe also write them to a text file.

Solutions folder: 2.8

This browser is a very small program, but it doesn’t really help that much with our core learning objective here, so if you don’t feel motivated to write it, just copy it from the accompanying software. It has also been included in the `support` folder. Check out the code, and make sure you are comfortable using it. Keep it around, you may find it very useful in certain kinds of troubleshooting in this book and even later in actual professional tasks.

Chapter 3
Making your server more dynamic

Here is the next problem - how do we change our server so some other Java programmer, who doesn't know the inside code of our server, can write Java programs to use our server for useful purposes?

For this problem, you have to think with two different hats. So far, you have been using the "server author" hat. Now switch to a server user (who is also a programmer, let us call him/her "the server side programmer") hat, and try to figure out what you would need. Then you can put your server author hat back again, and provide what's needed.

To make our task concrete, let's start with a very simple task. The server side programmer (that's you in the server side programmer's hat) wants to let the website user (one more hat for you) enter his/her name in a form. Then the programmer wants to read that name in a Java program, and have the Java program write a web page that says "Hello, <name>". That's all.

Form Processing

We saw earlier that when a form is processed, the results of the form processing come in as part of the request line, e.g. a form containing a "first" and a "last" field might result in

```
GET /Some.html?first=Julius&last=Caesar HTTP/1.0
```

The "server side programmer" will need to be able to get the information

```
first=Julius&last=Caesar
```

Also, if there were multiple server side programmers, each of them would have to figure out how to do the task of breaking up this string above, and extract the fact that "first" was given a value of "Julius" and "last" was given a value of "Caesar". Instead of having each of them break apart the string above, it would be nice to do it for them, so they can just call a method to ask "What did the user enter for 'first'?"

That's most of what we need do for form processing. In our web server, we don't need to worry about POST, which is very similar.

Output Generation

If the server side programmer has to be able to write dynamic output, we have to give him or her some mechanism to write that output from the Java program.

In addition, we also need to let the server side programmer change the content-type (as he/she may decide after some processing that the correct output type is an audio file, and not an HTML file, for example.)

We need to let the server side programmer send out an error (like "File not found".)

And we need to be able to let him/her "re-direct" the browser, something like "this is not the web page you want, go here instead..."

The MyWeblet class

Java provides convenient mechanisms for letting two parties hookup their Java codes. In this case, the two parties are you in your server author hat, and you in your server side programmer hat, but the same technologies apply.

A common method is that one party provides a class, and the other party sub-classes it and customizes it as per their needs.

This is the method we will be using.

We could also provide some supporting objects to do various tasks, but to keep it simple, we would provide all the support in a single class.

Full-fledged Java servers have servlets. Our little version will be MyWeblets, represented by the class `mvows.MyWeblet`.

This class will have to have an entry point that the server side programmer can override, and where the server side programmer gets notification "Hey, a new request has come in as per your specifications, do what you will with it". In overriding this method, the server side programmer can give some HTML output back to the server author, whose code will send that output over to the client (the browser.)

This class will have a method that can be used to change the content type of the outgoing page. (We will default it to "text/html" for convenience.) It will also have a method to send an error back, and one to redirect the browser.

It will also have a way to retrieve the form data in a nice, already parsed form.

Here is the definition of this class. At this point we don't have the full implementation, we will add some more code as we move along.

```
package mvows;

import java.io.*;
import java.util.*;

public abstract class MyWeblet
{
 // Override this method to process the request
 public abstract void doRequest(
   String resource,
              // The URL, minus query string
   String queryString
              // query string or null
   HashMap<String,String> parameters,
              // Parsed parameters
   PrintWriter out
              // The output from the method
 );

 // Call this method to set content-type,
 // default being text/html

 protected void setContentType(String contentType)
 {
 }

 // Call this method to send back an error.
 protected void setError(int errorCode,
                         String description)
 {
 }

 // Call this method to tell the browser to
 // go to a different URL instead
 protected void sendRedirect( String newUrl )
 {
 }

}
```

Listing 3.1

That's all! Once this class implementation is complete, it is enough to let Java server side programmers write useful Java server side programs using My Very Own Web Server.

We also have to let the server side programmer specify when does the MyWeblet get called. This is done by hooking up the MyWeblet to a URL, e.g. a specification like

```
"/HelloWorld", HelloWorldWeblet.class
```

which saying that whenever the browser asks for `/HelloWorld` (because the user wants the url `http://localhost/HelloWorld`,) the web server should load the class `HelloWorldWeblet`, instantiate an object of that class, and use that object to process the request.

Typically, this will be loaded from an XML config file, but we aren't that much interested in how to read XML files, so we will just add a Java file, let's say `MyWebletConfigs.java`

```
package mvows;

public class MyWebletConfigs
{
 String url;
 Class cls;

 public MyWebletConfigs( String url, Class cls )
 {
   this.url = url;
   this.cls = cls;
 }

 static MyWebletConfigs[] myWebletConfigs =
       new MyWebletConfigs[]  {

   // Add one MyWebLet per line

   // e.g.

   // new MyWebletConfigs(
   //     "/HelloWorld",
   //     myapps.HelloWorldMyWeblet.class ),
 };
}
```

Listing 3.2

The configuration is done within this file, and one line is added as shown.

You may not have come across the syntax `myapps.HelloWorldWeblets.class` - this is how you specify a class in Java (instead of an object of the class.) Given the class, you can create a new instance of the class by calling methods provided in the `java.lang.Class`.

After we write any MyWeblets, we will come back to this `MyWebletConfigs.java` file, and add the MyWeblet.

Your first MyWeblet

Writing MyWeblets is relatively easy compared to the web-server authoring work, so we will start by writing a very simply MyWeblet, and then we will add in the implementation of `mvows.MyWeblet` until the MyWeblet works.

Let us use the package `myapps` to keep our server-side-programmer-hat work in a separate package from our server-author-hat work.

The MyWeblet class we write needs to subclass `myvows.MyWeblet` and it must provide an implementation of the abstract method `doRequest`. In addition, it can call the other methods provided in the class. (But our first MyWeblet will not need to to call any such methods.)

Let us call the class `HelloWorldMyWeblet.class`. Here is an implementation of `doRequest`

```
package myapps;

import java.io.*;
import java.util.*;

public class HelloWorldMyWeblet
                 extends mvows.MyWeblet
{
 public void doRequest(
              String resource,
              String queryString,
              HashMap<String,String> parameters,
              PrintWriter out )
 {
   out.println( "<HTML>");
   out.println( "<BODY>");
   out.println( "<H2>Hello, World</H2>" );
   out.println("Hello from My First MyWeblet");
   out.println( "</BODY>");
   out.println( "</HTML>");
 }
}
```

Listing 3.3

That's it. It should be very obvious what this MyWeblet is doing - it is returning the HTML that it wants the web-server to send back to the client.

Now let us configure it. That just means adding

```
   new MyWebletConfigs(
        "/HelloWorld",
        myapps.HelloWorldMyWeblet.class ),
```

in MyWebletConfigs.java.

Now whenever a browser visits the URL http://localhost/HelloWorld, our MyWeblet code should get called, and the HTML output from the MyWeblet should get sent back to the browser.

Our next task is to make all of that happen.

Looking up the MyWeblet

Going back to the web-server code from chapter 2, in "Serving Files", we took the resource name and looked for a file by that name.

Before we do that, now we need to add a check to see if the resource being named is one of the MyWeblets that have been configured. If it is, then we forget the rest of the processing and let the MyWeblet do the output.

This check is quite simple:

```
 for ( int i = 0;
       i < MyWebletConfigs.myWebletConfigs.length;
       i++ )
 {
   String url =
         MyWebletConfigs.myWebletConfigs[i].url;

   Class cls =
         MyWebletConfigs.myWebletConfigs[i].cls;

   // Compare the url against the resource name

   if ( url.equalsIgnoreCase( resource ))
   {
     MyWebletProcessor mwp =
                      new MyWebletProcessor();

     mwp.processMyWeblet( cls, outputWriter );

     // Don't do the rest of the processing,
     // processMyWeblet has done it all

     inputStream.close();
     socket.close();

     return;
   }
 }
```

Listing 3.4

We are looping through our array of configured MyWeblet's, and if we find any with a `url` that matches the current `resource`, then we are going to use that MyWeblet. Otherwise, we will just fall through to the rest of the code, for the usual processing.

And now we need to write the class `MyWebletProcessor` containing the method `processMyWeblet`. It is an unusual method in that its parameter is actually a class. But we can pass classes around just like anything else in Java.

```
package mvows;

import java.io.*;
import java.util.*;

public class MyWebletProcessor
{
 void processMyWeblet( Class cls,
                       PrintWriter outputWriter )
 {
   try {
     Object instance = cls.newInstance();
         // This makes a new object of class 'cls'

     MyWeblet myWeblet = (MyWeblet) instance;
       // The class 'cls' subclasses
       // the mvows.MyWeblet class,
       // therefore we can cast the object down

     ByteArrayOutputStream byteArray =
                 new ByteArrayOutputStream();
```

```
      PrintWriter out =
                    new PrintWriter( byteArray );

      // For now, let's ignore the other parameters
      // to "doRequest" except "out"

       myWeblet.doRequest( null, null, null, out );

      // When "doRequest" returns, the output is in
      // the ByteArray "byteArray"

      // Send the default headers
      outputWriter.println("HTTP/1.0 200 OK" );
      outputWriter.println(
                  "Content-Type: text/html" );

      outputWriter.println();
                  // End headers with empty line

      // Send content

      out.flush();
         // Make sure all output is in 'byteArray'

      outputWriter.println( byteArray.toString());

    } catch (Exception ex)
    {
      ex.printStackTrace();
    }

    try {
      // Make sure the connection is
      // closed in any case.

      outputWriter.close();

    } catch (Exception ex)
    {
    }
  }
}
```

Listing 3.5

Solutions folder: 3.1

Given the class `cls`, we call `cls.newInstance()` which is the same as doing `new HelloWorldMyWeblet()`, except that at this point in the code, we don't know whether we have to call `new HelloWorldMyWeblet()` or `new SomeOtherMyWeblet()`, but `cls.newInstance()` calls the appropriate constructor for us. Because we know that whatever the class, it has to be derived from `MyWeblet`, we can cast it. The casting is effectively as if we did the following

```
 HelloWorldMyWeblet instance =
          new HelloWorldMyWeblet();
 MyWeblet myWeblet = (MyWeblet) instance;
```

Because `HelloWorldMyWeblet` extends `MyWeblet`, the casting is valid.

Once we have the `MyWeblet` object, we are giving it a `PrintWriter` object, that we made from a `ByteArrayOutputStream`. The `HelloWorldMyWeblet` is going to print some HTML into the `PrintWriter`, and all that HTML will be saved in the `ByteArrayOutputStream` as a byte array.

Putting it all together, now our server should print the output from `HelloWorldMyWeblet` when the URL `http://localhost/HelloWorld` is visited in a browser. We did that by giving a "byte array output output stream" to the MyWeblet. When the MyWeblet is done, we collected all the data from the "byte array output stream" (which data comes in the form of a byte array, obviously) and turned it into a string and sent it out to the client.

Something rather important happened in the handling of the `output` variable above. Why didn't we just pass along the `outputWriter` to the myWebLet? Look it over, make sure you understand how that would have simplified the code.

We could have done that. But note that the order of the output is

```
HTTP/1.0 200 OK
```

followed by the content-type

```
Content-Type: text/html
```

followed by an empty line. And only then can we send out the output of the MyWeblet.

So if we wanted to pass along `outputWriter` to the myWebLet, that would have worked fine, except that we would have had to send these headers and the empty line before we called the myWebLet.

That would have worked for now - but remember that later on, we will allow the myWebLet author to change the content-type, or even the success-code, in which case that mechanism would not have worked.

So instead of passing along the `outputWriter`, we are sending a `PrintWriter` that actually sends its output to a `byte array`, and we can get that output from the `byte array` at the appropriate time. By that time, we know the content-type, so we can send that before sending the output from the `byte array`.

Full-fledged Java application servers, in fact, usually do pass along their version of `outputWriter` to the servlets, and face the exact same problem. The typical solution is more complicated than what we did above, because what we did above has a significant efficiency concern. It uses up a `byte array` that could get very large in some cases.

The solution taken in typical Java application servers to get around this problem has some complications, that we will be discussing when discussing the servlets. Understanding the code above will help in understanding the very exact nature of those complications.

Changing the Content-Type from your MyWeblet

In `processMyWeblet` we are still using "text/html" as the content type. This is ok for default, but we want to let the server side programmer be able to change the content type.

This is quite easy, all that's required is to have a `contentType` member in the abstract class `mvows.MyWeblet`, and to have the `setContentType` method change this.

When it is time to output the content-type, check the content type as set in the `myWeblet` variable - if it's null, output text/html as always, otherwise output the content type.

To test this, change HelloWorldMyWeblet to set content-type to "text/plain".

Now when you visit the HelloWorldMyWeblet, you should see the HTML markup as well (because you are viewing it as text, not as html.)

Redirecting and Error Returns

Let us add the line

```
sendRedirect( "http://someurlhere/" );
    // Put some real world URL here
```

to the test program, the HelloWorldMyWeblet.

In the abstract class `mvows.MyWeblet`, save the redirect in a member variable.

Now in `processMyWeblet`, check if the "redirect" is set. If it is, do not send the "200 OK" header, instead, send

```
HTTP/1.0 302 Found
```

for a header, and then (before sending the empty line), send a header of the form

```
Location: <<the redirect url>>
```

The "redirect url" is the one saved from the call to sendRedirect, e.g.

```
Location: http://someurlhere/
```

Do not send anything after the empty line.

Give this a try, when this is working correctly, it should cause the browser to visit that URL.

Error Returns are handled similar to redirections, instead of the "200 OK" header, send the appropriate error header with description, e.g.

```
HTTP/1.0 404 Not Found
```

Solutions folder: 3.2

QueryString and Parameters

Above, we passed `null` values to `doRequest` instead of the `resource`, `queryString` and `parameters`.

It's time to fix this and pass in the actual values.

We already have the `queryString` from the previous chapter. Just pass the resource string without the query String, and the queryString to `processMyWeblet`. For the parameters, we need to a `HashMap` in `processMyWeblet` from breaking up the query string into individual parameters. The following method will parse the query string into parameters:

```
HashMap<String,String> parseQueryString(
            String queryString )
       throws UnsupportedEncodingException
{
  HashMap<String,String> parameters =
                new HashMap<String,String>();

  if ( queryString == null )
    return parameters;

  StringTokenizer qtokens =
         new StringTokenizer( queryString, "&" );

  while ( qtokens.hasMoreTokens())
  {
    String[] ptokens =
             qtokens.nextToken().split("=");

    if ( ptokens.length == 2 )
    {
      String parameterName =
         URLDecoder.decode( ptokens[0], "utf-8" );

      String parameterValue =
         URLDecoder.decode( ptokens[1], "utf-8" );

      parameters.put( parameterName,
                      parameterValue );
    }
  }

  return parameters;
}
```

Listing 3.6

The code above is just some string manipulation, so it is included in the `support` folder. All it is doing is turning the query string to a `HashMap` containing the parameter values from the query string.

The code is converting escape sequences like `%20` etc. by calling the `URLDecoder.decode` method on the parameter names and parameter values. This requires importing `java.net.URLDecoder`. The `decode` method throws `UnsupportedEncodingException`, but we don't need to worry about that because "utf-8" is supported.

We can now write a MyWeblet that uses the parameters. The task is simple, we are going to process a form where the user enters a first and a last name, and then we will print a page using the first and a last name as greeting.

Here is the form, EnterName.html

```
<HTML>
<BODY>
  <FORM ACTION="/ProcessName" METHOD=GET>
  Please enter your first name
  <INPUT NAME="firstname" TYPE=TEXT SIZE=10>
  <P>
  Please enter your last name
  <INPUT NAME="lastname" TYPE=TEXT SIZE=10>
  <P>
  <INPUT TYPE=SUBMIT>
  </FORM>
</BODY>
</HTML>
```

Listing 3.7

The form needs to be placed along with other HTML files, in C:\MyOwnServerFiles (or any other location you are using for this purpose.)

Notice the ACTION for this form. We have specified the resource URI `/ProcessName`, so we will have to configure a MyWeblet for the `/ProcessName` URI. That browser will make sure to call that MyWeblet by asking for that URI. The browser will also pass us the values entered for first and last name.

Let us call the MyWeblet class `myapps.NameProcessor`. The initial code fragment for this class is

```
public class NameProcessor extends mvows.MyWeblet
{
  public void doRequest(
             String resource,
             String queryString,
             HashMap<String,String> parameters,
             PrintWriter out )
  {
    out.println( "<HTML>");
    out.println( "<BODY>");

    // Write out the first name and the last name

    out.println( "Hello " +
               parameters.get("firstname")
               + " " +
              parameters.get("lastname"));

    // Add some other dynamic info
    out.println( "<P> The time is now ");
    out.println( new java.util.Date());
    out.println( "<P>The Resource Name is " +
                         resource );

    out.println( "<P>The Query String = " +
                         queryString );
    out.println( "</BODY>");
    out.println( "</HTML>");
  }
}
```

Listing 3.8

The output is simple, but now we are working with the `parameters` as well. We are adding the values of the parameters "firstname" and "lastname" to the HTML output. Make sure to deploy it by adding a line in MyWebletConfig.java

```
  new MyWebletConfigs(
       "/ProcessName",
       myapps.NameProcessor.class ),
```

Now build and restart the server, and when you visit the EnterName.html form using a browser, and put in a first and last name and submit the form, you should see the output generated by this MyWeblet.

Solutions folder: 3.3

Debugging the MyWeblet

For now, you should be able to run and debug the entire server in your IDE. Just start the server in debug mode, and visit the page you are interested in, using a browser.

Later on, when working with full-fledged Java servlets, you won't usually be able to run the full web-server in a debugger. (Sometimes it is possible.) But there usually are mechanisms available to run your code in the debugger. For instance, some IDEs like Eclipse offer a "remote debugging" mechanism, which lets you debug code in a running process by attaching to it.

Note that MyWeblets (and servlets later on) are running as Java program on the server. So one approach to debugging that is always available in local environments, is to print lots of output using `System.out.println` which will print output to the console, and will help you keep track of what is going on.

In the above code, if you are not seeing the first name and the last name correctly, you can dump the whole parameter list to the console by something like

```
System.out.println("Numbers of parameters = " +
                    parameters.size());

for (String paramName: parameters.keySet())
{
  System.out.println("Parameter Name = " +
      paramName +
      ", Value = " +
      parameters.get(paramName));
}
```

Listing 3.9

This can be helpful in catching spelling mistakes etc.

Also, the browser from the end of Chapter 2 is also very useful for debugging, as it will show you the actual headers and content that are being sent back by the server.

Supporting Cookies in MyWeblets

Remember cookies? They can help us tell one user from the next, and are very useful in server side programming. They are used to implement "sessions", and sessions are used to keep track of all the user information.

Here is a quick recap of cookies.

A cookie is "given" to the browser by the server. After that, when that browser visits that same server again, it will tell the server "here is the cookie you gave me". This lets the server tell one visitor from the next.

The request contains a header starting with "Cookie: " if the browser has any cookies to present.

To set a cookie, the server sends a header containing "Set-Cookie: <<name>>=<<value>>".

To implement cookies, let us add the member

```
 HashMap<String,String> requestCookies = null;
```

in the abstract class `mvows.MyWeblet`, to hold any incoming cookies that came in with the request. Then the MyWebletProcessor can initialize this member, and set it to the actual cookies.

Also add the member

```
 HashMap<String,String> responseCookies
    = new HashMap<String,String>();
```

to hold any cookies that the MyWeblet writer may want to send out.

The MyWeblet author can put any outgoing cookies in this, and the MyWebletProcessor can turn these into appropriate "Set-Cookie:" headers.

In the `ServerInstance` class `run` method, where we are looking at the header lines, we need to check if the line starts with `Cookie:`. If it does, save that header line in a variable, and add an extra parameter to `MyWebletProcessor.processMyWeblet` to pass that header line.

The new signature for `processMyWeblet` is

```
   void processMyWeblet(
          Class cls,
          PrintWriter outputWriter,
          String resource,
          String queryString,
          String cookieHeaderLine )
```

where `cookieHeaderLine` contains either null or a header line starting with "Cookie:" if such a header line came in from the browser.

In `MyWebletProcessor.processMyWeblet`, we need to parse the cookie header line, which looks like

```
Cookie: name=value
```

If there are multiple cookies, they will come in the same header line

```
Cookie: name=value; name2=value2
```

The following method will break up the cookie names, and return them as a `HashMap`.

```
 HashMap<String,String> getRequestCookies(
                          String cookieHeaders )
 {
   HashMap<String,String> requestCookies =
       new HashMap<String,String>();

   if ( cookieHeaders == null ||
        ! cookieHeaders.startsWith( "Cookie: "))
   {
     return requestCookies;
     // Empty container, no cookies
   }

   // Remove the leading "Cookie:"
   String lineSubstr =
     cookieHeaders.substring("Cookie: ".length());

   // Tokenize the string, as
   // it may contain multiple cookies

   StringTokenizer cookieHdrTokenizer =
           new StringTokenizer( lineSubstr, ";" );
   while ( cookieHdrTokenizer.hasMoreTokens())
   {
     String[] cookieTokens =
       cookieHdrTokenizer.nextToken().split( "=" );
     if ( cookieTokens.length == 2 )
     {
       // Get the cookie name,
       // make sure to remove spaces around it

       String cookieName = cookieTokens[0].trim();
       String cookieValue = cookieTokens[1].trim();
       requestCookies.put( cookieName,
                           cookieValue );
     }
   }

   return requestCookies;
}
```

Listing 3.10

Again, this is only string processing, and the code is included in the `support` folder. All it is doing, is breaking up a cookie string into its parts, and returning a `HashMap` containing the cookie names and values.

Now `processMyWeblet` needs to pass the `requestCookies` over to the MyWeblet in case the MyWeblet is interested in cookies. So all that is needed is to add

```
   myWeblet.requestCookies =
      getRequestCookies( cookieHeaderLine );
```

and the `MyWeblet.requestCookies` member will now be initialized with any cookies that came in with the request.

To provide our web programmer access to any incoming cookies, let us add a method in the `mvows.MyWeblet` class

```
 public String getRequestCookie(
                  String cookieName )
 {
   return requestCookies.get( cookieName );
 }
```

It will return `null` if a cookie by that name did not come in as part of the request, otherwise it will return the value of that cookie.

Similarly, let us handle response cookies by first providing a method in the `mvows.MyWeblet` class

```
public void setResponseCookie(
            String cookieName,
            String cookieValue )
{
  responseCookies.put( cookieName, cookieValue );
}
```

This method lets the MyWeblet author add any outgoing cookies that need to be set.

In the MyWeblet processor, if `responseCookies` have been added, we have to send them out as a `Set-Cookie` header.

One important change in setting cookies (compared to previous chapter) is that we will also send something known as a "Path" component. The "Path" should be set to the / character.

So we will put in

```
Path=/
```

so the entire cookie header will be like

```
Set-Cookie: <name>=<value>; Path=/
```

e.g.

```
Set-Cookie: MyCookie=cookie-1; Path=/
```

Setting Path to "/" makes the cookies shared across the entire website, since all pages start with the "/" character.

Just like "Path", other items can be added to the cookie string. The full cookie string can be a bit more complicated, with all the available options. But what we have above is enough for getting useful work done. We will get back to cookies later when learning full-fledged Java servlets, and will learn more details, such as cookie expiration and domain issues.

While all cookies come in as a single line, in the outgoing direction, if the MyWeblet programmer wants to send more than one cookies, one header needs to be sent for each cookie. The following code will print out the headers to `output` from the `responseCookies` which contains the cookies as a `HashMap`.

```
void printCookieHeaders(
         HashMap<String,String> responseCookies,
         PrintWriter output )
{
  for (String cookieName:
           responseCookies.keySet())

  {
    output.print("Set-Cookie: " );
    output.print( cookieName );
    output.print( "=" );
    String cookieValue =
        responseCookies.get( cookieName );

    output.print( cookieValue );
    output.println( "; Path=/" );
  }
}
```

Listing 3.11

This needs to be called from `processMyWeblet`, after the `HTTP/1.0 200 OK` line, and before the empty line (so the cookies get sent as part of the headers.) It also needs to be called for the code for redirecting and error handling, because any cookies still need to be set in all those cases.

```
// Print out the cookies as part of the headers
printCookieHeaders(
     myWeblet.responseCookies, outputWriter );

// Code to print the empty line comes next
```

Solutions folder: 3.4

Testing Cookie Support

Ok, now we have added cookie support in our MyWeblets. Time to switch hats and start using this support, by writing a MyWeblet that sets cookies if those cookie don't already exist in the request headers. We can just extend our `NameProcessor` class.

```
static int cookieNumber = 1;
static Object cookieLock = new Object();

public doRequest(
            String resource,
            String queryString,
            HashMap<String,String> parameters,
            PrintWriter out )
{
  out.println( "<HTML>");
  out.println( "<BODY>");

  // Write out the first name and the last name

  out.println( "Hello " +
         parameters.get("firstname") +  " " +
         parameters.get("lastname"));

  // Check if the cookie exists
  String cookie =
            getRequestCookie( "TestCookie" );

  // Set cookie if not found, print out cookie
  // info to the Java console for verification

  if ( cookie == null )
  {
    // Give a unique cookie to each browser

    synchronized (cookieLock)
    {
        cookie = "Cookie_" + cookieNumber++ ;
    }

    setResponseCookie( "TestCookie", cookie );
    System.out.println(
         "Visiting browser does not have cookie");

    System.out.println("Setting cookie to: " +
                       cookie );
  } else {
    System.out.println("Browser has cookie: " +
                       cookie );
  }

  // Add some other dynamic info
  out.println( "<P> The time is now ");
  out.println( new java.util.Date());
  out.println( "<P>The Resource Name is " +
                resource );

  out.println( "<P>The Query String = " +
                queryString );
  out.println( "</BODY>");
  out.println( "</HTML>");
}
```

Listing 3.12

Note the `synchronized` for getting unique cookie numbers. This is important - multiple instances of the MyWeblet could be running simultaneously in multiple threads. Since they all share the static `cookieNumber` variable, it is important to synchronize it when we are increasing the cookie number.

To test cookies, use multiple browsers, e.g. use a Chrome, and Internet Explorer and a Firefox - and visit http://localhost/EnterName.html which should forward to /ProcessName (configured to use the NameProcessor MyWeblet) using each of the browsers. Each browser should have get its own cookie from the MyWeblet, and when the browser visits the same page again, it should get the same cookie back.

If cookies don't work, use the test client (from end of the previous chapter, also included in `support` folder) to see the actual headers that are being sent. In the test client, you can use the hostname "localhost", your server's port (80 or otherwise), and resource `/ProcessName?firstname=First&lastname=Last`, and in the response, you should see the Set-Cookie header.

For example, if you are sending out

```
Set-Cookie: TestCookie:Cookie_4; Path=/
```

instead of

```
Set-Cookie: TestCookie=Cookie_4; Path=/
```

the cookies will not work. You can see and fix such errors using the test client.

Solutions folder: 3.5

Sessions

Once cookies are working well, you are ready to add "sessions" to your MyWeblet.

Note that each browser gets its own unique cookie. We can use this to associate a bunch of objects with that browser. For example, let us suppose we have obtained the first and last name from each browser's users. We can then always greet the user, using the correct first and last name.

To do this, we save the first and last name for each browser, in a session that's specific to that browser. How do we keep a unique session per browser? By giving a unique cookie to that browser, and then using that cookie to retrieve the session whenever the browser visits.

We have already seen how to give a unique cookie to every different visitor's browser.

The "session" is just an object, that can hold things like first name and last name strings. A convenient way to do this is to use a `Hashmap<String,Object>`, which will let the programmer attach objects (called "attributes") to various key string.

Let us define the class for the session

```
package mvows;

import java.util.*;

public class MyWebletSession {
 HashMap<String,Object> data =
      new HashMap<String,Object>();

 public void setAttribute( String key,
                           Object value )
 {
     data.put( key, value );
 }

 public Object getAttribute( String key )
 {
     return data.get( key );
 }
}
```

Listing 3.13

There will be one instance of this class corresponding to each "session".

To get the session from a cookie, we can keep the sessions in another HashMap, where the cookie is a key, and each cookie maps to a `MyWebletSession` object.

But instead of a `HashMap`, a `Hashtable` is more convenient this time, so it will be thread safe. The map needs to be shared by all server threads, because the browser request from the same browser could be getting handled by any one of the threads. Also there could be multiple browsers hitting the server at the same time, and they all could have a session stored, so thread safety is important.

This static `Hashtable` can be added to `MyWeblet` abstract class, where we will be adding our session creation method.

We don't need to create a session for all page requests, or for all MyWeblets. In case the MyWeblet author wants a session, we can provide a method `getSession()` that can be called, adding it to the `mvows.MyWeblet` definition.

When this method is called, if a session has not been added, we wo;; add one. We need to store the session using some cookie as a key. Let us use the cookie name `mwsessionid` for My-WebServer-Session ID. This name needs to be standardized, so the MyWeblet-author doesn't use it by mistake!

Here is the process to create a session, in Java:

```
 static int sessionCookieId = 1;
 static Object sessionCookieLock = new Object();

 static
   Hashtable<String,MyWebletSession> sessionMap =
     new Hashtable<String,MyWebletSession>();

 protected MyWebletSession getSession()
 {
   // First, check to see if we received
   // the session cookie.

   String sessionCookie =
           getRequestCookie( "mwsessionid" );

   if ( sessionCookie == null )
   {
     // The browser doesn't have a cookie,
     // give it one.

     int id;
     // Make up a cookie.
     synchronized (sessionCookieLock)
     {
       // make sure the id's are incremented
       // with thread safety

       id = sessionCookieId++;
     }
     sessionCookie = String.valueOf( id );

     // We just need any unique string for the id.


     // Add the new cookie as an outgoing cookie

     setResponseCookie( "mwsessionid",
                        sessionCookie );
   }

   MyWebletSession session =
            sessionMap.get( sessionCookie );

   if ( session == null )
   {
     // Instantiate a session,
     // put it in sessionMap

     session = new MyWebletSession();

     sessionMap.put( sessionCookie, session );
   }

   return session;
}
```

Listing 3.14

First of all, we are checking for the existence of a cookie named `mwsessionid`. If the browser doesn't have this cookie, we give the browser this cookie (making up a fresh new value using the `sessionCookieId` interger, incrementing it in a synchronized block.)

The we check our hashtable `sessionMap` to see if it has seen that value of the cookie. If not (which would happen when we just made that value up freshly!) then we add a new session to the `sessionMap`.

With that, now we have implemented sessions in MVOWS.

Let us try using it for storing the user's first and last name.

The idea is to have a `MyWeblet` at

```
   http://localhost/SessionTest
```

as the new entry point for our small test application, instead of the `EnterName.html`.

This MyWeblet greets the users by first and last name, retrieving it from the session. If the session doesn't have the first and last name, it sends the browser to our familiar form `EnterName.html` where the user enters the first and last name.

We will also modify the MyWeblet `/ProcessName` which processes the form.

This MyWeblet will now simply add the first and last names to the session, and forward the browser back to `/SessionTest`, which will be happy this time.

Note that the HTML generating code for `/ProcessName` needs to be moved out of there, and into `/ProcessName`, because `/ProcessName` now just does a `sendRedirect` to send the browser to `/SessionTest`.

Here is the algorithm in pseudo code

```
Main entry point is /SessionTest

/SessionTest checks if the first and
 last name are in session.

  If not
    Does sendRedirect to /EnterName.html
  Else
    Prints welcome message using first
        and last name
  End

/EnterName.html
  Is an HTML FORM with ACTION set
  to /ProcessName, and INPUT fields
 firstname and lastname

/ProcessName
  Retrieves first and last name
  from request parameters.

  If first and last name not found
      Does sendRedirect to /EnterName.html
  Else
      Saves first and last name in session
      Does sendRedirect to /SessionTest
  End
```

Users are expected to enter the site at `/SessionTest` URI, in other words at the `http://localhost/SessionTest` URL.

We already have the EnterName.html, and it requires no changes.

The new code for ProcessName (in NameProcessor.java) is much simpler. (You can remove the cookie testing from there now.)

```
public void doRequest(
         String resource,
         String queryString,
         HashMap<String,String> parameters,
         PrintWriter out )
{
  String firstname = parameters.get("firstname");
  String lastname = parameters.get("lastname");

  if ( firstname == null || lastname == null )
  {
    sendRedirect( "/EnterName.html" );
    return;
  }

  MyWebletSession session = getSession();

  session.setAttribute( "firstname", firstname );
  session.setAttribute( "lastname",  lastname );

  sendRedirect( "/SessionTest" );
}
```

Listing 3.15

This MyWeblet is now retriving `firstname` and `lastname` from the parameters entered by the user at `/EnterName.html`. If parameters are not found, we can just send the user to `/EnterName.html` again.

Once we have the parameters, we put them in the session, and send the browser over to `/SessionTest` url.

That's it. We don't need to worry about any output, because in all cases, this MyWeblet is sending a redirect.

The code for the new MyWeblet `/SessionTest` is similar to the old `/ProcessName` but picks up the name from session.

```
public void doRequest(
           String resource,
           String queryString,
           HashMap<String,String> parameters,
           PrintWriter out )
{
  MyWebletSession session = getSession();

  String firstname =
       (String) session.getAttribute("firstname");

  String lastname =
       (String) session.getAttribute("lastname");

  if ( firstname == null || lastname == null )
  {
    // Name is not in session, get via form

    sendRedirect( "/EnterName.html" );

    return;
  }

  out.println( "<HTML>");
  out.println( "<BODY>");

  // Write out the first name and the last name

  out.println( "Hello " + firstname
                  + " " + lastname );

  // Add some other dynamic info
  out.println( "<P> The time is now ");
  out.println( new java.util.Date());
  out.println( "<P>The Resource Name is "
               + resource );

  out.println( "<P>The Query String = "
              + queryString );

  out.println( "</BODY>");
  out.println( "</HTML>");
}
```

Listing 3.16

Here we are trying to retrieve the first and last names from the session. If not found, we are sending the browser to `/EnterName.html` to prompt the user to enter that data.

If found, we just print out the first and last name in the HTML we generate.

Make sure to configure this MyWeblet at `/SessionTest` in the `MyWebletConfigs` class.

That's it. Now give this a test by visiting `http://localhost/SessionTest` in a browser. The first time you visit it in a browser, your name is not in session. Effectively the server doesn't "know" you. So it should ask for your name. Once you provide that, the first and last name are sitting in the session, so it will not send you to the form again if visit it in the same browser again! If you refresh the `/SessionTest` URL or visit some other URL and come back to it, it should already know and display your name.

Also, open up a few different browsers (e.g. an instance of IE, another of Chrome, another of Firefox etc.) Give a different name in each browser. Each browser should remember the name you gave in that browser, and you should be able to intermix which browser you use without affecting the individual session for the browsers.

Solutions folder: 3.6

Deleting Sessions

The way we have it, the sessions will keep getting added to our session map, but will never get deleted!

This will lead to out-of-memory situations, so we need to delete sessions.

How do we know when to delete a session? Even if the user has gone away and is not visiting our site pages, the user could come back!

In practice, a practical solution is used - say, if the user hasn't visited our pages for 20 minutes, the user has probably left the site for good. How do we know if the user hasn't visited our pages for 20 minutes? Well, we assume that "our pages", i.e. the pages we are interested in, are all using a session. So if nobody has asked for a session for 20 minutes, we can delete that session.

We need to track "time of last use" for each session.

The `getSession()` method provides a good point marking the "time last used."

Let's add a

```
   long lastUsed = System.currentTimeMillis();
```

field to the definition of the `MyWebSession` class. We need to initialize it with the time of creation, so it doesn't get deleted immediately if left initialized as zero.

Then in the `MyWeblet.getSession()` method, just before it returns the `session`, mark the timestamp.

```
   session.lastUsed = System.currentTimeMillis();
```

Now we know enough to delete sessions. If the session has not been used for 20 minutes - in our case let us actually make it 2 minutes for convenient testing - we can go ahead and delete it.

For this we need a separate thread, that can be started from the main program in Server.java. This thread just goes through all items in the Hashtable, and deletes all the unused ones. Simple! The thread only needs to wake up once in a while, because the 20 minute expiration doesn't have to be accurate. For a 20 minute expiration, perhaps waking up once every 30 seconds would be fine. Our expiration is 2 minutes, so let us give it a 10 second wait before it loops.

Here is the class definition for this thread.

```
package mvows;

import java.util.*;

public class SessionDeleter implements Runnable
{
 public void run()
 {
   for (;;)
   {
     try {
       Thread.sleep( 10000L ); // Sleep 10 seconds
     } catch (InterruptedException ex) {}

     long expirationTime =
         System.currentTimeMillis() - 120000;
         // Now - 2 minutes

     Enumeration<String> keys =
                 MyWeblet.sessionMap.keys();

     while ( keys.hasMoreElements())
     {
       String key = keys.nextElement();
       MyWebletSession session =
              MyWeblet.sessionMap.get( key );

       if ( session.lastUsed < expirationTime )
       {
         // Not been used for expiration
         // duration, delete session

         System.out.println("Deleting session "
                     + key );

         MyWeblet.sessionMap.remove( key );
       }
     }
   }
 }
}
```

Listing 3.17

This is just a "for ever" loop, sleeping 10 seconds each time through the loop, then checking if any sessions are too old and need to be deleted. Our definition of "too old" is "now - 2 minutes". If the session has not been touched in the last 2 minutes, we delete it by removing it from the `sessionmap`.

This thread needs to be started, so in Server.java, add

```
   new Thread( new SessionDeleter()).start();
```

before the main loop.

That should enable session deletions in your MVOWS.

Solutions folder: 3.7

To test it, visit the /SessionTest from a browser. If you refresh it, it will keep the name. It will keep the name as long as you keep refreshing it rapidly, but if you leave it alone for 2 minutes (with a lag of up to 10 seconds), the session will get deleted, and if then you refresh the page, you will have to enter your name again.

Listener Interfaces

Suppose the Java programmer wants to do something every time the server starts. As the MVOWS designer, how do we provide the programmer such access?

Just like MyWeblets, we can define an abstract class - or since this case is simpler, simply a Java interface. The Java programmer than has to implement the interface and tell us the name of the class, i.e. `register` the class with the server.

When the server starts up, we just call the appropriate method in the registered class.

Here is a simple interface

```
package mvows;

public interface ServerStartup {
   void onServerStartup();
}
```

In our Java programmer hat, we can write a class that implements `mvows.ServerStartup`

```
package myapps;

public class MyServerStartup
       implements mvows.ServerStartup {

 public void onServerStartup()
 {
   System.out.println(
        "TBD: Server startup actions...");
 }
}
```

Listing 3.18

This is just a simple implementation of `mvows.ServerStartup` that just prints out a string. Typical tasks will be more complicated than just printing out a string, but this shows the basic essence of listeners.

Note that the class must have a public default constructor (a constructor without any arguments), because the server must be able to instantiate it. Since we haven't provided any constructors above, the system will add in a default constructor.

To `register` this class, typically the Java programmer will add the fully qualified name of the class `myapps.MyServerStartup` in a configuration file.

For our purposes, it is sufficient to do it the same way we handled MyWeblet registrations.

Let us modify the MyWebletConfigs.java, and in our MVOWS designer hat, add to it

```
   public static String[] serverStartupClasses = {
   };
```

In the Java programmer hat, we edit it as follows

```
 static String[] serverStartupClasses = {
   "myapps.MyServerStartup"
 };
```

This time, we are using a string instead of using a `Class` variable. That is just to show how this done in Java from a string. Also, in full-fledged servers, it is always done from strings, that have been typically configured in XML configuration files.

The server, during startup, simply has to instantiate an object of this class, and call the `onServerStartup` method. This can go right in Server.java.

```
for (String className:
       MyWebletConfigs.serverStartupClasses )
{
  try {
    Class cls = Class.forName( className );
    ServerStartup serverStartupObject =
          (ServerStartup) cls.newInstance();
    // Call the startup method
    serverStartupObject.onServerStartup();
  } catch (Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Note the `Class.forName` which returns a class, given its fully qualified name.

The code above will call `myapps.MyServerStartup.onServerStartup` during server startup. Note the `try` and `catch`. These important, because we don't want an error on the part of the Java programmer to bring down the server.

When you run all this, you should see the message

```
TBD: Server startup actions...
```

on the console.

Solutions folder: 3.8

That concludes our direct experiments with MyWeblets, but we will be using MyWeblets to build MSP (My Server Pages), which will teach us how the JSP (Java Server Pages) technology works.

Chapter 4
My Server Pages

MyWeblets as we have them now, are in fact good enough to do a vast majority of actual dynamic programming tasks. Compared to full-fledged servlets, what they leave out are mostly low-level and not very useful details.

But there is one big problem with them. All the HTML has to be put into Java statements like

```
 out.println("<HTML>");
 out.println("<BODY>");
 out.println("<P>Some text");
 out.println("</BODY>");
 out.println("<HTML>");
```

Writing all HTML like this can get very tedious rather quickly. And if you look at the source HTML for typical modern websites, this is a rather hopeless task, what with the numerous image references, embedded JavaScript, cascading style sheets, and so on...

It is also very tedious to make changes to HTML written as strings inside Java programs. Plus, usually the HTML designer is someone different from the Java programmer. The HTML designer has HTML design skills, which DO NOT include putting everything inside `out.println` statements!

Moreover, there are lot of sophisticated HTML authoring tools out there. But they don't generate MyWeblet Java code, they generate HTML.

So for all these reasons, it seems clear that MyWeblets are not the perfect solution for having dynamic HTML. While they can embed HTML, it is not practical to do significant HTML writing this way.

What about turning this inside out? Instead of having HTML in Java programs, could we have Java fragments inside HTML?

That idea makes a lot of sense. We could have bits of pieces of Java inside HTML pages, and they could generate dynamically changing output. Everything else would stay as HTML, and HTML authors and authoring tools would continue to work with that HTML as usual.

For example, let us suppose we could write the following HTML

```
<HTML>
<BODY>
<P>The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

where the <%= and %> characters are enclosing a Java expression. At runtime, somehow or the other that expression gets evaluated (and turned into a String) in Java, and the

```
<%= new java.util.Date() %>
```

gets replaced by the string generated from that expression, so the browser actually receives something like

```
<HTML>
<BODY>
<P>The time is now Thu Apr 04 20:36:55 EDT 2013
</BODY>
</HTML>
```

And of course, every time a browser visits, it always gets the current time!

This solution would be very useful, and therefore such a solution in fact exists, and is highly used in dynamic server side programming.

In Java, this solution is known as JSP, or Java Server Pages.

We will be implementing a version of this that we will call MSP, for MyVeryOwn Server Pages.

We will keep it very close to JSP, though it will be a subset.

In addition to implementing the Java expressions enclosed by <%= and %> we will also use <% and %> (without the = sign) to enclose general Java code (not expressions, but statements.)

For example, the HTML above could also be written as

```
<%
   java.util.Date date = new java.util.Date();
%>
<HTML>
<BODY>
<P>The time is now <%= date %>
</BODY>
</HTML>
```

Here the <% and %> characters are enclosing a Java statement that declares a `date` variable. Later on, we evaluate the `date` variable as a single expression by enclosing it in <%= and %> characters.

Implementation strategy for MyVeryOwn Server Pages

So how would we implement such a technology, using what we already have?

Well, we already have MyWeblets available, which is our foundation technology. So what we need to do is, to turn an MSP into a MyWeblet. Instead of a person writing the MyWeblet code, we just need an automatic way to turn an MSP into a MyWeblet.

It is not really that hard a task. What we have to do is to take the above HTML, and turn it into a `doRequest` method that looks something like

```
 public void doRequest(
             String resource,
             String queryString,
             HashMap<String,String> parameters,
             PrintWriter out )
 {
   // Output the stuff between <% and %> as it is.

   java.util. date = new java.util.Date();


   // Output all HTML as strings,
   // using output.println statements
   out.println( "<HTML>");
   out.println( "<BODY>");
   out.println( "<P>The time is now " );

   // Output the Java expression between
   // <%= and %> using output.println.
   // Unlike HTML, in this case we don't
   // turn this into strings.

   out.println( date );

   // Output the remaining HTML as strings
   out.println( "</BODY>");
   out.println( "</HTML>");
 }
```

That's it! Make sure you understand exactly what is going on above, as this is the essence of JSP technology.

In addition to <%= expression %> and <% code %>, we will also handle <%! declarations %> where anything between <%! and %> will be output before the method doRequest, as class member variables and class methods.

For example consider

```
<%!
   // Method to compute a number cubed
   int cube( int n )
   {
       return n * n * n;
   }
%>
<HTML>
<BODY>
<P>Cube of 5 = <%= cube( 5 ) %>
</BODY>
</HTML>
```

We will translate this into

```
 // Output the declaration(s) before starting
 // the 'doRequest' method
```

```
// Method to compute cube
int cube( int n )
{
    return n * n * n;
}

public void doRequest(
            String resource,
            String queryString,
            HashMap<String,String> parameters,
            PrintWriter out )
{

  // Output all HTML as strings,
  // using output.println statements
  out.println( "<HTML>");
  out.println( "<BODY>");
  out.println( "<P>cube of 5 = " );

  // Output the Java expression between
  // <%= and %> using output.println.
  // Unlike HTML, in this case we don't
  // turn this into strings.

  out.println( cube( 5 ));

  // Output the remaining HTML as strings
  out.println( "</BODY>");
  out.println( "</HTML>");
}
```

Since are using `out.println` which can print various types of things besides Strings, we don't have to ask the user to give us String valued expressions inside the <%= and %> characters. It could be int or float or String or double or any other Java type, and `out.println` would turn it into an appropriate string first.

Note so far we have been using `println`, but actually, we should be differentiating between `print` and `println`, because in some types of output, such as `text/plain`, that makes a difference. So in this case we should be using

```
  output.print( "<P>cube of 5 = " );
```

i.e. use `print` instead of `println` because a new line does not start between this HTML and the <%= cube(5) %> expression.

All of the above is straight out of Java Server Pages (JSP), and we will implement all of this in MSP (MyVeryOwn Server Pages.) MSP is a straight subset of JSP, and covers the core JSP technology.

Parsing MSP Files

We need to be able to take all the text from an MSP file, and split it into several parts, namely the straight text pieces (expected to be HTML) that needs to be output as strings, pieces of Java code surrounded by <% and %> characters, Java expressions surrounded by <%= and %> characters, and Java declarations surrounded by <%! and %> characters.

Since string processing is not the theme of this book, just use the provided class `MspPart` for breaking up the text from an MSP file. This is included in the `support` folder in addition to the `solutions` folder.

Solutions folder: 4.1

The `MspPart` class has a static method `MspPart.parseMsp` which takes a `java.io.File` as argument. The file is assumed to contains MSP code. The `MspPart.parseMsp` method will read the contents of the file, and break up into a list of MSP Parts. The method signature is

```
   public static List<MspPart>
        parseMsp(File file)
             throws IOException
```

The `MspPart` class also contains the following

```
   public enum MspPartType
   { Text,
     TextLine,
     Code,
     Expression,
     Declaration };

   public MspPartType partType;
         // The 'type' of the MSP part

   public String partText;
         // The text of the MSP part
```

Let us start with the simple MSP

```
<%
   java.util.Date date = new java.util.Date();
%>
<HTML>
<BODY>
<P>The time is now <%= date %>
</BODY>
</HTML>
```

Listing 4.1

Let us put this MSP in `C:\MyOwnServerFiles` where all the HTML files reside, and let us call it `CurrentTime.msp`.

The following code fragment will print out the components of this MSP file.

```
 File file =
    new File(
      "C:\\MyOwnServerFiles\\CurrentTime.msp" );
 List<MspPart> parts = MspPart.parseMsp( file );
 for ( MspPart part: parts )
 {
   switch ( part.partType )
   {
     case Text:
       System.out.println( "Text : " +
                           part.partText );
```

```
        break;
      case TextLine:
        System.out.println( "TextLine : " +
                            part.partText );
        break;
      case Declaration:
        System.out.println( "Declaration : " +
                            part.partText );
        break;
      case Code:
        System.out.println( "Code : " +
                            part.partText );
        break;
      case Expression:
        System.out.println( "Expression : " +
                            part.partText );
        break;
    }
  }
```

Listing 4.2

The output of running this code is

```
Code :
   java.util.Date date = new java.util.Date();

TextLine :
TextLine : <HTML>
TextLine : <BODY>
Text : <P>The time is now
Expression :  date
TextLine :
TextLine : </BODY>
TextLine : </HTML>
```

The next task is to take these parts of an MSP, and write a program that will write another program! The program that's being written automatically, is Java code for a MyWeblet.

The algorithm for turning the parts of this MSP automatically into good Java MyWeblet code, is rather simple.

For a `Text` part, output the text as a string using an `out.print` statement and by enclosing it in double quotes. (To avoid trouble with embedded quotes and backslash characters etc. within the string, the MspPart.parseMsp method already escapes those for Text and TextLine parts.)

The `TextLine` part is the same, except use an `out.println` statement.

For a `Code` part, just output the text as is, with no changes.

For an `Expression` part, output the text using `out.print`, but do not enclose in double quotes.

We will handle `Declaration` MSP parts later.

We also have to generate some Java code at the beginning and the end, before we get into handling the parts of the MSP.

Generating Java code from the parsed MSP

First task is to get a class name. This can be complicated involving naming rules and all, so we will adopt a simple convention - we will only allow MSP filenames that will also be valid Java class names. So there can be no spaces and special characters in our MSP Filename, and it cannot be named after a Java reserved word.

Where to place the file also needs to be resolved. For our purposes, we can generate the Java file in the same location where the HTML files are and where the MSP file will be (e.g. C:\MyOwnServerFiles folder.)

Now let us examine one of our MyWeblets, for instance NameProcessor.java, and write code to generate something similar, except that we should use a different package name, let us say `msp` to keep these separate.

Given an MSP file `mspFile`, the code to begin writing the matching Java file is

```
// First of all, parse the MSP file
List<MspPart> mspParts =
        MspPart.parseMsp( mspFile );

// Now create a Java file

File javaFile = new File(
         mspFile.getAbsolutePath().replace(
             ".msp", ".java" ));

String className =
   javaFile.getName().replace( ".java", "" );

PrintWriter javaWriter =
  new PrintWriter( javaFile );

// Write out the class

javaWriter.println( "package msp;" );
javaWriter.println();
javaWriter.println( "import java.io.*;" );
javaWriter.println( "import java.util.*;" );
javaWriter.println( "import mvows.*;" );
javaWriter.println();
javaWriter.println( "public class " +
                    className       +
                    " extends mvows.MyWeblet" );
javaWriter.println( "{" );

// TBD: The "Declaration" parts
// need to be handled here

// Now let us output the 'doRequest' method.

javaWriter.println( "  public void doRequest(" );
javaWriter.println( "    String resource," );
javaWriter.println( "    String queryString," );
javaWriter.println(
      "    HashMap<String,String> parameters," );
javaWriter.println( "    PrintWriter output )" );
javaWriter.println( "  {" );

// CODEGEN: This is where we output
// the code for doRequest.

javaWriter.println( "  }" );
javaWriter.println( "}" );

javaWriter.close();
```

Listing 4.3

This code is physically creating a .java file on the hard drive, and deriving a class name for that.

Then it starts coding the .java file, pretty much you or I might code it. It writes `package msp;` at the top, then writes out the `import` statements that will be needed, and then starts out with the class declaration

```
public class <<classNameHere>>
    extends mvows.MyWeblet
{
```

This is followed by writing code for the `doRequest` method.

If you run all this code, you should see an empty framework for the `doRequest` method.

Now we have to fill in this framework, and put some code-generation code where it says `CODEGEN` in the above comments.

The code-generation is as per the algorithm above.

```
for ( MspPart part: mspParts )
{
  switch ( part.partType )
  {
    case Text:

      javaWriter.println(
          "    output.print( \"" +
          part.partText          +
          "\" );" );

      break;

    case TextLine:

      javaWriter.println(
         "    output.println( \"" +
         part.partText            +
         "\" );" );

      break;

    case Declaration:
      // NOt handled here
      break;

    case Expression:

      javaWriter.println(
         "    output.print( " +
          part.partText +
          ");" );
         // No double quotes this time

      break;
    case Code:
      javaWriter.println( part.partText );
          // Just print it out
      break;
  }
}
```

Listing 4.4

That's the basic algorithm for turning an MSP into a MyWeblet (and also for turning a JSP into a servlet.)

To handle the declarations, a separate pass is needed. Before starting to write the `doRequest`, just loop through the `mspParts` and output all the declarations.

```
for ( MspPart part: mspParts )
{
  switch ( part.partType )
  {
    case Declaration:
      javaWriter.println( part.partText );
      break;
    }
  }
```

Listing 4.5

That's it!

Now if you get all this to work on C:\MyOwnServerFiles\CurrentTime.msp, your code should generate a valid MyWeblet Java file. Examine this file and make sure your program has written a correct program!

Solutions folder: 4.2

Compile this MyWeblet file as a part of your server, and configure it as usual at some URL, e.g. /CurrentTime, and test it out by vising http://localhost/CurrentTime in a browser.

Solutions folder: 4.3

Your My-Very-Own-Web-Server has just handled an MSP file!

Automating the MSP deployment

Above, we manually ran an MSP processor, compiled the output of the MSP processor (a MyWeblet Java file) as a part of the server, and configured the resulting Java class manually in the configuration file.

It's not reasonable to rebuild the server for every MSP file, of course.

But turns out, we can actually easily automate all of that in Java with a little bookkeeping.

The method `Runtime.exec` lets us run commands from Java. So after we have built a Java file from an MSP file, we can run `javac`, the Java compiler, from inside our server using `Runtime.exec` and compile that Java file.

The `javac` will end up creating a `.class` file. In Java we can load that class file into a class in the running server, by defining a subclass of the `ClassLoader`.

You don't need to understand the details of class loading, and can just use the provided solution in the `MspClassLoader.java` file. It has been included in the `support` folder. It's not very complicated, and just involves reading the bytes of the class file and calling `defineClass` and `resolveClass` methods.

Solutions folder: 4.4

Using `MspClassLoader`, we will be able to automate the MSP processing.

Naturally, we don't want to do the time-consuming MSP-to-Java translation and the compilation and class-loading every time we see an MSP. That will make the whole system rather slow and inefficient. We want to do this when (1) we haven't already loaded a class for a given MSP file or (2) we did load a class for a given MSP file, but the MSP has changed since the time we loaded the class.

It's time to start our `MspProcessor` class, which, given a MSP file, does all of the above automatically. If necessary, it creates a matching Java file, compiles that Java file, loads the class, and processes it as a MyWeblet, and sends the result back to the browser. But if all of that is not necessary, it just uses the previously loaded class for that MSP as a MyWeblet, and handles the request.

To keep track of the loaded MSP classes, we need to keep a `Hashtable` (since this is a multi-thread operation, using `Hashtable` rather than `HashMap` avoids some multi-threading safety issues.)

We also need a small inner class within `MspProcessor` to keep track of the loaded class as well as the timestamp. (The package for `MspProcessor` is `mvows` because it needs to work with the previous `mvows` classes. We are using the package name `msp` for the Java files generated automatically.)

```
private static class MspInfo
{
  public Class mspClass;
     // The loaded class
  public long lastModified;
    // The timestamp of the MSP file
}
```

Listing 4.6

When given a new MSP file, the first step is to see if we already have a valid `MspInfo` object for it. If not, we need to build it and save it in the Hashtable.

```
Hashtable<File,MspInfo> mspTable =
                new Hashtable<File,MspInfo>();

public Class getMspClass( File mspFile )
                throws IOException
{
  // Check if we already have an mspInfo

  MspInfo info = mspTable.get( mspFile );
  if ( info != null &&
       info.lastModified ==
              mspFile.lastModified())
      return info.mspClass;

  // Don't have the class in memory,
  // or it's out of date, find or build it

  Class cls = loadMspClass( mspFile );

  info = new MspInfo();
  info.mspClass = cls;
  info.lastModified =
               mspFile.lastModified();

  mspTable.put( mspFile, info );
         // Save for future in Hashtable

  return cls;
}
```

Listing 4.7

The `loadMspClass` method first of all needs to check if there is already a valid .class file. (It could be there since the previous time the server was run.) If not, the MSP needs to be processed into a Java file, and compiled. In either case, the .class file needs to be loaded.

There needs to be a convention of where to keep the class file. A simple solution in our case is to put it in a folder called C:\MyOwnServerFiles\msp, as that will conform to the Java package hierarchy convention.

```
Class loadMspClass( File mspFile )
       throws IOException
{
  // Get the path of the .class file
  String classFileName =
    mspFile.getName().replace( ".msp", ".class" );

  File classFile = new File(
        new File( "C:\\MyOwnServerFiles\\msp" ),
        classFileName );

  // If classFile doesn't exist, or has a
  // timestamp older than mspFile, build
  // a new ClassFile

  if (( ! classFile.exists()) ||
      classFile.lastModified() <
                mspFile.lastModified())
  {
      System.out.println("Building MSP " +
                mspFile.getAbsolutePath());

      buildMspClass( mspFile );
  }

  // At this point, the classfile either
  // already exists or has been built.
  // Load it.

  MspClassLoader classLoader =
          new MspClassLoader( classFile );

  return classLoader.getLoadedClass();
}
```

Listing 4.8

Note that the `MspClassLoader` cannot be reused because the same class cannot be defined more than once within the same class loader. But because we plan to rebuild and recompile the MSP file every time it changes, we will need to load the same class multiple times. A simple solution for this is simply to instantiate a new `MspClassLoader` every time, which will be discarded and garbage collected when no longer in use.

What is missing now is the `buildMspClass` method.

The `buildMspClass` method needs to (1) build the Java file and (2) compile it.

```
void buildMspClass( File mspFile )
            throws IOException
{
  // Build the java file
  File javaFile = buildJavaFileFromMsp( mspFile );

  // Compile it.

  compileJavaFile( javaFile );
}
```

Listing 4.9

We already have the code for `buildJavaFileFromMsp` to turn the MSP file into a Java file. This is the same code that was used to test the `MspParts` parser.

Once we have the Java file, to compile it into a class file, we use `Runtime.exec` method.

Before doing that, we need to make sure we have the correct command line. On Windows, the command line will be something like `c:\somepath\javac.exe -classpath classpath -d c:\MyOwnServerFiles absolute-path-to-source-java-file` The classpath needs to include the location where the server classes, e.g. mvows.MyWeblet and mvows.Server etc. are located.

Make sure you get this string exactly right from a command line, before trying to add this into a program.

Then try it out from a small stand-along program to make sure it works and puts the class file in the right place.

```
  Process p = Runtime.getRuntime().exec(
     new String[] {
        "<path-to-javac>\\javac",
        // E.g. "c:\\jdk1.6.0_01\\bin\\javac.exe"
        "-classpath",
        "<classpath, including mvows>",
                 // E.g. "C:\\MVOWS\\classes"
        "-d",
        "C:\\MyOwnServerFiles",
        javaFile.getAbsolutePath()
   }
  );

  // It is important to read the 'error'
  // output to catch compiler errors

  InputStream in = p.getErrorStream();
  int c;
  while (( c = in.read()) >= 0 )
      System.out.print( (char) c );
  try {
      p.waitFor();
      // Wait until the compile is done
  } catch (InterruptedException ex) {}
```

Listing 4.10

These steps need a lot of testing. Make sure that given an MSP file, your code can take it all the way to loading a class for it. To test that the class is successfully loaded, make sure you can print its name.

That completes the automation of the MSP files.

Solutions folder: 4.5

Important: Note that the provided solution needs to be modified for the actual path to javac and the actual classpath.

Hooking up the MSP processor to the server

Now that the MSP Processor can turn any valid .msp files into a Java class, we need to hook this up to the server.

In `ServerInstance`, we have a loop through `MyWebletConfigs.myWebletConfigs` to search for any matching MyWeblets, and then we check to see if the resource file exists.

After we have verified that the resource file exists, check to see if ends with `.msp`, and if it does, use the `MspProcessor` to load a Java class for it. The rest of the processing is exactly identical to `MyWeblet` processing, because we have actually turned the MSP into a `MyWeblet`.

```
 if ( file.getAbsolutePath().endsWith(".msp"))
 {
   Class cls =
         new MspProcessor().getMspClass( file );

   MyWebletProcessor mwp =
         new MyWebletProcessor();

   mwp.processMyWeblet( cls,
                        outputWriter,
                        resource,
                        queryString,
                        cookieHeaderLine );

   // Don't do the rest of the processing,
   // processMyWeblet has done it all

   inputStream.close();
   socket.close();

   return;
}
```

Listing 4.11

Solutions folder: 4.6

That's it! Now you can put .msp files in C:\MyOwnServerFiles folder, and just visit them in a browser, e.g. http://localhost/Hello.msp where Hello.msp contains

```
<HTML>
<BODY>
Hello, the time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

There is no configuration required. All configuration is happening automatically.

As long as you don't change the .msp file, if you reload it in the same or a different browser, there should be no rebuilding involved.

When you change it, the server should automatically recognize it, and rebuild and reload it and display the results with the changes.

Your server now has the MSP technology.

The "out" and other reserved variable names

Recall that when we expand an MSP into Java code, we put everything inside a method, e.g.

```
 public void doRequest(
             String resource,
             String queryString,
             HashMap<String,String> parameters,
             PrintWriter out )
 {
     // All the code here is automatically
     // generated from the MSP
 }
```

In this, the MSP is free to include not just Java expressions, but also regular Java code using the <% and %> characters.

What if that regular Java code included a declaration of an `out` variable? This will cause a naming conflict, because we have already named one of the method parameters as `out`!

So we have to warn the MSP author to not use `out` or any other variables we have already used up!

But wait, let's rethink it. Could this be declared a feature, not a bug?

Actually, it certainly could. Instead of telling the MSP author that they have this restriction about not using `out` as a name within the Java code - we could tell them instead they have this amazing `out` variable that we have provided for them, and that they could use to `print` or `println` to the web page output!

This is a good and in fact a very useful solution. Instead of always having to drop out of the Java code using <% and %> in order to write HTML output, the MSP authors could just write HTML straight from the Java code, to the `out` variable. E.g.

```
Some html.
<%
   // Some Java computations
   out.println("Some <b>more</b> html".)
%>
```

Here first of all, the MSP processor turns the

```
Some html.
```

into

```
   out.println("Some html.");
```

Then the Java code is included as is, which just becomes

```
   // Some Java computations
   out.println("Some <b>more</b> html".)
```

The effect of both `out.println` is the same, they both output the strings to the web page output.

We can (or rather, must) similarly make other variables available for MSP authors, namely `resource`, `queryString` and `parameters`, which is not a bad deal as these are all useful to an MSP author.

To retrieve a parameter at any point, the MSP author can simply do a

```
   parameters.get("paramName");
```

within the MSP.

A few server pages

Now that your server has the MSP technology, it's time to take it out for a few test runs.

Here are some tests.

The following MSP will compute the cube of a number, using an MSP declaration. Let us call it `cube.msp` and give it a try.

```
<%!
   // Method to compute cube
   int cube( int n )
   {
       return n * n * n;
   }
%>
The cube of 6 = <%= cube( 6 ) %>
```

Listing 4.12

Let us now change this MSP to accept an argument, a number for which to print the cube.

For this, we need to have an HTML form where the user can enter the number in an INPUT field.

Note that our server doesn't handle POST, so the METHOD of a FORM tag must always be GET.

```
<HTML>
<BODY>
<FORM METHOD=GET ACTION="cube.msp">
Enter number: <INPUT TYPE=TEXT SIZE=4 NAME="number">
<P>
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Listing 4.13

We also need to modify the `cube.msp` to retrieve the number the user entered on this form. If the user didn't enter a valid number, an error message needs to be printed out, otherwise the cube of the number needs to be shown.

The `cube` method can be left alone, we just need to modify the line that prints out the results.

```
<%
try {
   int n = Integer.parseInt(
              parameters.get( "number" ));

   out.print("The cube of " + n + " = " +
              cube( n ));
} catch (NumberFormatException nex)
{
   out.print("Please enter a valid number." +
             " You entered: " +
             parameters.get( "number" ));
}
%>
```

Listing 4.14

Note that here it was more convenient programmatically to use the `out` variable, while keeping everything inside <% and %> characters.

This is a programmer choice, whether to use the `out` variable or to generate output directly. Just use whichever is more convenient.

Another important thing to note is that there was no configuration involved. Unlike MyWeblets, MSPs are very easy to work with. You just add them to the documents folder, and then you can access them from the browser. If you want to change them, you just change and save them - the server handles all the bookkeeping and configuration issues.

This makes MSPs very easy to use. On the other hand, they are much harder to debug. Therefore, in large programming tasks, frequently the JSPs (their equivalent of our MSPs) are used to start things off, but the heavy-duty Java is done in methods and classes that are accessed from the JSPs.

The difficulty of debugging MSPs (or JSPs later) should not dissuade you from taking advantage of what they do provide, which is (1) ease of writing HTML programs and (2) ease of configuration and modification.

Just remember that if you start writing complex Java inside an MSP or a JSP - you should step back a bit, and consider pulling that Java into a separate new or old class, and just invoking it from the MSP or the JSP.

For simple Java pieces, however, the server pages are an excellent technology.

Solutions folder: 4.7

A server page that targets itself

MSPs have the dual advantage of making it easy to write HTML, as well as doing server side programming.

This makes it easy for them to make themselves the target of their own FORM tags!

For example, the above code fragment to compute the cube of a number, could be handled with a single MSP which does double duty as a FORM as well as providing the computing.

For this, we simply add the HTML right in the MSP. But before the HTML, we check to see if a "number" parameter has been passed along. If it has, we do the computation and output the result!

Here is the new cube.msp. Note the form "ACTION" loops back to itself.

```
<HTML>
<BODY>
<%!
   // A method to return cube of a number

   int cube( int n )
   {
       return n * n;
   }

%>
<%
if ( parameters.get( "number" ) != null )
{
   try {
       int n = Integer.parseInt(
                   parameters.get( "number" ));

       out.print("The cube of " + n + " = " +
                    cube( n ));
   } catch (NumberFormatException nex)
   {
       out.print("Please enter a valid number." +
          " You entered: " +
          parameters.get( "number" ));
   }
   out.println("<HR>");
}
%>
<FORM METHOD=GET ACTION="cube.msp">
Enter number: <INPUT TYPE=TEXT SIZE=4 NAME="number">
<P>
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
```

```
</HTML>
```

Listing 4.15

Now we have combined the two tasks within one MSP.

Solutions folder: 4.8

Now let us use our new MSP technology with our earlier task of remembering the user's name, from the previous chapter.

Sessions and other features in MSPs

Because an MSP just gets turned into a MyWeblet, and subclasses `mvows.MyWeblet`, all the methods of `mvows.MyWeblet` are available to the Java code inside an MSP.

That includes the `getSession` method, therefore using a session from an MSP is the same as using it from a MyWeblet.

This makes it very straightforward to write an MSP that gets a user's name from a session, and redirects to a FORM if the name has not been provided in the session. Again, redirecting the browser is the same as in MyWeblets, because the `sendRedirect` method is available within the MSP.

Compare this solution to the MyWeblet version. While the code is essentially similar, with MSPs there is no configuration involved, and HTML output generation is much simpler, thus balancing the problem of the poorer development and debugging environment.

Solutions folder: 4.9

Note that in the above sample, `EnterName.html` needs to have its target changed to `NameProcessor.msp`.

While the sample code uses `EnterName.html` as previously (to demonstrate `sendRedirection`), it is not required to have a separate HTML form. If necessary, the `NameProcessor.msp` could have conveniently included the form as well, and shown it if the name was not already present in the session.

Solutions folder: 4.10

The sample also shows conditional output of HTML pages. Here, the same MSP is generating one of two HTML pages, either a FORM or a display page, depending upon input conditions.

This concludes My Very Own Server. Feel free to conduct more experiments with it, as it can help to understand other concepts ahead.

Chapter 5
Getting started with a full-fledged Java server

In this book, we have already taken a very in-depth look at the core Java server side technologies, and already have a strong framework of understanding. This framework will continue to support your knowledge acquisition in the future.

In this chapter, we will do a brief theoretical overview of some of the main components of the full-fledged Java server side technology.

In the following chapters, we will look at the details of these components as used in Java servlets (which correspond to our MyWeblets) and JSP (which correspond to MSP) technologies, and the details of configuration.

We will not cover the various interfaces in full detail. Javadoc for them is easily available on the web, as are detailed explanations.

Instead, we will cover the type of features that are available, and why, and how they relate to what we have already learned and understood via our MVOWS work.

The Request Interface

Recall that when a browser asks the server for an HTML page or another resource, it sends some lines of the general form

```
GET / HTTP/1.1
Host: localhost
Connection: keep-alive
<<and a lot more gobbledygook…>>
```

All of these lines constitute the browser's "request". While in MVOWS we only dealt with the most useful aspects of the request, in a more full-fledged environment, all of the request needs to be made accessible for the programmers. Given any obscure part of the request, it's likely sooner or later some programmer somewhere is going to need to deal with it!

Also, in MVOWS and this book we are dealing with only HTTP requests, which are what power the web. At present HTTP is the main technology that we need to address. But theoretically it's possible that other types of requests may come over the socket, which are similar to HTTP in certain ways but not identical.

In Java server side technology, the basic interface to access the browser's request is `ServletRequest`. This is further subclassed by `HttpServletRequest`, which adds more HTTP protocol-specific methods.

Because `HttpServletRequest` extends `ServletRequest`, it has all the methods related to the request object. We only need to worry about HTTP protocol, therefore we do not need to differentiate between `HttpServletRequest` and `ServletRequest`, and we can assume that we are always working with `HttpServletRequest`.

Request Headers

We have been using the term `browser` rather loosely. Actually, the request need not always come from a regular browser, but may come from various other sources, e.g. it may come from a program you wrote, or the `client` program provided in the `support` folder, or it may come from Google's web crawling bots. The generic term for the calling program is `User Agent`.

The request object is expected to contain an identification string for the `User Agent`. This can be retrieved programmatically via the `getHeader` method, e.g.

```
   String userAgentString =
        request.getHeader( "User-Agent" );
```

The user-agent string comes in the "gobbledygook" part of request as something like

`User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:13.0) Gecko/20100101 Firefox/13.0.1` This is known as a "header".

The string "User-Agent" is the name of the "header", and everything after the colon character is the value of that header.

Other headers, as we have seen, may be "Host", "Connection", "Cookie" etc.

Using `HttpServletRequest.getHeader`, a programmer can obtain the value of any of the headers that came in as a part of the request. The names of all available headers in the request can be obtained via the method `getHeaderNames`.

Some of the headers contain integer or date values. The `HttpServletRequest` interface contains methods to retrieve and parse these values into standard Java integers or date objects.

Request Line

As we saw during MVOWS implementation, the very first line

```
GET / HTTP/1.1
```

is quite important. Therefore the `ServletRequest` and `HttpServletRequest` interfaces provides multiple methods to get to different parts of it.

`ServletRequest.getProtocol` returns the "HTTP/1.1" or "HTTP/1.0" string. These are the HTTP protocols commonly available at the time of this writing. It is conceivable that at some point other versions of the HTTP protocol will become available, or that `ServletRequest.getProtocol` will return one or more non-HTTP protocols.

The "GET" and the "/" in the request line above, are HTTP-protocol specific. Therefore, they are available via the `HttpServletRequest` interface. The method `HttpServletRequest.getMethod` returns the method, in this case "GET".

Turns out, the "/" is much more complicated.

This is the "resource URI" part of the request. As we have seen, the "/" may be turned internally into something like C:\MyVeryOwnWebserver\index.html. The "/" part, in practice will often refer to longer other resources, e.g. "/Test.html" which refers to a Test.html file, or "/Test.html?myQueryString" which is still referring to Test.html but with a "query string", or "/Test.html?x=a&y=b", which is still referring to Test.html and has a query string but the query string now consists of parameters named "x" and "y" with values of "a" and "b".

The `HttpServletRequest` interface provides the `getRequestURI` method to get the URI, excluding the query string if any.

The query string itself can be retrieved via another method, of course, named `getQueryString`.

Other methods are also available to parse parts of the URI, to match up against physical paths, etc.

Parameters

Very frequently in server side programming, we are interested in the parameters that are passed in. For example, we had the "firstname" and "lastname" parameters passed in via the `EnterName.html` form, and we processed it via a MyWeblet and an MSP.

The parameters could come in as a part of the request URI as we saw. This is how they get passed for GET requests.

We did not handle POST requests in MVOWS. In POST requests, the parameters are similar, e.g.

```
x=a&y=b
```

but are not passed in as a part of the request line. Instead, the parameters follow the headers and the empty line at the end of the header, so the data sent from the browser could be

```
POST /Test.msp HTTP/1.1
Host: localhost
Connection: keep-alive
Content-Length: 7

x=a&y=b
```

Note the "Content-Length" header above. It is describing the length of the content, which in this case is the string "x=a&y=b", and is of length 7.

The content is just sent as is, and is not terminated by an end-of-line. (Which was the reason for skipping it in MVOWS - the rest of MVOWS fits nicely into neatly separated lines, and while learning how to handle POST requests would have added slightly to the learning imparted by MVOWS, it's not useful enough to justify the added complications. So consider the above information a rounding up of your MVOWS education, and make sure you understand how POST requests are passed in.)

While the parameters are passed differently for GET vs. POST methods, the Java programmer doesn't usually need to worry much about this difference. Whether a GET or a POST, the parameters are available via the same mechanisms for the Java programmer. (The primary difference is the browser behavior, how it acts when the user clicks refresh or the back button. Also, for very large data, POST is preferred. Web servers may have a restriction on how big a GET request can get.)

The `ServletRequest` interface has a method `getParameterValue` to get the value of any parameter, whether it came in via a GET or a POST. It also has a method `getParameterNames` to get the list of all parameters.

Checkboxes with same name but multiple values in a form, can return more than one values with the same parameter name. There is a method `getParameterValues` available, that retrieves and returns all these values in a string array. If the query string is of the form `x=a&x=b&x=c` then this method, when asked for the values of parameter "x", would return an array containing the strings "a", "b" and "c".

Getting cookies and sessions from the Request

The request interface can also be used to get any cookies from the incoming request.

The "session" feature, as implemented in MVOWS, is also available in Java server side, and the method `getSession` can be used to retrieve the session associated with the current request. This method takes an argument, which specifies whether or not to create the session if it doesn't already exist.

The request input stream

The data in a POST is often parameters, but sometimes it may not be. If you want, you can handle the POST data yourself instead of asking Java to give it to you in the form of parameters. This is just like query strings. Java will parse the query string and give it to you as parameters, but if you want you can retrieve the entire query string yourself.

To get at the POST data, you access the "request input stream". This gives you a stream you can read, and the data you read from this string is the posted data.

In the above POST data example, if you obtain a stream and read it, the stream will return

```
x=a&y=b
```

and then will return end of file.

File upload handling

In HTML forms, you can use the INPUT TYPE of FILE, to let the users upload files. If the form has an INPUT tag of type FILE, the METHOD cannot be GET, it has to be POST, and also the FORM tag needs to include `enctype="multipart/form-data"`, e.g.

```
<HTML>
<BODY>
<FORM ACTION="/FormProcessor"
     METHOD="POST"
     ENCTYPE="multipart/form-data">
Select form to upload
    <INPUT TYPE="FILE" NAME="MyFile">
<P>
<INPUT TYPE="SUBMIT">
</FORM>
```

The form above will let the user select a file to upload, and when the user clicks submit, the file contents will be sent to `/FormProcessor` for processing.

The Java code at `/FormProcessor` needs to be able to retrieve the data of the file.

This is done using the `getPart` method on the request. Calling `getPart("MyFile")` will return a `Part` object, and calling `getInputStream` on that `Part` object will return a stream. This stream can be used to retrieve the file data.

The `Part` object can also be used to get the name of the file that the user selected.

Underneath the hood, the HTTP protocol has a specification for sending a file within a POST. This involves marking out the start and end of file data with special marker strings. The marker strings are chosen by the sending "user agent" (i.e. the browser in most cases), and must be different from the file bytes. The marker strings may look like

```
------WebKitFormBoundaryeAbcDD3xkFFrn31X
```

and are repeated throughout the POST data - to separate out the file bytes, as well as to separate out the file name, and the name of the field used.

The POST data may look, in part, like

```
 << misc headers >>
------WebKitFormBoundaryeAbcDD3xkFFrn31X
 << file bytes >>
------WebKitFormBoundaryeAbcDD3xkFFrn31X
 << file name >>
------WebKitFormBoundaryeAbcDD3xkFFrn31X
 << field name, e.g. "MyFile" above >>
------WebKitFormBoundaryeAbcDD3xkFFrn31X
```

Each part, e.g. <<file bytes>> contains some headers describing what it is, followed by an empty line, followed by the actual data.

The Java server does the string processing involved in extracting the file bytes from all the POST data, by looking for the markers and extracting anything in between.

Internationalization

The request interfaces can be used to find out the "locale" of the visitor, as well as the character-encoding used in the request. This can be used to properly internationalize the response you send.

The Response Interface

As we learned in MVOWS, the server's response to a request may look something like this

```
HTTP/1.0 200 OK
Content-Type: text/html

<HTML>
<BODY>
Hello from My Very Own Web Server
</BODY>
</HTML>
```

where we first have some header lines followed by an empty line, and then the content of the response.

Similar to the handling of the request, the response is encapsulated by two Java interfaces, the Java interface `ServletResponse` and its sub-interface `HttpServletResponse`.

In MVOWS, we directly wrote to a `PrintWriter` object to send back the response data.

The `ServletResponse` has a method `getWriter` which returns the same type of a `PrintWriter` object, which is used to send back the response data.

In addition, `ServletResponse` also has a method `getOutputStream` which returns a type of `OutputStream` object - and you can use this to write out the response data, instead of a `PrintWriter`.

You can use either an output stream or a PrintWriter. A `PrintWriter` is frequently used to send text data. An output stream can be used to send raw binary data.

The response interfaces also provide methods (similar to MVOWS) to set content-type, change the status from the default of OK to an error code, to set headers in the response, to add cookies, to send a "redirect" response and so on.

The response buffer

As we noticed in MVOWS, the status and headers need to be sent out before the data.

So what happens if the Java programmer writing the servlet or the JSP, writes out some data and THEN decides to change the content-type or the status?

In MVOWS, this was not a problem, because the data was buffered up, and only sent out when everything is done.

In typical Java application servers, this approach is not used, because it could be inefficient with large amounts of data.

The `PrintWriter` or output stream that is returned by `ServletResponse.getWriter` or `ServletResponse.getOutputStream` actually writes data out to the socket. But you can still change the headers, even after you have written some data. This is tricky, but the reason it works is that the underlying output stream is buffered. It saves everything you write to it in a buffer, and only sends the data out on the socket when the buffer is full. For example, if the application server is using a buffer size of 4000 bytes, nothing will go out to the socket until you have written 4000 bytes out (or when you are finished, if that happens in less than 4000 bytes.)

Therefore the application server can still let you set or change headers such as content-type, as long as the buffer has not filled and nothing has been written out. Once the buffer fills for the first time and gets written out to the socket, the server has had to send out status and headers before that first write to the socket - so you can no longer change headers or status etc. This is known as the response being "committed".

This works well for many cases, because usually you know the content type or status or other headers by the time you need to write out 4000 (or other large buffer sizes) bytes, even if you may not know it at the very beginning of your code.

But in certain cases, if the first buffer has already been written, and that means the server has already sent out status and headers, if you try to change status or headers after that, it will silently fail.

You can also manually commit the response, by calling the `flush` method on the output stream or the `PrintWriter` object.

Internationalization

The response interfaces also let you specify the character-set and encoding you want to use, in case you need to deal with these issues. You cannot change the character encoding once you call `getWriter` even if the response is not committed yet - because the `getWriter` has already committed that part by picking a character encoding for you.

You can also set a locale using `setLocale` which will set the character-set if not already set, and if `getWriter` has not already been called.

The character-set can also be specified as a part of content-type, and this is similar to setting it explicitly. You cannot do this once the `getWriter` has already been called.

Session

A session object can be obtained from the request interfaces, and is of class `HttpSession`.

It is similar to the MVOWS sessions - but has more methods available.

The basic store-an-object and retrieve-an-object operations are available via the `getAttribute` and `setAttribute` methods.

In addition, there are methods to invalidate a session, get names of all attributes stored in the session, get the time when the session was created, get or set the inactive interval (the time when the session, if not used for that long, will get deleted automatically) etc.

Session Serialization

It can be important to store only fully serializable data in the session.

Sometimes, in high volume websites, more than one web-server hosts (machines) are used to service the requests. In such cases, it is possible that the first page the client sees, it serviced by one web-server machine, but another page from the same client is serviced by another web-server machine. In such cases, the second web-server machine needs to be able to access the same session data as the first one.

This requires that the session contents be actually copied from one web-server host to another. The Java application server will do this copying of the data behind the scenes for you, but it can only do this if the data is serializable. It writes out the data as bytes on the first machine, sends the bytes to the second machine, and de-serializes the data back into a session on the second machine.

Cookies

There is also an interface available to deal with cookies. Sessions use a reserved cookie named "jsesssionid" to avoid conflict with any cookies the programmer may want to use.

While we handled cookies in MVOWS, there are some more details available in cookies, which are made available via the `Cookie` class.

Cookies can be retrieved from the `HttpServletRequest` and set via `HttpServletResponse`.

The Request Dispatcher Interface

We have already seen what `sendRedirect` does. It tells the browser to go to some other particular URL.

What if that URL happened to be on the same server? It is kind of wasteful to tell the browser to go to /continueShopping or /checkout on the same server (as might happen if you are processing the results of a form that allows a shopper to continue shopping or to checkout.) This involves the server sending a response to the browser, the browser reading that response and sending another request back to the server. Some unnecessary back-and-forth traffic.

Can't we handle the `sendRedirect` internally on the server, without involving the browser at all?

We didn't in MVOWS, but in Java servers we can. In Java servers, this task of internal forwarding has been abstracted in a `RequestDispatcher` interface. In addition to forwarding to another resource, it can also be used to include the contents of another resource without forwarding.

This interface can be obtained from the request. It only has two methods, `include` which is used to include the contents of another URL, and `forward` which is used to forward the browser to another URL.

To include or to forward, the server basically acts like a browser (i.e. it has some code that is client code.) Acting as a browser, it then retrieves the data from the URL being included or forwarded. The retrieved data is either included in the current page in case of include, or replaces the contents of the current page in case of forward.

Of course, there are some shortcuts possible since the URL is within the server itself, but the above describes the basic concept of the include and forward operations.

Configuration Data

Frequently there are details that do not belong in actual Java code, but in some sort of a configuration file.

For example, some programs may send out an email message to a certain individual. E.g. upon the successful completion of an order being placed, or upon an error occurring in a web page.

It may not seem like the best idea to hardcode the email address in Java. It could be placed in a database, but that may be considered an overkill, especially if it is a single email address. In that case, a configuration file could be a good choice.

Similarly, the "connection string" for a database is a good candidate for being placed in a configuration file.

For such configuration data, Java servers provide a `ServletContext` class. It can be used to retrieve initialization data configured in the configuration file.

In addition to the application-wide `ServletContext`, there is a `ServletConfig` class which is specific to configuring particular servlets.

The configuration data is typically stored in XML files.

Listener Interfaces

Java server side programming has some interfaces available that let the programmer "listen" to and respond to various events. These are similar to typical Java "listener" interfaces, for example in AWT and Swing.

The programmer must implement a class that implements the interface. Then the programmer registers an object of that class, and receives callbacks when the event of interest happens.

For example, suppose you are interested in taking some action whenever the server starts. We already have seen how this is implemented in MVOWS. In Java server side programming, it is very similar except that the class names have to be listed in a configuration file. Also, instead of our very simple interface, the interfaces tend to be more complicated and each of them can handle multiple listening-related tasks.

In Java, to take actions whenever the server starts, the listener class is called `ServletContextListener` which also has a method for when the server shuts down. If you are not interested in the server shutting down event, you still must implement this, but your implementation can be empty.

Once you have an implementation of `ServletContextListener`, you simply register it in the config file.

There are various other listener interfaces which can come in handy, for instance whenever a new request is received/destroyed, whenever a session is created/destroyed, etc.

Getting Started with a Java Application Server

For further work, you will need a standard Java application servers. Tomcat is very popular, but there are others available. For our purposes, any good Java server will work. You need to know where to place HTML (most Java servers can serve HTML files directly) and JSP files, where is the configuration file in which servlets will be configured, and how to add servlet classes to the server's classpath.

Configuration by itself is an issue of some depth. But for getting started, all that we need to know about is the `web.xml` file.

The web.xml file

This is the main configuration file in Java servers. In MVOWS, we configured MyWeblets by adding a line of Java code. Obviously, this is not practical for full-fledged Java servers. They use `configuration files` for this purpose. The configuration file is named `web.xml`. This is an XML file, and all items are configured using XML. If you don't know XML, don't worry, it is very self-explanatory.

MyWeblets are replaced by `servlets` in Java servers, and they are configured by adding some XML to the configuration file. Most servers monitor the `web.xml` file for changes, and re-read it whenever it changes. In some servers, it may be necessary to take some action or reboot the server in order to get it to re-read the `web.xml` file.

In addition to servlets, the configuration file contains many other items, such as the location of the directory (folder) containing the HTML files.

Running and testing the server

If you don't have a Java server such as Tomcat already installed, download and install it. Then locate the directory where you can place web files for the server, and make sure you are able to view HTML files from the server, in a browser.

You may also want to set the port to 80, if it's not already. If the port is 80, you can access it using

```
http://localhost/
```

otherwise you can access it using the port number. E.g. if the port number is 8080, it will need to be accessed as

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

For Tomcat, this is controlled from the `server.xml` file in the `conf` folder. The Tomcat default is 8080.

Also for Tomcat, there is a default location webapps\ROOT where you can place HTML files and see them in the browser via the URL

```
http://localhost/MyFile.html
```

or

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

Make sure you specify the JAVA_HOME for your server as the JDK location, not the JRE. The JDK includes the "javac" compiler, which is required for processing JSPs by many Java servers.

Also, before proceeding, spend a bit of time reviewing the file structure for your Java server. You should see some `WEB-INF` folders. These are configuration folders, and will typically contain a `web.xml` file. See what they look like, and become familiar with their XML structure.

Chapter 6
Writing Servlets

If you have been following the book, you have not only already written your very own "MyWeblets", you have in fact provided the container programming for them. So you should have a very good understanding already of what's going on in "servlets" in Java.

Java "servlets" are very similar to the "MyWeblets", except that they have quite a bit more detail. In terms of architecture, they are identical to "MyWeblets". But they are written for professional usage around the world, therefore they have to take care of all possible nooks and crannies.

Also, they need to be configured via an XML configuration file - it's not reasonable to expose the container code to all programmers and have then add in lines within that code! That approach would require a container rebuild and restart for any new servlets, which is unrealistic in a production environment.

A simple servlet

In Java servers, the primary class of interest for writing a servlet is `javax.servlet.http.HttpServlet`. There is also an interface `javax.servlet.Servlet`, which is implemented by `javax.servlet.http.HttpServlet`.

The `javax.servlet.http.HttpServlet` class is similar to our `mvows.MyWeblet` class. The servlet writer extends this class and implements various methods, just like a MyWeblet writer.

The primary method of interest, analogous to our `doRequest` method, is

```
protected void service(HttpServletRequest request,
                      HttpServletResponse response)
     throws ServletException, IOException;
```

But Java application servers handle additional methods besides the GET method that we handled. Remember the

```
GET / HTTP/1.0
```

header line? This line was using the method GET for retrieving the resource.

The HTTP protocol also specifies POST, PUT, HEAD, TRACE and DELETE methods in addition to the GET method. A brief description is given below.

GET and POST are the most widely used. GET is used for almost everything, POST is used in FORMs where the FORM author has set the method to be POST. We have already discussed details of the POST method. It is very similar to GET, but the data is included in the body of the message, instead of in a query string.

The methods other that GET and POST are relatively little used.

HEAD is very similar to GET, but the client is asking the server to only send the header (until the first empty line), and no actual data. This can be used to ensure the headers being returned are correct or to use header information, without incurring the full data transmission.

PUT is meant as a method for updating files on a server. DELETE is meant to delete files on the server. TRACE is used like a "ping" command, to trace the path of data.

The servlet class includes a base `service` method which can be used to handle all of these - but in addition, it provides specific methods that you can write just to handle the GET or POST or any of the other methods.

So if you want to handle GET but not POST requests, you do not need to override the `service` method. There is a

```
protected  void doGet(HttpServletRequest req,
                      HttpServletResponse resp)
     throws ServletException, IOException;
```

method. You can override this method instead. The default implementation of `service` will route all GET requests to this `doGet` method.

Similarly, there are `doPost`, `doHead`, `doPut`, `doDelete`, and `doTrace` methods available.

Once we have decided to override one or more of these methods (typically `doGet`), the programming is very similar to the MyWeblets. But in servlets, we do not get a nicely configured

```
PrintWriter out
```

directly, there is one extra step involved. You have to ask for such a `PrintWriter` object, from the `resp` object, as

```
PrintWriter out = resp.getWriter();
```

Once you have taken this extra step, you do have your `PrintWriter out` available, just as in MyWeblets.

The advantage of this extra step is that a `PrintWriter` does not have to be used. You can also ask for an `OutputStream` object instead, or in the situations where you don't want to do any output, you don't have to ask for either one.

In any case, we now know enough to write a simple `Hello, World` servlet.

```
package myservlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class MyHelloWorld extends HttpServlet
{
 protected  void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
     throws ServletException, IOException
 {
   PrintWriter out = resp.getWriter();
   out.println( "<HTML>");
   out.println( "<BODY>");
   out.println( "<H2>Hello, World</H2>" );
   out.println("Hello from My First Servlet");
   out.println( "</BODY>");
   out.println( "</HTML>");

 }
}
```

Listing 6.1

This is almost identical to your very first MyWeblet. So let us compile and configure and run it! Note that the javax.servlet.http and javax.servlet packages are not included in the standard Java distribution. Instead, the Java server is supposed to include them. So you need to locate where these are, and add them to the compilation classpath. E.g. for Tomcat, the file "servlet-api.jar" in the "lib" folder contains these packages, and needs to be given to the Java compiler.

If you have an IDE like Eclipse available, which has its own servlet container for testing purposes, you can immediately run this servlet.

To run it inside a servlet container like Tomcat, some configuration is required.

Configuring a servlet

There are many places a servlet can be configured in a Java server. The "root" of the "default" application is usually a good place to start. In Tomcat, this is in "webapps\ROOT", and we will be editing the `WEB-INF\web.xml` folder there, and placing our compiled class files there.

Configuring a servlet in a `web.xml` file involves two XML tags, `servlet` and `servlet-mapping`. They are connected by the "servlet name". The "servlet name" is just a string that the servlet programmer or configurator makes up, and which should be descriptive. The `servlet` tag connects that name to a Java class. The `servlet-mapping` connects that name to a URL. Together, they tell the server that when some particular URL is invoked, the server should instantiate a particular Java class and use it to service that URL.

To configure the servlet above, you need to add these two XML tags into the `web.xml` file in the WEB-INF folder.

```
<servlet>
 <servlet-name>
    MyHelloWorldServlet
 </servlet-name>
 <servlet-class>
    myservlet.MyHelloWorld
 </servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>
    MyHelloWorldServlet
 </servlet-name>
 <url-pattern>
   /MyHelloWorldServlet
 </url-pattern>
</servlet-mapping>
```

Listing 6.2

Just put these lines in a syntactically correct location within the XML. Right before the </web-app> tag is good.

In addition, you must make sure that the server's classpath is configured to access the `myservlet.MyHelloWorld` class, or else place that class file in the WEB-INF\classes folder (where the server knows to look for class files.) For Tomcat, since initially we are working in the "ROOT" application, we need to place the classes in the WEB-INF\classes folder located in webapps\ROOT\WEB-INF. You may have to create the "classes" folder under WEB-INF, if it is a new installation. The class is in a package "myservlet", therefore under "classes", a folder "myservlet" also needs to be created, and the HelloWorldServlet.class file needs to be placed inside that "myservlet" folder. You can have "javac" take care of all this by using a command such as

`javac -d "%tomcat_root%\webapps\ROOT\WEB-INF\classes" -classpath .;"%tomcat_root%\lib\servlet-api.jar" MyHelloWorld.java` where tomcat_root is pointing to the folder where Tomcat is installed.

That's it, once you have made the configuration change and put the class in the right place (and restart the server as well), your servlet should be available at

```
http://localhost/MyHelloWorldServlet
```

or something like

```
http://localhost:8080/MyHelloWorldServlet
```

if your server is not using port 80.

If you make any changes to your servlet, you can simply stop and restart your Java server to reload it. Later, we will see how this is done without having to stop/restart the server.

Solutions folder: 6.1

Changing the content type, redirecting and error returns

In MyWeblets, you saw how to change the content type of the output, how to redirect the browser to another URL, and how to return errors.

It is all very similar in servlets. In servlets, these are all handled via the "response" object. To change the response type, simply add

```
   resp.setContentType("text/plain");
```

at the top of the servlet code, and observe how the output changes.

Then try adding the line

```
   resp.sendRedirect( "http://someurlhere/" );
   // Put some real world URL here
```

to your servlet, and comment out everything else - this should send the browser to the URL you put in there.

Similarly, the line

```
   resp.sendError( 404, "Not Found" );
```

will send an error code and string back to the browser.

Processing Input

It is now time to recall some of the form processing we did in chapter 3.

Solutions folder: 3.3

In the example in that chapter, we processed a "firstname" and a "lastname" parameters, and printed some other information.

We can reuse the HTML files from that code section without any change, but we have to rewrite the Java code slightly.

First of all, we do not get a `parameters` argument. Instead, we have to ask the request object for the parameters. Then we have to do something similar for the URL and the query string. Otherwise, the code stays identical.

The theme to observe here is that the servlet only gets a "request" and a "response" parameters. Everything else that is needed, is packed into these two objects.

```
package myservlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class NameProcessor extends HttpServlet
{
 protected  void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
     throws ServletException, IOException
 {
   PrintWriter out = resp.getWriter();
   out.println( "<HTML>");
   out.println( "<BODY>");

   // Write out the first name and the last name

   out.println( "Hello " +
                req.getParameter("firstname") +
                " " +
                req.getParameter("lastname"));

   // Add some other dynamic info
   out.println( "<P> The time is now ");
   out.println( new java.util.Date());
   out.println( "<P>The Resource Name is " +
              req.getRequestURL());

   out.println( "<P>The Query String = " +
              req.getQueryString());

   out.println( "</BODY>");
   out.println( "</HTML>");
 }
}
```

Listing 6.3

We need to configure this servlet at the servlet-url `/ProcessName` because that's what the HTML files are referring to. Once you have this servlet configured, it will work in the same manner that the MVOWS version worked in. Visit `EnterName.html` (the `EnterName.html` file needs to be place directly in "ROOT" folder), put in a first and last name and submit the form, and you should see the output generated by this servlet.

Solutions folder: 6.2

The Java server is doing all the same text processing that you did in MVOWS. The details are slightly different, because in MVOWS we did not want to complicate the learning process by spending time building up complex request and response classes. But the processing is very similar to what you did.

In a servlet, you can do debugging steps such as dumping the parameter list to `System.out`, just the same as you could in MVOWS. The syntax is slightly different, of course!

```
 java.util.Enumeration params =
          req.getParameterNames();
 while ( params.hasMoreElements())
 {
   String paramName =
            (String) params.nextElement();
   System.out.println("Parameter Name = " +
         paramName +
         ", Value = " +
         req.getParameter( paramName ));
}
```

Listing 6.4

This will usually go to the "Console" (remember to stop/restart the Java server if you have made this addition after seeing the previous output.) In addition Java servers may also write this info to a "log file" so you can review it in a text editor.

Cookies

Cookies were useful in MVOWS because they are used to implement sessions. In general, cookies are not very useful once we have sessions. But they still have their uses. In particular, the "persistent" cookies can be used to store user data, e.g. their preferences, in the user's browser.

In servlets, cookies are set via the `response` object, and are retrieved via the `request` object, as you might expect. When you set the cookie via the `response` object, the next time the same visitor visits, the `request` object will contain that cookie for you.

Here is the servlet version of our MyWeblet to try out cookies.

```
static int cookieNumber = 1;
static Object cookieLock = new Object();

protected  void doGet(HttpServletRequest req,
                      HttpServletResponse resp)
    throws ServletException, IOException
{
  PrintWriter out = resp.getWriter();
  out.println( "<HTML>");
  out.println( "<BODY>");

  // Check if the cookie exists
  String cookie =
             getRequestCookie( "TestCookie" );

  // Set cookie if not found, and print out
  // cookie info to the Java console
  // for verification

  if ( cookie == null )
  {
      // Give a unique cookie to
      // each browser/visitor

      synchronized (cookieLock)
      {
          cookie = "Cookie_" + cookieNumber++ ;
      }
      setResponseCookie( "TestCookie", cookie );

      out.println(
         "Visiting browser does not have cookie");

      out.println(
        "<P>Setting cookie to: " + cookie );

  } else {
      out.println("Browser has cookie: " +
                   cookie );
  }

  // Add some other dynamic info
  out.println( "<P> The time is now ");
  out.println( new java.util.Date());
  out.println( "</BODY>");
  out.println( "</HTML>");
}
```

Listing 6.5

You can visit this using different browsers (e.g. IE, Firefox, Chrome etc) and they should all receive their unique cookies.

Again, using `synchronized` for getting unique cookie numbers is important - multiple instances of the servlet could be running into separate threads. Anything that is common to multiple instances of the servlet, should have synchronized access.

Because the `cookieNumber` is declared `static` and shared by all threads, access to it needs to be synchronized.

Important: Note that what we did above is NOT the correct way to set cookies. The `cookieNumber` will get reset to 1 every time the server is restarted, and therefore the cookie numbers will start recycling. Very bad! Cookie numbers must be unique. We could have used the timestamp and appended the cookie number to it, to get a more unique cookie string. For our purposes, it is better if cookie strings are easy to read so we can see what is going on. But in actual applications, using a timestamp with a cookie number appended to it is a minimum safeguard. At the other extreme, we could be retrieving and updating cookie numbers from a database.

Solutions folder: 6.3

Also note that `Cookie` is a full-fledged class, instead of just a name and a value as we used in MyWeblets.

This class provides more details that can be set and/or retrieved.

The `path` of the cookie is important. For the example, we did not specify the path, so it would default to `http://localhost/`, the path retrieved from the URL `http://localhost/CookieTest`. This would make the cookie visible everywhere in `http://localhost/` hierarchy.

But if the original URL were `http://localhost/somepath/CookieTest`, the cookie would not be visible from, say, `http://localhost/CookieUse`, because it does not contain the `somepath` part of the path!

To override the default behavior, the `Cookie` class lets you get and set the path to be used.

You can also set or get the maximum age of the cookie. The default is -1, which means the cookie will be lost when the browser is closed. This is known as a "temporary cookie". You can also set the maximum age to any number of seconds, and the cookie will be kept for that many seconds, even beyond browser shutdown. This is known as a "persistent cookie".

For example, to set a cookie and ask the browser to keep it for 100 days, you would set the maximum age as follows.

```
cookie.setMaxAge( 100 * 24 * 60 * 60 );
  // Number of seconds in 100 days =
  //       100 days * 24 hrs *
  //       60 minutes * 60 seconds
```

The server will then send an expiration date to the browser along with the cookie. The browser will attempt to keep this cookie available in the browser for 100 days after you set it. You can also update it at any time, so the 100 day countdown could start over.

There are also methods available to set and retrieve a comment to go with your cookie, in addition to the value of the cookie.

Sessions

With MVOWS, we have seen how sessions work under the hood. So now we need to re-run our test program using a regular Java server, and then get an idea of what extra bells and whistles are provided in sessions in Java servers.

Here is our session test algorithm in pseudo-code again.

```
 Main entry point is /SessionTest

 /SessionTest checks if the first and last
     name are in session.

   If not
     Does sendRedirect to /EnterName.html
   Else
     Prints welcome message using first
     and last name
   End

 /EnterName.html
   Is an HTML FORM with ACTION set to /ProcessName,
   and INPUT fields firstname and lastname

 /ProcessName
   Retrieves first and last name from
   request parameters.

   If first and last name not found
     Does sendRedirect to /EnterName.html
   Else
     Saves first and last name in session
     Does sendRedirect to /SessionTest
   End
```

We can reuse our EnterName.html, and then we have to implement a servlet at /SessionTest and modify the /ProcessName servlet.

The session is now retrieved by `req.getSession( true )`, where the `true` argument is specifying that a session should be created if it isn't already existing. Without this argument, if a session didn't already exist, we would get a `null` back.

Here is the new version of the /ProcessName servlet.

```
package myservlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class NameProcessor extends HttpServlet
{
 protected  void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
     throws ServletException, IOException
 {
   String firstname =
              req.getParameter("firstname");

   String lastname =
              req.getParameter("lastname");

   if ( firstname == null || lastname == null )
   {
       resp.sendRedirect( "/EnterName.html" );
       return;
   }

   HttpSession session = req.getSession( true );

   session.setAttribute( "firstname", firstname );
   session.setAttribute( "lastname",  lastname );
```

```
    resp.sendRedirect( "/SessionTest" );
  }
}
```

Listing 6.6

If you compare it to the MVOWS version, you will notice that changes are syntactical only: `parameters.get` gets replaced by `req.getParameter`, `sendRedirect` gets replaced by `resp.sendRedirect`, `MyWebletSession` gets replaced by `HttpSession`, and `getSession()` gets replaced by `req.getSession( true )` but the code structure is essentially the same - in MVOWS we provided these same features using a different syntax.

Similarly, we need to add a servlet for /SessionTest.

```
package myservlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class SessionTest extends HttpServlet
{
 protected  void doGet(HttpServletRequest req,
                      HttpServletResponse resp)
     throws ServletException, IOException
 {
   HttpSession session = req.getSession( true );

   String firstname =
       (String) session.getAttribute("firstname");
   String lastname =
         (String) session.getAttribute("lastname");

   if ( firstname == null || lastname == null )
   {
     // Name is not in session, get via form

     resp.sendRedirect( "/EnterName.html" );
     return;
   }

   PrintWriter out = resp.getWriter();
   out.println( "<HTML>");
   out.println( "<BODY>");

   // Write out the first name and the last name

   out.println( "Hello " + firstname
                  + " " + lastname );

   out.println( "<P> The time is now ");
   out.println( new java.util.Date());
   out.println( "</BODY>");
   out.println( "</HTML>");
 }
}
```

Listing 6.7

Once these have been deployed correctly, visiting http://localhost:8080/SessionTest or http://localhost/SessionTest as the case may be - should result in the following behavior, just like in MVOWS.

If the server doesn't "know" you (because the session doesn't have your name in it), it will redirect you to EnterName.html. EnterName.html will forward you to /ProcessName, which will save the name in the session and then send you back to /SessionTest. This time the server "knows" you, so /SessionTest will say hello.

After this, if you keep coming back to /SessionTest, it will always "know" you, and will not ask for your name again, and will keep saying hello.

But if you do not visit for 20 minutes (the typical default configuration for session expiry), the server will delete the session, and therefore forget all about you. If you visit again, it will ask for your name again!

That covers the basic session usage, including session deletion like we implemented in MVOWS.

You can also programmatically delete the session by calling the method `HttpSession.invalidate()`. There are methods `HttpSession.getMaxInactiveInterval()` and `HttpSession.setMaxInactiveInterval()`,to get or set the time for session expiry.

There is a method `HttpSession.isNew()` to check if a session was just created in the current URL.

There are also methods to find out the session creation time, and the time it was last used by a visitor.

Servlet Debugging

Servlet debugging tends to be specific to the Java server and IDE, therefore we won't spend much time on it. With many servers (including Tomcat), it is possible to have an IDE like Eclipse actually connect to the running instance of the server.

Once you have figured out how to configure the IDE to connect to your server, the debugging proceeds normally with breakpoints etc.

Since the details are server specific, we will not be going deeply into that. Debugging is mentioned here merely to point you in this direction - to know that servlets are usually fully subject to the debugging tools.

Include and Forward

To include the contents of another URL from a servlet, we use

```
RequestDispatcher dispatcher =
  request.getRequestDispatcher( "/another_url" );

dispatcher.include( request, response );
```

To forward to another URL from a servlet, we use

```
RequestDispatcher dispatcher =
  request.getRequestDispatcher( "/another_url" );

dispatcher.forward( request, response );
```

This is different from `sendRedirect` as discussed in the previous section. Whereas `sendRedirect` sends an instruction back to the browser, these methods are processed within the Java server itself, without involving the browser. This can be done easily, because the code or the file for `/another_url` is available to the Java server, so the Java server can process it directly without involving the browser.

However, for redirecting the browser to an external URL, `sendRedirect` should still be used. A major difference from the user's perspective is that when you use `sendRedirect`, the user will see the new URL in the browser. If you use the include and forward methods of the `RequestDispatcher` interface, the browser is not involved, so the user will see the original URL in the browser.

Listener Interfaces

For various tasks, Java servers also provide various `listener` interfaces, similar to the listener interface we implement in MVOWS to track server startup.

In Java, there is an interface known as `ServletContextListener` that can be used to known when the `context` is being initialized or destroyed. We will discuss `context` in more detail later, but it is sufficient to know for now that if you want some initialization routine to be called before any of your servlets above run, this is a good listener to implement.

This interface provides two methods, `contextInitialized` and `contextDestroyed`.

Here is an implementation to print something out on `contextInitialized`.

```
package myservlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class InitializationTest
     implements ServletContextListener
{
   public void contextInitialized(
               ServletContextEvent sce)
   {
       System.out.println(
           "*** INITIALIZATION OCCURRED ***");
         // The asterisks are to make sure it
         // stands out among all console messages!
   }

   public void contextDestroyed(
               ServletContextEvent sce)
   {
   }
}
```

Listing 6.8

For configuring it, an XML tag named `listener` needs to be added to the `web.xml`, and it needs to contain a tag `listener-class` containing the fully qualified name of the class, i.e. `myservlet.InitializationTest` in this case.

If the class is compiled and placed correctly, and the `web.xml` is configured correctly, you should see the output

```
*** INITIALIZATION OCCURRED ***
```

on the console, during the Java server initialization.

There are other similar listener interfaces, and they all work the same way. You implement the interface of interest and configure it, and the server will call your implementation(s) at the appropriate time.

That covers our review of the servlet (and related listener interfaces) technology, and we will now proceed to the much more powerful and effective (but harder to work with) JSP technology, also known as Java Server Pages.

Chapter 7
Java Server Pages

JSP (Java Server Pages) is a very powerful technology, though it can be somewhat difficult to work with.

The original intent of the JSP technology was to provide a presentation layer. It was meant to be basically HTML on steroids, with Java embedded inside it. However, it has a set of powerful features that make it useful for non-presentation purposes as well. On the negative side, Java IDEs do not work too well with JSP. It was mentioned in the previous chapter that servlets are usually fully debuggable in many environments. This is not true of JSPs. Never mind debugging, even syntax highlighting and syntax checks within a JSP tend to be too much for IDEs to handle correctly. As a result, JSP is a difficult technology to program in.

While working with our own version of JSP, which we called MSP, we have already seen the core technologies of how a server implements JSP. The server turns the page into a servlet, exactly as we turned our MSP's into MyWeblets. Then the server automatically compiles and deploys the newly generated servlet, just as we did for our MSP technology.

The automatic deployment is no doubt a boon. As you may have realized if you made any mistakes during the servlet configuration of the last chapter, deployment can be a hassle. The automatic deployment of JSPs takes that hassle out of the equation. Not only that, when you change a JSP, the servers immediately re-deploy it for you.

Those are good things, balanced by the difficulty of programming Java in JSPs. A good balance can be to start things off in JSPs, but have the significant part of the code in regular Java classes, which classes are then called from the JSPs. That way, most of the Java code is subject to regular syntax checks, syntax highlighting and debugging etc. and we can still take advantage of the auto-deployment and other features (which we will meet in this chapter) of JSP.

In addition to the certain amount of difficulty of programming JSPs, the biggest problem is that many Java programmers do not understand what is going on behind the scenes in a JSP, therefore they tend to avoid it as much as possible, thinking it is something rather magical. To readers of this book, that certainly should not be a problem, and therefore they should be able to make good objective use of JSPs, with reference to the actual pros and cons of using JSP in a particular case, instead of being afraid of its mysticism!

So let us get started with using JSP in a full-fledged Java server environment.

Hello, world

An HTML file can be made into a JSP just by renaming its extension to .jsp. This is a good exercise to try out. Create the file `HelloWorld.jsp` as

```
<HTML>
<BODY>
Hello, World
</BODY>
</HTML>
```

and place it in the root web folder of your Java server (where you would place any other HTML file), and load it from the browser by visiting `http://localhost/HelloWorld.jsp` (or `http://localhost:8080/HelloWorld.jsp`, as your server is configured.)

There is no configuration required, the server will parse, compile, configure and load the JSP to show you the `Hello, World` all by itself! If you reload it, there will no parse, compile, and configure cycle. But if you change the JSP, the server will rebuild it again.

This is a good time to search the web server files. Somewhere in the web-server hierarchy (in Tomcat, it's somewhere under a folder named "work") there are a .java and a .class files that have been created for your `HelloWorld.jsp` file. In some servers the .java file is deleted after compilation, but this can be controlled by server configuration flags.

It is a good idea to become familiar with your server's details on where and how the .java file is placed.

If you find the correct .java file, you should be able to find lines in there matching your JSP, something like

```
 out.write("<HTML>\r\n");
 out.write("<BODY>\r\n");
 out.write("Hello, World\r\n");
 out.write("</BODY>\r\n");
 out.write("</HTML>\r\n");
```

When you change and save the JSP and load it again (e.g. if you change the output to "Goodbye, World"), the .java file immediately gets recompiled and rebuilt, very much like we did in our MSP. The very same technology of checking the file's timestamp is being used by the Java server.

A dynamically changing JSP

In the chapter on MSP, we started off with a very simple MSP

```
<HTML>
<BODY>
<P>The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

This is also an excellent second JSP to test out.

If you run this JSP through the Java server, you should be able to see the current time every time you refresh the browser page.

If you look in the Java code behind this JSP, you should see something like:

```
     out.print( new java.util.Date());
```

in the .java file. The .class file generated by this is what gets loaded in the server, and every time that class is run, it prints out the current value of `new java.util.Date()`.

Errors

Try adding an error to the file, e.g.

```
<HTML>
<BODY>
<P>The time is now <%= new java.util.xxDate() %>
</BODY>
</HTML>
```

For such a small error, usually the server is able to provide a good error message. For more complicated errors, the server may end up referring you to the Java line number. These really make little sense to those who do not understand what is happening behind the scenes, and therefore such error messages confuse and intimidate the typical Java programmers.

Fortunately, your experience with MSP should really help you here, because you know what is going behind the scenes, therefore you will have a much better understanding of the error messages than typical Java programmers.

Declarations within JSPs

The syntax for declaring a method is the same in a JSP as in an MSP.

The MSP code

```
<%!
   // Method to compute a number cubed
   int cube( int n )
   {
       return n * n * n;
   }
%>
<HTML>
<BODY>
<P>Cube of 5 = <%= cube( 5 ) %>
</BODY>
</HTML>
```

will work just fine as a JSP, all that is required is to change its extension to .jsp, and place it in the correct place, and it can be loaded and will display the result.

Synchronization in JSP declaration

It is important to realize that anything declared in a JSP will be shared by multiple threads.

If we declare an integer using the declaration syntax

```
<%!
  int counter = 3;

  int getCount()
  {
       counter = counter + 1;
       return counter;
  }
%>
```

then the `counter` can be shared by multiple threads, and there are well known issues involving that. The count can be wrong because of the interleaving of threads.

Other kinds of code can produce a "deadlock".

A full discussion of synchronization is not in the scope of this book. This is just to highlight that a declaration block in a JSP is very much like declarations and methods in a regular Java class (in fact, it does gets turned into a regular Java class as we are well aware from MSP.) Therefore multi-threading synchronization issues fully apply, and the programmer needs to be aware of that.

JSP as target of a FORM action

A JSP can be the target of an ACTION in a FORM, and in fact this is a very frequent and common usage of JSPs.

Just like we saw when working with MSPs, a JSP can be targeted from an MSP, as well as from other JSPs, and even by itself!

The `cube.html` and `cube.msp` from our earlier FORM ACTION exercise using MSPs will work just fine as a JSP example. We need to change the name of `cube.msp` to `cube.jsp`, and to edit the `cube.html` and change the reference in it from `cube.msp` to `cube.jsp`.

In addition, there is one more change required. Recall that the JSP gets turned into a servlet, just like we used to turn our MSPs into MyWeblets. So the code

```
parameters.get( "number" )
```

needs to be changed to

```
request.getParameter( "number" )
```

in the two places it occurs.

That's it, now the example will work as a JSP.

Finally, the example where `cube.msp` targeted itself, will also work just fine with `cube.jsp` with similar changes.

Here is the modified cube.jsp:

```
<HTML>
<BODY>
<%!
   // A method to return cube of a number

   int cube( int n )
   {
       return n * n * n;
   }

%>
<%
if ( request.getParameter( "number" ) != null )
{
 try {
   int n = Integer.parseInt(
               request.getParameter( "number" ));

   out.print("The cube of " + n + " = " +
             cube( n ));

 } catch (NumberFormatException nex)
 {
   out.print("Please enter a valid number.  " +
             "You entered: " +
             request.getParameter( "number" ));
 }
 out.println("<HR>");
}
%>
<FORM METHOD=GET ACTION="cube.jsp">
Enter number: <INPUT TYPE=TEXT SIZE=4 NAME="number">
<P>
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Listing 7.1

Solutions folder: 7.1

Predefined variables in JSP

Above, we changed

```
parameters.get( "number" )
```

to

```
request.getParameter( "number" )
```

in order to get the JSP to work.

Just as `parameters` was defined by MVOWS (if you are not sure how MVOWS did this, go back and take a look) `request` is defined by the Java server.

Along with `request`, there are also `response`, `out` and `session`, with the expected behavior.

In addition, there are `application`, `config`, `pageContext` which we will cover later.

There is also `page`. This is not very useful, and is just set to `this`, but it does mean you cannot use `page` as a variable name in your JSP. Just like you cannot use any of the other pre-defined JSP variables name yourself in new declarations.

Using the "out" variable vs. mixing HTML and scriptlets

As you have seen, any normal HTML (or other) text you write in a JSP just gets turned into `out.print` or `out.println` method calls. E.g.

```
Some <B>Bold text</B>
<P>
```

may just generate something like

```
out.println("Some <B>Bold text</B>");
out.println("<P>");
```

Within a scriptlet, you can do this yourself, so the above is exactly like you writing in a JSP

```
<%
 out.println("Some <B>Bold text</B>");
 out.println("<P>");
%>
```

In terms of Java code, there is no difference in the two methods.

This gives the JSP programmer a choice of when to use the `out` variable in a scriptlet, and when to drop out of scriptlets and just write HTML directly.

This is strictly a style choice.

HTML and scriptlets can be mixed in various ways, and since you know the internal mechanics of what is happening, it should be very easy for you to do such mixing. For instance, it should be obvious what

```
<% if ( success ) { %>
     <H2>Success!</H2>
     The task succeeded
<% } else { %>
     <H2>Failure!</H2>
     The task failed
<% } %>
```

is doing. This will just turn into the code

```
if ( success ) {
   out.println("<H2>Success!</H2>" );
   out.println("The task succeeded" );
} else {
   out.println("<H2>Failure!</H2>" );
   out.println("The task failed" );
}
```

and therefore is valid JSP programming. Similarly, it is very valid to write

```
<TABLE>
<% for (Row row: myTable.getRows()) { %>
 <TR>
   <TD><%= row.getData1() %></TD>
   <TD><%= row.getData2() %></TD>
 </TR>
<% } %>
</TABLE>
```

Here, we are embedding HTML inside a Java for loop to generate a table.

Be aware that such things can become hard to read without good and consistent indentation and spacing. Just like any other Java code, but for other Java code the IDE will format the code nicely for you. For JSPs, you have to stick to good and consistent indentation practices yourself, because IDEs don't typically handle JSPs nicely.

At the same time, it should be kept in mind that direct use of `out` variable is always available in a JSP. So at any point, if it will be more clear to read an `out.println` instead of dropping to HTML, that is a very reasonable option.

As mentioned above, JSP programming is inherently more difficult than other kinds of server side programming, due to lack of IDE support and the extra trasnalation-into-Java step. Therefore all options should be used towards keeping the code clear, and easy to read. This is not just for others reading your code, but for yourself as well. When you look at a JSP you wrote three months ago, it can be quite opaque even to you if you have not used good style practices. It doesn't matter what exactly those practices are, as long as you are consistent and the code makes its structure obvious at a glance.

JSP directive

The JSPs can include various "directives". This is something we did not do in MVOWS.

A "directive" in a JSP page is introduced by using the <%@ characters (an @ sign after the JSP signature.)

A very commonly used directive is the "page" directive. It has various attributes, one of the common one's being `import`.

E.g.

```
<%@ page import="java.util.*,java.io.*" %>
```

If this directive is present, the JSP generator simply adds

```
import java.util.*;
import java.io.*;
```

at the top of the Java servlet code that it generates. Obviously, this is a useful syntactic convenience, and allows the JSP programmer to avoid having to write `java.util.Date` instead of `Date` and so on.

The page directive can have many other attributes. Some of the more useful one's are explained below.

The `contentType` attribute lets you specify the content-type of the output page. By default, you will get `text/html`, but if you want a different content-type, the page directive is a convenient alternative to having it set using the `response` variable. E.g.

```
<%@ page import="java.util.*,java.io.*"
   contentType="text/plain" %>
```

The example above also shows that attributes can be used together, you do not have to use a separate page directive for multiple attributes.

The `errorPage` attribute lets you specify a URL which will handle any unchecked exceptions that might arise in the JSP code. The error page itself can be a JSP, in which case you can specify that it is an error page by setting `isErrorPage="true"` in the target page.

The `session` attribute is useful in defining whether or not sessions are available in your JSP. By default, this is set to true. But if you add

```
<%@ page session="false" %>
```

to your JSP page, that page will not have a session available to it by default, and the predefined variable `session` will not be available.

There are also some page directive attributes to help you control buffering behavior.

In addition to the page directive, there are also an `include` directive and a `taglib` directive.

The `include` directive is used to simply physically include the contents of some file in the current JSP before the JSP is processed. E.g.

```
<%@ include file="header.jsp" %>
```

When the JSP parser gets to this point, it is supposed to read the contents of the file `header.jsp` and pretend that all that was a part of the current file. This can be useful for doing things like headers and footers. Note that this is not the same as the `RequestDispatcher.include` method. The `RequestDispatcher.include` actually loads the target resource as if a browser loaded it, and includes the result. The include directive is simpler, it is treated as if you physically wrote the entire contents of "header.jsp" file at that point within the current JSP.

As to the `taglib` directive, it involves "tag libraries".

It is not very common, and is a rather advanced topic to write "tag libraries", so we will not cover how to write tag libraries. But we will discuss how to use tag libraries written by third parties. Basically, a tag library lets third parties write a small subset of a JSP "language".

Given a third party tag library, you can use the `taglib` directive to include that third party tag library in your JSP code, e.g.

```
<%@ taglib uri="/lib/SomeTaglib.tld" prefix="stl" %>
```

specifies that there is a tag library which you have placed at the URL `/lib/SomeTaglib.tld` in your application, and which you want to address as "stl" within your JSP.

The tag library would define various tags, let us say `MyTag1` and `MyTag2`, which you would be able to access from the JSP. For example,

```
<stl:MyTag1> any enclosed stuff here </stl:/MyTag1>

<stl:MyTag2/>
```

The exact names of the tags, and details of how to use the tags and what they would do, would be totally dependent upon the tag library, and their documentation should provide such details.

JSP Tags

We have seen how external tag libraries work.

JSP does define a few tags internally as well. The "prefix" for these is "jsp". Two frequently used tags are `jsp:include` and `jsp:forward`.

The `jsp:include` and `jsp:forward` tags merely translate to `RequestDispatcher`'s `include` and `forward` method calls.

```
<jsp:include page="somepage.jsp"/>
```

gets processed into something like

```
request.getRequestDispatcher(
   "somepage.jsp").include();
```

and

```
<jsp:forward page="somepage.jsp"/>
```

gets processed into something like

```
request.getRequestDispatcher(
    "somepage.jsp").forward();
```

Therefore `jsp:include` is just a way of inserting the results of processing "somepage.jsp" at that point.

The more frequently used `jsp:forward` is a good way to internally forward the request around.

There are several other tags, the details of these are available from JSP references - but there are some tags related to "beans" which are quite useful, and which we will cover next.

JSP Beans

Perhaps the most useful feature in a JSP is the ability to use a "bean".

In this context, a "bean" is just a Java object. Any Java object (also known as POJO or Plain Old Java Object in Java programmer lingo and which sounds cooler!) will do. The only requirement is that it must have a default (no parameters) constructor.

To use a bean, you use the syntax

```
<jsp:useBean id="theBean"
      class="pkg.MyBean" scope="session">
```

This translates into a declaration in the Java code, something like (simplified version)

```
   pkg.MyBean theBean = new pkg.MyBean();
```

so in the rest of the JSP, you can use the variable `theBean`, e.g.

```
The result is <%= theBean.getResult() %>
```

assuming, of course, that the object `theBean` does provide a `getResult` method.

This by itself is not so useful yet. The "scope" makes it more interesting. A "scope" of "session" means that the declaration will not exactly be like the above. Instead, it translates into something like

```
   pkg.MyBean theBean = (pkg.MyBean)
       session.getAttribute("pkg.MyBean");

   if ( theBean == null )
   {
       theBean = new pkg.MyBean();
       session.putAttribute( "pkg.MyBean",
                             theBean );
   }
```

So as you can see above, the `jsp:useBean` tag is simplifying a very common idiom of putting something in a session, and using it from the session.

In addition to the "session", the "scope" can also be "page", "request", and "application". The scope of "application" means the bean gets initialized once, and kept in the application forever. Instead of the session, it is kept in the context attributes.

The "page" scope is only mildly useful, as you can do the same thing in scriptlets, and it doesn't help much, except for conveniently collecting data as we will see next.

The "request" scope saves the bean in the current "request" object, so as long as you are forwarding the request along to another page, the bean will stick around. But once the request is completed, the bean is lost.

The "session" scope is typically the most useful.

So what can be do with the bean, other than referencing it from code?

Well, there are two important tags, `jsp:getProperty` and `jsp:setProperty`.

```
<jsp:setProperty name="theBean"
             property="data1" value="somevalue"/>
```

```
<jsp:getProperty name="theBean" property="data1"/>
```

is equivalent to the scriptlet

```
   theBean.setData1( "somevalue" );
   out.print( theBean.getData1());
```

Not tremendously useful, it's not even shorter than the code it is replacing! But an interesting thing to note is that the Java server will figure out the method's expected type. E.g., it will treat

```
void setData1( String val )
```

different from if the method is declared as

```
void setData1( int val )
```

In the second case, where the expected parameter is an integer, the code generated will be

```
theBean.setData1( Integer.valueOf( "somevalue" ));
```

In this case it will fail, because instead of "somevalue", it is expecting a string representing an integer, e.g. "22", so

```
<jsp:setProperty name="theBean"
     property="data1" value="22"/>
```

will work correctly.

But the interesting thing is, how does the Java server know what the method declaration is, whether the parameter is declared as "String val" or "int val" within the `pkg.MyBean` class?

The Java server does this by using something called "reflection". If you have not come across it, "reflection" is a Java feature that lets you ask a class questions like "Please tell me about your methods" (as well as fields, etc.)

So when you use a bean in a JSP, the Java server actually can find out the details of the class.

Notice that property name is "data1", but the method name is "setData1".

This is a convention - the first letter of the property name is turned to upper case, and a "get" or a "set" is added before it. The property name "prop" becomes either the method name "getProp" or "setProp", depending upon the context.

When the Java server sees `property="data1"`, it asks the class whether it happens to have `setData1` or `getData1` methods, as the case may be. (This is of course a simplification, a server may choose to do processing differently. For example, a server may ask for all methods that start with `get` and `set` and save that list, and match it against property names.)

So far, even with the reflection, the bean still is not very useful. All it can do is provide a shortcut for saving a Java object in a session.

What makes a bean really useful in a JSP, is the following syntax

```
<jsp:setProperty name="theBean" property="*">
```

Here we are not specifying the property name, but simply using an asterisk instead of the property name.

When you use this syntax, you put the Java server to some real work.

It works as follows (the details may vary in actual implementations obviously, but the effect is as described here.)

It goes through all the parameters that came in the request object. For example, the JSP you are writing might be processing a form with name, address, city and state. Then the request will have parameter names like `name`, `address`, `city` and `state`, and will have values supplied by the user for each of these parameters.

The Java server will go through each of `name`, `address`, `city` and `state`, and for each case, it will use reflection to find out if the bean has a matching "set" method.

If the bean has a "setCity" method, something like the following will take place.

```
theBean.setCity( request.getParameter( "city" ));
```

but only if the request does have a "city" parameter. Missing parameters have no effect. Also, if the bean doesn't have a "setCity" method, then the city parameter is simply ignored in this case.

Why is this so useful? Suppose you are collecting a lot of data from a user. Let us say, you collect name, address, city, state on one HTML page, then on another HTML page you collect items to purchase, on another HTML page you collect credit card data and so on.

You can make one class that has "set" methods for all of these items. At each page, you process the results of the form just by writing

```
<jsp:useBean id="theBean"
          class="pkg.MyBean" scope="session">
<jsp:setProperty name="theBean" property="*">
```

You could even use the same JSP to process each of these HTML pages. In each case, the `property=*` means something different! In each case, the correct data will be absorbed.

This is very convenient for collecting data from multiple pages. The user can go back and forth over all the pages in any order, and the appropriate data will be collected at each page.

Even for a single page, collecting all the data instead of having to write out statements for each individual field, is quite helpful.

That's why `setProperty *` even makes the `page` scope beans useful. While the `page` scope beans are not typically very useful, the ability to easily collect data can save you a lot of work and make the code easier to work with, even if you do not need to share the bean via a session.

Showing Existing Data

When the user arrives at a page, it could be the second or third time, and the user may already have entered the data. E.g. perhaps the user has already entered the address, but wants to correct it and goes back to that page.

Or perhaps it's a registered user who has logged in, and whose name and address we already have, and we just want to show it.

In several such cases, it would be useful to do the reverse of what "setProperty *" does. We want to be able to take a bean, and populate the fields on a form from that.

Unfortunately, there is no "getProperty *" to match the "setProperty *" we have discussing.

This has to be done manually. It is not very difficult, you just have to turn your HTML pages into JSP pages. Then you can include the bean in the jsp, and initialize the FORM elements from the bean. E.g.

```
City: <INPUT TYPE=TEXT SIZE=20 NAME="city"
      VALUE="<%= theBean.getCity() %>">
```

The expression given to VALUE will show the city if the user has already entered it or if we already have it (e.g. from a database from which we have loaded "theBean"), and if we don't, it will show the empty string. (Make sure to default initialize to the empty string!)

You can also preload "theBean" from a database, on some entry page, if you have some of the data already sitting in a database.

This covers the major topics in JSP. Finally, we will proceed to cover various useful-to-know odds and ends about working with Java servers.

Chapter 8
Miscellaneous Topics

In this chapter we briefly cover miscellaneous useful topics that did not fit into the flow of earlier topics. Again, this is not meant to be a reference or a set of recipes. This is meant to make you aware of features and techniques that exist when doing server side programming in Java, so you can look up the details when you need them.

Applications and WAR files

We worked in the "ROOT" (or similar) folder of the Java server.

Java servers have a useful concept of an "Application". This is one of the non-ROOT folders in your web applications folder. In Tomcat, you can place such an "application" in the "webapps" folder.

For example, if under webapps you create a folder MyApp and then under that you place a MyHtml.html file, it will be available at

```
http://localhost/MyApp/MyHtml.html
```

or

```
http://localhost:8080/MyApp/MyHtml.html
```

This is standard behavior for all servers, what makes it a Java application is that you can place a folder called "WEB-INF" in that application, and then you can place its own `web.xml` file in that "WEB-INF" folder.

Not just that that, you can place a "lib" folder within that WEB-INF to contain jar files used by your application. You can also create a "classes" folder in WEB-INF to be the root of any classes.

Effectively, placing a "WEB-INF" in MyApp makes MyApp a self-contained application, which can have its own configuration, classes, servlets and JSPs, completely independent of the rest of the Java server.

It will share any classes that the Java server's classpath includes, but essentially it defines its own classpath extensions by having its own "classes" and "lib" folders, and it defines its own configuration items.

What is also useful that you can use the Java "jar" command to take all the contents of MyApp and package them up in a file, let's say called MyNewApp.war. Note the extension, it's `.war` instead of the normal `.jar` extension. But it's a good old .jar file, just with its extension renamed.

If you look in the MyNewApp.war file, you should not see a top level "MyApp" that contains stuff. Instead, you should see the WEB-INF at the very top level.

If you have created it correctly, you can drop the MyNewApp.war file in the web applications folder, and immediately you will have

```
http://localhost/MyNewApp/MyHtml.html
```

or

```
http://localhost:8080/MyNewApp/MyHtml.html
```

Note that MyNewApp no longer is corresponding to a physical folder. It will still behave like a physical folder, but all the actual content is coming from the MyNewApp.war file. The MyNewApp.war file can also contain servlets, JSPs, listeners, etc.

This is a very convenient way to package entire applications. Deploying these is also very easy, because you can drop or replace them in a running Java servers, and most Java servers can handle it without needing the server to be brought down.

Earlier, we have been always stopping and restarting the Java server to re-deploy servlets.

With WAR files, it is not necessary, you simply copy the new WAR file over the old WAR file.

Existing user sessions will continue to use the old WAR file, but new one's will start using the new WAR file.

You can hava various "Applications" running in the same server. Each "Application" comes with its own "context", which is of type `ServletContext` and is basically a way to refer to items specific to that entire application. It can be used to store/retrieve "attributes" common to the entire application. JSP beans can also have an "application" scope, which means they will be available throughout the application, are stored in the context, and will not be released until the application is unloaded (which does happen when you update an application with a new WAR file.)

Within JSPs, the predefined variable `application` refers to the context, and can be used directly. The predefined variable `pageContext`, though does not refer to the the application context, it refers to a JSP specific and page specific object of type `javax.servlet.jsp.PageContext`.

The predefined varialbe `config` refers to the servlet or JSP's configuration data. It contains a reference to the context, and the `application` variable can also be retrieved by calling `config.getServletContext()`. In addition, the `config` also provides methods to retrieve initialization parameters for a specific servlet or JSP. Such parameters can be set in the `web.xml` configuration file for the application.

Passing data between JSPs (and/or servlets)

Often times, you will want to do some processing in a JSP or servlet, and then pass control over to another JSP or servlet. You have already seen how to do this, using include/forward. But when you do this, you will often want to pass some data long. Sometimes, you will want to pass data after a user interaction, e.g. after the user clicks a submit button.

Long term data, of course, belongs in the "session". This is very convenient for most purposes.

Short term data or data that you do not wish to put in the session, can be passed using various techniques. For example, using HIDDEN form input elements. If you have not come across using HIDDEN form elements, it is worthwhile to review it.

HTML FORM tags can have inside them, HIDDEN input elements. These elements are not visible to the browser user, except when doing view-source. These elements are saved on the page, without requiring a session.

E.g.

```
<FORM ACTION=/someurl.jsp METHOD=POST>
<INPUT TYPE=HIDDEN NAME="myData"
      VALUE="<%= myData %>">

<INPUT TYPE=TEXT NAME="myTest" SIZE=20>
<INPT TYPE=SUBMIT>
</FORM>
```

Note that this FORM is itself in a JSP, and we are using `<%= myData %>` to evaluate the Java expression "myData" and place its result in the form. If "myData" evaluates to 33, the actual HTML will be

```
<FORM ACTION=/someurl.jsp METHOD=POST>
<INPUT TYPE=HIDDEN NAME="myData" VALUE="33">

<INPUT TYPE=TEXT NAME="myTest" SIZE=20>
<INPT TYPE=SUBMIT>
</FORM>
```

The JSP that processes this form (here named /someurl.jsp) will retrieve the value of "myData" as a parameter, along with the other form data. In that JSP, evaluating `request.getParameter( "myData" )` will return 33. Moreover, a bean can have a field "setMyData" and collect that parameter easily using set-prop *, along with any other parameters.

This allows us a way to pass along any data to the next JSP in the sequence, without storing it in the session.

Another way is to have a "request" scope bean, or simply setting and retrieving request "attributes". Such data is only useful for include/forward, it cannot be used for going through a form (like the HIDDEN fields shown above.) It gets lost as soon as the response page has been delivered to the browser.

You can also put data in the "query string" for HREFs or ACTIONs. E.g.

```
Click <A HREF="/someurl.jsp?myData=<%= myData %>">
this link</A> to go to next page.
```

When the user clicks on the link, the control goes to /someurl.jsp?myData=33 and /someurl.jsp can then retrieve the value of "myData" and process it.

Non-UI JSP

JSP was originally designed to display HTML, but due to it's ease of deployment and its ability to collect form-data easily, it also provides the least complex way to implement "controllers".

A controller is just some code that reads the results of a FORM submit, or a link click, processes the data, and then decides where to forward the user to.

A controller does not need to display HTML.

Just because a JSP can handle HTML, doesn't mean it needs to be restricted to that role.

For example, here is a simple "controller" that processes the results of a form submit.

```
<jsp:useBean id="theBean"
         class="pkg.MyBean" scope="session">
<jsp:setProperty name="theBean" property="*">

<%
 if ( theBean.getNext().equals( "checkout" ))
   response.sendRedirect( "/checkout.jsp" );
 else if ( theBean.getNext().equals( "continue" ))
   response.sendRedirect( "/continue.jsp" );
 else
   response.sendRedirect( "/home.jsp" );
%>
```

Note that this JSP is not displaying any HTML ever! It is only used to collect the form data in "theBean", and to send the browser to the next location. (Btw, while include/forward are efficient, sometimes they have the effect of confusing the user's forward/back buttons. For best results and if you are not pressed for efficiency, use sendRedirect in all cases where it makes sense for the user to see the new URL in their browser.)

If the processing gets very complex, you can add a method or two to the bean, and call that from the JSP.

```
<jsp:useBean id="theBean"
         class="pkg.MyBean" scope="session">
<jsp:setProperty name="theBean" property="*">

<%
 theBean.callDatabaseAndSoapMethodsAndStuff();

    // Do complex processing in pkg.MyBean class

 theBean.otherMoreComplexProcessing();

 if ( theBean.getNext().equals( "checkout" ))
   response.sendRedirect( "/checkout.jsp" );
 else if ( theBean.getNext().equals( "continue" ))
   response.sendRedirect( "/continue.jsp" );
 else
   response.sendRedirect( "/home.jsp" );
%>
```

There is no reason to put complex processing in JSP. Scriptlets are just Java code, and they can call anywhere in your Java classes!

However, JSPs do provide a very convenient starting point for processing.

XML, VXML, audio, video etc.

In almost all our discussion of JSP, we have been assuming we will be dealing with HTML.

This is not required. JSPs have a "contentType" parameter in the @page directive, and this parameter can be used to easily change the content type to other types of data, e.g. XML, VXML etc. Alternatively, the content-type can be set explicitly via the response object.

JSPs can easily be used to emit XML and VXML, just like they can be used to emit HTML. They don't care what the textual format is, they will simply process scriptlets and expressions and directives etc.

However, there is a very important caveat here. JSPs can only be used in output types where whitespace is not significant! That means XML and VXML are fine, but you cannot use JSP for audio/video etc. where the data is binary. Here is the example of the problem that arises:

```
<%@ page contentType="audio/mpeg" %>
<%
 MyAudioGenerator gen = new MyAudioGenerator();
 OutputStream ostream = response.getOutputStream();
 gen.sendAudio( ostream );
%>
```

Looks fine? Actually, it has big problems. There is an end-of-line at the end of the line containing the `@page` directive, and another at the end of the scriptlet. These end-of-lines (and any spaces outside of the scriptlet if there happen to be any) will go right into the output stream!

For binary data, they will corrupt the output.

So for binary data where inserting extra characters is going to mess up the data, do not use JSPs. Servlets are the correct technology to use there.

AJAX, REST, JSON And Other Frameworks

AJAX is a very useful modern technology, that can reduce the burden on the server and can improve the user experience. A full discussion of AJAX doesn't belong here, because it is not really a server-side technology. It is a client-side technology, but it does require some assistance from the server.

Depending upon the AJAX technology you are using, the server can provide an HTML fragment (which often gets parsed and replaces some HTML on the page with a matching "id"), or an XML document or a plain text document (which are parsed and processed on the client.)

For most of these purposes, JSPs are well suited. They can generate the server side data in the appropriate format.

There are some other server side technologies worth mentioning.

REST, JSON, SOAP etc. are very useful server side technologies. But unlike the technologies presented in this book, these server side technologies are not deep or particularly complex. They are merely ways to encapsulate data in XML or similar notations, any complexity arising solely from sheer volume. In fact, JSPs and servlets can very effectively be used to generate all of these. With the understanding built up in this book, if the need arises, generating any of these types of output is simply a matter of understanding the formatting. Fortunately, good tools are usually available to generate the appropriate formatting, therefore such need should not arise frequently.

There are many other Java server-side frameworks available, that this book doesn't touch upon. The technologies presented here provide a strong base for most server-side projects, and for understanding various frameworks. JSPs, contrary to first and superficial impressions, actually are great for writing "controllers" even if no data presentation is involved. A JSP does not **have** to output HTML or any presentation level data! JSPs with some servlets for binary data, along with all complex processing in regular Java classes, are very suitable for building applications. That having said, new technologies are always emerging, and many of these would prove actually useful in the field, and many would not. But any framework or design approach should not be chosen out of a mystical fear of JSPs - a fear that readers of this book should not have.

JSPs should be viewed as simply Java classes, written in a special syntax more suited to Web applications, and very conveniently deployed automatically by Java servers.

Login (Authorization, Authentication) Mechanisms

Some web sites require users to log-in, typically using a username or email address, and a password. (Called authentication credentials.)

There is support for this in HTTP protocol. An error code of 401 (Not Authorized) can be sent back by a server, and the browser will present a dialog box asking for the username/password, and will resend the request with the authentication credentials.

Because people often want more control over the look and feel of the login dialog box and want it done from a regular HTML page, or they want more control over the authorization mechanisms, more commonly login is done without using the HTTP 401 code, but just using cookies/sessions.

It is not very difficult to build your own login mechanism. You could put a variable, something like `isLoggedIn` in the session. Then in every single URL, you can check the session to see if the user has logged in, and if not, redirect or forward to a login page.

Java servers do make it much easier than that. You can specify in the configuration file that you want your site to be under login control, you can specify a login page, and you can specify how to authenticate the username/password. The details of checking an `isLoggedIn` type variable, are all handled by the server in this case. This is usually handled in the application's `web.xml` configuration file, using tags like `auth-constraint`, `security-constraint` and `login-config` that provide the details of the login.

Conclusion

This book has provided you with a conceptual framework for server-side programming on Java.

It has not provided cookbook recipes, rather it has provided in-depth understanding of server-side technology. If you have faithfully worked along, now you have the understanding you need to make your own recipes. You also have the understanding necessary to choose good designs for your applications objectively.

Here is to happy and insightful server-side programming using your newfound understanding!

Code Download

Accompanying the book is a zip file containing code listings and full solutions.

Please download the zip file from

```
http://www.javaserverbook.com/code_download/
```

using the access code B9XCQVL8 on the download page.