19 COMMON GATEWAY INTERFACE (CGI)

KEY OBJECTIVES

After completing this chapter readers will be able to—

- understand the concept of server-side programming
- get an idea about the execution philosophy of CGI programs
- get an overview about languages used to write CGI programs
- understand CGI environment variables and their importance
- write basic CGI programs using C/C++, Perl, Python, etc.
- learn how to retrieve parameters from CGI programs
- get an overview about the shortcomings of CGI programs

19.1 INTERNET PROGRAMMING PARADIGM

Internet programming can be classified into two categories: *client-side programming* and *server-side programming*. In the client-side programming paradigm, programs/scripts are downloaded, interpreted, and executed by the browser. The author of the programs does not have any idea about the type and version of the browser used to execute them. So, if the browser is not compatible with the technology used, the content will not be presented properly. Let us take a specific example.

Applets are client-side programming technology, where special Java programs are embedded directly into web pages with the help of the `<applet>` tag. When a browser loads such a web page, the applet byte code is also downloaded and executed on the client side. If the browser uses an old Java Runtime Environment (JRE), applet byte code cannot be executed properly and the entire thing becomes garbled. Moreover, for large applets, download time becomes significant. Hence, this technology tends to be unacceptable. These issues have enforced businesses to use server-side programming.

19.2 SERVER-SIDE PROGRAMMING

Server-side programming solves the problem discussed in Section 19.1. The basic idea of this paradigm is that programs are executed in the web server. Hence, there is no issue of browser incompatibility or long download time. The web server sends web pages (containing simple code generated by those programs) that even an old browser can understand.

The Common Gateway Interface (CGI) is one of the important server-side programming techniques. The CGI connects a web server to an external application. Figure 19.1 shows the CGI architecture.

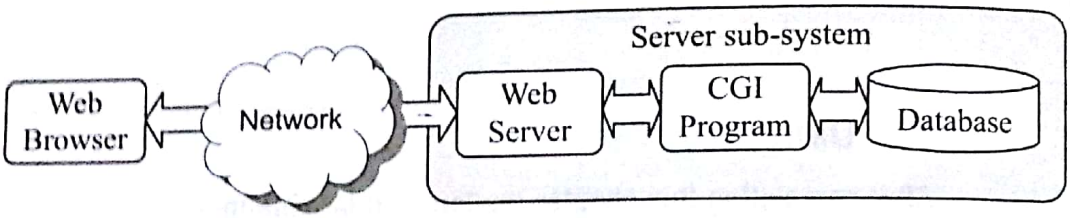


Figure 19.1 CGI Architecture

When a CGI-enabled web server receives a request for a CGI program, it does not send the file as it is. Instead, the web server executes [Figure 19.1] the program at the server end and sends the output back to the client, which is then displayed in the browser's window. This simple and elegant idea can be used to develop many powerful applications.

CGI has numerous advantages. Most of the web servers have built-in support for CGI. So, if you have your web server installed in your computer, you can start writing without any further effort.

Moreover, the CGI specification is independent of any programming language. It defines how information is transferred from a web server to an external application and from the external application to the web server. So, these external applications may be developed using a programming language that fits the application.

19.3 LANGUAGES FOR CGI

The most powerful feature of CGI technology is that virtually any programming language can be used to write CGI programs as long as it can read from a standard input and write to a standard output. Following are some popular CGI-programming languages.

C/C++

C is one of the popular programming languages. It is well known for its extremely good performance. It is widely used on many different software platforms and is a primary language for CGI. There is virtually no computer architecture for which a C compiler does not exist.

C++ supports Object Oriented Programming (OOP) and is suitable for large complex applications. So, if your CGI program should handle a large problem, C++ is ideal.

Perl

This interpreted language provides powerful text processing and file manipulation facilities. Perl borrows features from other programming languages such as C, shell script (sh), AWK, and

sed. Due to its flexibility and adaptability, it is widely used for CGI programming. It is also used for network programming, applications that require database access, system administration, and graphics programming.

Tcl

Tcl is a scripting language that was originated from "Tool Command Language", but is conventionally rendered as "Tcl". It is commonly used for CGI scripting as well as rapid prototyping, other scripted applications, GUIs, and testing.

Python

It is an interpreted, interactive, portable, Object Oriented Programming (OOP) language. Its significant power and clear syntax make Python an excellent instructional tool and ideal for Common Gateway Interface (CGI) programming. Language features include modules, classes, very high level dynamic data types, and dynamic typing.

Unix/Linux Shell

A shell is a command line interpreter that provides an interface to the users, to execute their commands. Unix/Linux shell is extremely powerful for manipulating files, pattern searching, and matching, and is ideal for CGI scripting in the Unix/Linux platform.

19.3.1 Preferred Language

The commonly used languages are Perl, C/C++, and shell script. However, your choice depends on what you want to do because different languages have different specialized features. For example, Perl is extremely powerful for string and file manipulation while C/C++ is better for complex and larger programs.

It depends on your taste as well as the facilities available on your system. If you prefer a *programming language* such as C/C++ or Fortran, you have to compile the program and generate an executable code before it runs. The original source code is no longer needed. This allows you to hide your sensitive program code from others who have access to the CGI directory. Moreover, the programs are already compiled and they take less time to start servicing than an interpreted one.

On the other hand, if you use any *scripting language*, such as Perl, Tcl, or Unix/Linux shell, you need to keep the script itself in the "cgi-bin" directory. Many programmers prefer CGI scripts instead of programs, as scripts are easier to modify, debug, and maintain than typical compiled programs.

19.4 APPLICATIONS

There are numerous applications of the CGI. It is usually used to build database applications in conjunction with HTML forms. For example, suppose we want to "hook up" our database to the World Wide Web (WWW), so that people from all over the world can query it, all we need to do is write a CGI program that can access this database. The web server will execute this program to store information to the database and receive results from the database, which are then sent back to the client.

19.5 SERVER ENVIRONMENT

Most the web servers standardize the CGI mechanism. Usually, the directory "cgi-bin" exists under the web server's installation directory. The files in this directory are treated differently. Any file requested from this special "cgi-bin" directory is not simply read and sent. Instead, it is executed in the computer where the web server is installed. The output of this program is actually sent to the browser that requested this file. The program is an executable file of typically a Perl or Python script.

Though the "cgi-bin" directory is usually considered as the container of CGI scripts, most web servers allow us to specify other directories as the CGI directory.

Suppose that you have entered the following URL in the address bar of your browser:

```
http://192.168.1.2/cgi-bin/hello.pl
```

The web server recognizes the file `hello.pl`, which is nothing but a Perl script in the `cgi-bin` directory. It then executes `hello.pl` and sends the output of this script to the browser, which then displays it on the screen.

Though CGI programs are typically written on a Unix platform, you can also write your CGI programs in a Windows environment. For this purpose, you need a web server. Perl is the typical language used for CGI programs, but you can use C/C++ or Python as well. In this section, we shall discuss how to install and configure Apache and Perl on your computer running Windows. Once Apache and Perl are successfully installed, you can test your CGI programs using the address `http://locahost` in your own computer. This is quite useful to test your CGI programs locally before installing them in the actual server.

Download the installer file for Apache from `http://httpd.apache.org/download.cgi`. Now, install it by double clicking on the installer file. We are assuming that you have installed Apache in the directory "D:\Apache Software Foundation".

Now, download and install ActivePerl from `http://www.activestate.com/Products/ActivePerl/`. We are assuming that you have successfully installed Perl in the "D:\Perl" directory.

19.5.1 Configuring Apache

To enable CGI programs, you need to modify the Apache configuration file `httpd.conf`, which can be found in the `conf` directory under the apache installation directory. Go to the `<Directory>...</Directory>` section. Uncomment or add the following line:

```
Options MultiViews Indexes SymLinksIfOwnerMatch Includes ExecCGI
```

The `Options` specifies what options are available in this directory. `ExecCGI` enables the CGI scripting. Go to the `AddHandler` section and add or uncomment the following line.

```
AddHandler cgi-script .cgi .pl
```

This causes files with the extensions `.cgi` and `.pl` to be treated as CGI programs. Finally, save the configuration file and restart Apache. Ensure that the server restarted successfully by checking `http://localhost/` in your browser.

If everything goes well, your web server is ready to serve the CGI request. Save your CGI programs in the `cgi-bin` directory under the Apache installation directory.

Use the following line as the first line of your Perl program:

```
#!D:/perl/bin/perl.exe
```

This tells the web server where to find the Perl interpreter to interpret Perl programs.

19.6 ENVIRONMENT VARIABLES

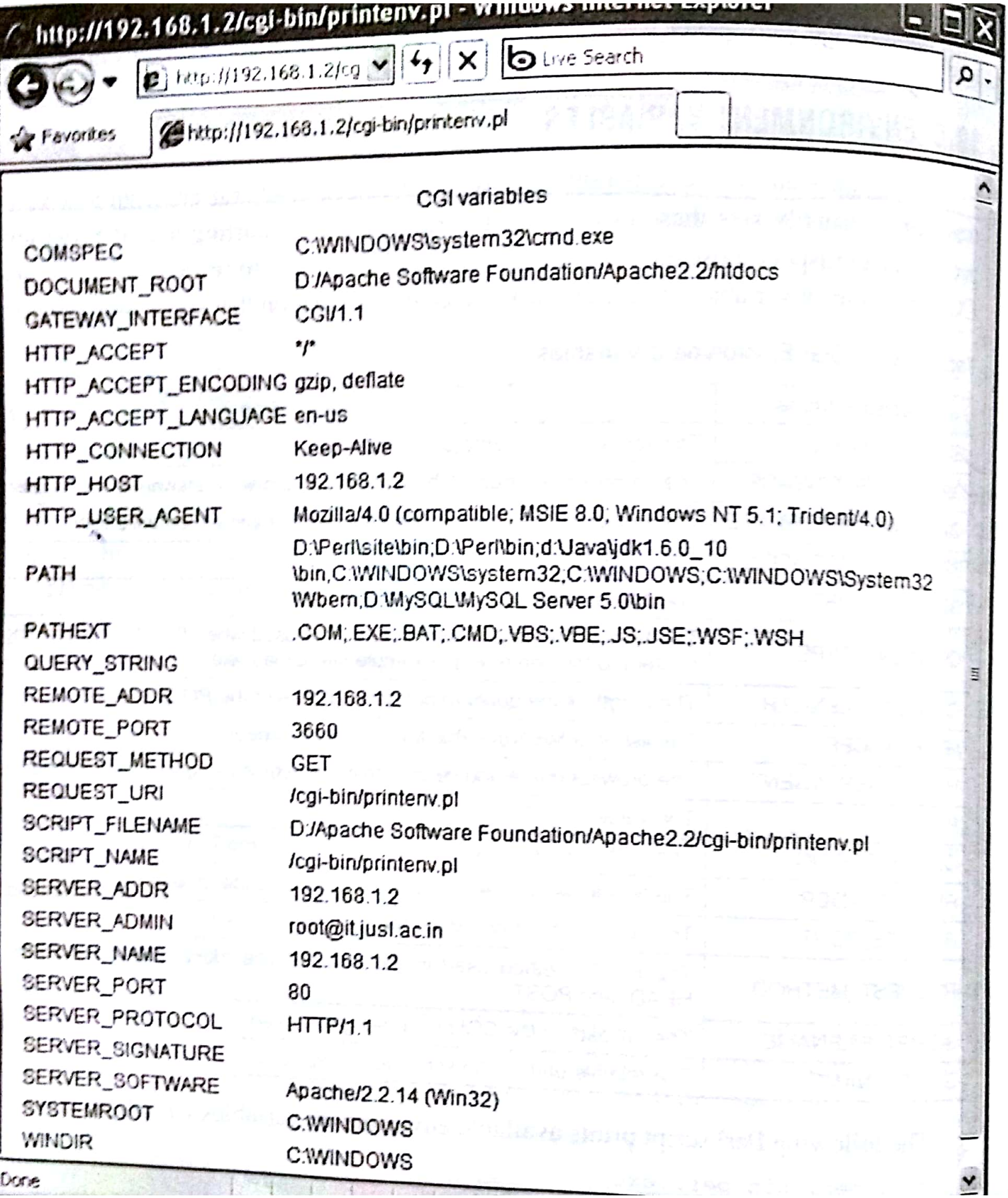
In order to pass and retrieve parameters, web servers use several environment variables. The web server usually sets these environment variables before starting a CGI program. The CGI program can inspect those environment variables [Table 19.1] to retrieve information. Note that CGI environment variables are the primary source of information that server-side programs can use.

Table 19.1 CGI Environment Variables

Variable Name	Description
SERVER_NAME	The server's DNS name or IP Address
SERVER_SOFTWARE	The name and version of the web server software answering the request
GATEWAY_INTERFACE	The version of the CGI specification that the server complies with
SERVER_PROTOCOL	The HTTP version used by the server
SERVER_PORT	The port number used by the web server
CONTENT_TYPE	The type of the data of the content. It is used when the client is sending attached content to the server e.g., example file upload, etc.
CONTENT_LENGTH	The length of the query information in case of the POST method
HTTP_ACCEPT	The list of MIME types that the client can accept
HTTP_USER_AGENT	The browser name and version that the user is using to make this request
PATH	The value of PATH environment variable
QUERY_STRING	The URL-encoded information which follows the ? in the URL
REMOTE_ADDR	The IP address of the remote host that made the request
REMOTE_PORT	The port number from which the request was sent
REQUEST_METHOD	The HTTP method used in the request. The most common methods are GET, HEAD, and POST
SCRIPT_FILENAME	The full path of the CGI script being executed
SCRIPT_NAME	The relative path of the script being executed

The following Perl script prints available environment variables with their values.

```
#!D:/perl/bin/perl.exe
print "Content-type: text/html\n\n";
print "<table style=\"font-size:12; font-family:arial\" border=\"0\" >";
print "<caption>CGI variables</caption>";
foreach $var (sort(keys(%ENV))) {
   $val = $ENV{$var};
   print "<tr><td>${var}</td><td >${val}</td></tr>\n";
}
print "</table>";
```



- Read parameters passed to this script
- Process these parameters
- Write the HTML response to the standard output

19.8 CGI SCRIPTING USING C, SHELL SCRIPT

Writing CGI programs using C/C++ language is somehow different from writing them in shell script. Here, shell refers to the Unix/Linux shell. The Windows shell is not so powerful and is hardly used. Note that shell scripts are interpreted by the underlying shell. You simply need to mention which shell should be used to interpret your script in the first line of your script as follows:

```
#!/bin/bash
```

C/C++ programs are, on the other hand, compiled programs. Writing CGI programs using C/C++ basically consists of two steps:

- Compile the program to generate the executable file
- Put this executable file in the `cgi` directory.

Compiling a C/C++ program is different for different compilers. Ask your system administrator to know the compilation procedure of C/C++ programs. Moreover, C/C++ programs are not platform-independent. So, you must compile your program in the computer where the web server runs, using a suitable compiler available on that platform.

19.9 WRITING CGI PROGRAMS

In this section, we shall develop simple programs to display "Hello World!" using different languages such as Perl, C, and Python. Following is the program (`hello.pl`) written in Perl.

```
#!D:/perl/bin/perl.exe
print "Content-type: text/html\n\n";
print "<html>\n";
print "  <head><title>First Perl script</title></head>\n";
print "  <body>\n";
print "     <h2>Hello, World!</h2>\n";
print "  </body>\n";
print "</html>\n";
```

The script `hello.pl` is a simple script that prints a simple HTML document on the standard output, i.e., screen. The first line of any Perl script should look like this:

```
#!D:/perl/bin/perl.exe
```

The `#!` indicates that this is a script. The path `D:/perl/bin/perl.exe` refers to the full path name of the Perl interpreter to be used by the web server to execute the Perl script. However, this could be different in different systems. For a Unix or Linux OS, this is typically `/usr/bin/perl` or `/usr/local/bin/perl`. If you are not sure about the path, type "whereis perl" or "which perl" at the command prompt, it shows the path where the Perl interpreter is stored. Alternatively, ask the webmaster about the location of the Perl interpreter.

The remaining part consists of actual Perl statements. Before printing anything else, you should use the following line:

```
print "Content-type: text/html\n\n";
```

This prints `Content-type: text/html` followed by a blank line. This is sent as a HTTP response header, which specifies the type of the content to be displayed in the browser's screen. In our case, we shall send an HTML document and that is why the `Content-type` header is specified as `text/html`. This Perl program, except `Content-type` header, basically generates the following:

```
<html>
   <head><title>First Perl script</title></head>
   <body>
      <h2>Hello, World!</h2>
   </body>
</html>
```

If you enter the following URL, you will see the result shown in Figure 19.3.

```
http://192.168.1.2/cgi-bin/hello.pl
```

Here, 192.168.1.2 is the IP address of the computer where the web server is running. If you are testing your program locally, you can also use the following URL:

```
http://localhost/cgi-bin/hello.pl
```

Alternatively, you can use

```
http://127.0.0.1/cgi-bin/hello.pl
```

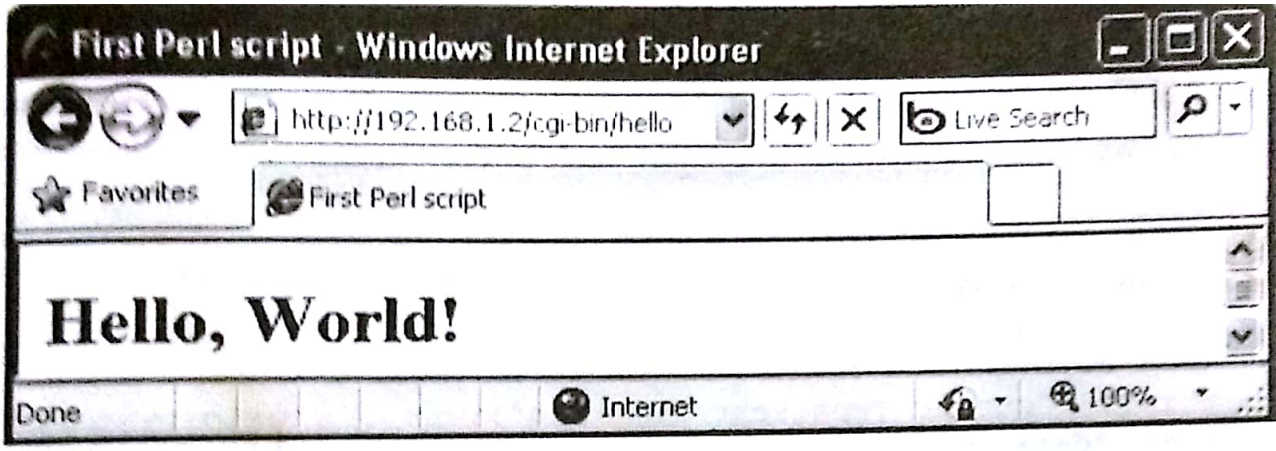


Figure 19.4 Hello world CGI program using C

The following program is written in Python (`hello.cgi`) and generates the same

```
#!D:\Python31\python.exe
```

: steps:

Get the information from the proper environment variable, depending upon the method type. For example, we can retrieve information from the `QUERY_STRING` environment variable for the GET method.

Change all placeholders to their correct values.

Split each name–value pair.

Convert hexadecimal values back to their original characters.

Find the respective name and value.

'he following C program (`add.c`) shows how to retrieve two parameter values. The program those two values and sends the result back to the client.

```
/add.c
include <stdio.h>
include <stdlib.h>
nt main(void) {
  long a, b;
  printf("Content-Type:text/html\n\n");
  char *data = getenv("QUERY_STRING");
  sscanf(data,"a=%d&b=%d", &a, &b);
  printf("%d + %d = %ld", a, b, a + b);
  return 0;
```

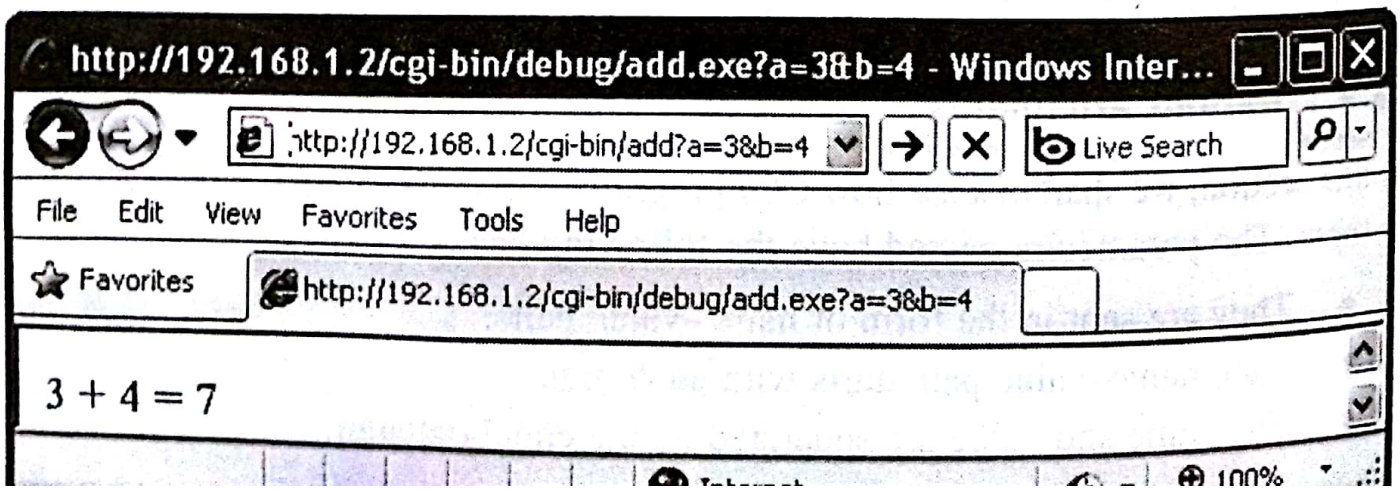
ere, we first obtain the entire parameter information using the `getenv()` function. The idual parameters are then obtained using the `sscanf()` function. Compile the program to an executable file add using the following command:

```
:c -o add add.c
```

ow, use the following URL to test the program.

```
:tp://192.168.1.2/cgi-bin/add?a=3&b=4
```

ıe result is shown in Figure 19.5.



Following is the equivalent Perl program.

```
#!D:/perl/bin/perl.exe
use CGI qw(:standard);
print "Content-type: text/html\n\n";
$a = param('a');
$b = param('b');
$c = $a + $b;
print "$a + $b = $c";
```

The following Perl script retrieves all parameters appended to the URL.

```
#!D:/perl/bin/perl.exe
print "Content-type:text/html\r\n\r\n";
$buffer = $ENV{'QUERY_STRING'};
@pairs = split(/&/, $buffer);
foreach $pair (@pairs)    {
  ($name, $value) = split(/=/, $pair);
  print "$name=$value<br>";
}
```

19.10 CGI SECURITY

The CGI technology allows users to run programs in a system remotely using their URLs. This may become vulnerable. Therefore, some security precautions need to be taken when CGI technology is used. Note that CGI programs are kept in a special directory, so that the web server can execute the program rather than just send them to the browser. This is the most important concept that CGI writers follow: The external users may try to use this concept to perform malicious events. The CGI directory is usually controlled by the webmaster, who should prohibit the average user from creating and running CGI programs. There are many ways to allow access to CGI scripts, but the webmaster decides how to set these up for you.

Although the CGI protocol is inherently secure, CGI scripts may become a major source of security holes. CGI scripts should be written with utmost care just as the server. The web administrator should not also trust the script writers and should not install arbitrary CGI scripts.

If you are a system administrator, a webmaster, or are otherwise involved with the administration of a network, you should create a security policy for your website. The following points should be taken into consideration while writing the security policy:

- Users who can access the system
- When they are allowed to use the system
- What operations they are allowed to perform
- A way to grant access to the system
- Acceptable permission to use the file system
- Procedure for monitoring the system
- Actions to be taken against suspected security breaches

19.11 ALTERNATIVES AND ENHANCEMENTS TO CGI

Each time the web server receives a CGI request, it starts a new process that serves the request. Starting a new process may take significant amount of time and memory than the actual task of generating the output. If users send CGI requests very frequently, they may quickly overwhelm the web server.

If the CGI program is an interpreted program such as Perl, Python, or shell script, it needs more time. We can use compiled programs such as C/C++, instead of scripting languages, to avoid overhead involved in interpretation.

The overhead in process creation may be reduced by using technology such as FastCGI or by using special extension modules provided by some web server.

FastCGI uses a single persistent process that handles many requests during its lifetime. This way, overhead involved in new process creation for every request may be avoided. Multiple simultaneous requests can be handled either by using multiple processes or using a single process with internal multiplexing. FastCGI allows web servers to perform simple operations, such as reading a file before the request is handed over to the FastCGI program. Some of the web servers that implement FastCGI are Apache HTTP Server, Microsoft IIS, Resin Application Server, Sun Java System Web Server, etc.

Another alternative to CGI is Simple Common Gateway Interface (SCGI), which is similar to FastCGI but easier to implement. In this technology, clients send requests to the SCGI server using an SCGI request message. The server sends the response back and closes the connection. Web servers that implement SCGI are Apache HTTP Server, Lighttpd, Cherokee, etc.

Another viable and effective solution to CGI is Java servlets. Servlets can avoid overhead involved in new process creation. Web servers load the servlet class when it starts up. The servlet can then serve many requests using a separate thread. Since threads are more lightweight than processes, servlets are more efficient than CGI programs.

Java Server Page (JSP) is an extension of Java servlet and is a potential solution to CGI technology. In the next two chapters, we shall discuss Java servlet and JSP, separately.

KEYWORDS

CGI alternatives: The competent technologies such as servlets, JSP, ASP, and PHP.

CGI directory: CGI programs are kept in some predefined special directory called CGI directory so that the web server knows that those programs must be executed, instead of being sent directly to the clients.

CGI Environment variables: The variables used to pass and retrieve information in CGI programs.

CGI Languages: The programming languages that can be used to write CGI programs.

Client-side programming: A programming paradigm where programs are executed in the client machine.

Common Gateway Interface: A standard that interfaces the HTTP server software with the CGI programs that run on the server.

Compiled CGI program: A program that is compiled to generate executable code, which acts as a CGI program.

Gateway program: A program that the web server contacts to process some request sent by the client.

Interpreted CGI program: A program whose statements are interpreted using an interpreter program.

SCGI: An extension to CGI called Simple Common Gateway Interface.

Server-side programming: A programming paradigm where programs are executed in the server machine.

SUMMARY

The Common Gateway Interface (CGI) is one of the popular server-side technologies. CGI has numerous advantages. Most of the web servers have built-in support for CGI. Moreover, the CGI specification is independent of any programming language; it defines how information is transferred from the web server to an external application and from the external application to the web server. So, these external applications may be developed using a programming language that fits the application.

When a CGI-enabled web server receives a request for a CGI program, it does not send the file as it is. Instead, the web server executes the program and sends the output back to the client, which is then displayed in the browser's window.

The most powerful feature of CGI technology is that virtually any programming language can be used to write the CGI program as long as it can read from a standard input and write to a standard output. Some examples are C/C++, Perl, Python, Tcl, shell, Ruby, etc. Compiled programs such as C/C++ run faster than interpreted programs. Interpreted programs, on the other hand, easier to modify, debug, and maintain.

Most web servers standardize the CGI mechanism. Usually, the directory "cgi-bin" exists under the web server's installation directory. The files in this directory are treated differently. Any file requested from this special "cgi-bin" directory is not simply read and sent. Instead, it is executed in the computer where the web server is installed. The output of this program is actually sent to the browser that requested this file. The program usually is pure executable file of typically a Perl or Python script.

To pass and retrieve parameters, web servers use several environment variables. The web server usually sets these environment variables before starting a CGI program. The CGI program can inspect those environment variables to retrieve the information.

Since the CGI program can be written in almost all languages and in all platforms, the exact procedure for executing the CGI program varies from one platform/web server to another.

The CGI technology allows users to run programs in a system remotely using their URLs. This may become vulnerable. Therefore, some security precautions need to be taken when CGI technology is used.

WEB RESOURCES

`http://www.ietf.org/rfc/rfc3875.txt`
The Common Gateway Interface (CGI) Version 1.1

`http://hoohoo.ncsa.illinois.edu/cgi/`
The Common Gateway Interface

`http://www.w3.org/CGI/`
CGI: Common Gateway Interface

`http://www.fastcgi.com/devkit/doc/fcgi-spec.html`
FastCGI Specification

EXERCISES

Multiple Choice Questions

1. Server-side scripting is about "programming" the behavior of the
   (a) Server (b) Browser
   (c) HTML (d) All of the above

2. What is the full form of CGI?
   (a) Common Graphical Interface
   (b) Cascading Gateway Interface
   (c) Class Generator Instance
   (d) Common Gateway Interface

3. Which of the following defines how a web server communicates with external applications?
   (a) HTTP (b) CGI
   (c) SMTP (d) TCP/IP

4. Which of the following is true regarding CGI?
   (a) It is a client-side technology
   (b) It is the name of a browser
   (c) It is a server-side technology
   (d) It is a web server

5. Which of the following languages can be used as a CGI language?
   (a) C
   (b) C++
   (c) Perl
   (d) All of the above

6. Which of the following directories is usually used to store CGI programs?
   (a) cgi-bin
   (b) cgi
   (c) cgiprogs
   (d) bin

7. Which of the following environment variables is used to retrieve URL-encoded information?
   (a) QUERY
   (b) QUERY_STRING
   (c) PARAM
   (d) URL_ENCODE

8. What should be the value of Content-type to generate HTML documents?
   (a) text/html
   (b) text/plain
   (c) plain/text
   (d) plain/html

9. Whenever a web server receives a CGI request, it
   (a) Creates a new thread
   (b) Uses an existing process
   (c) Uses an existing thread
   (d) Creates a new process

10. Which of the following statements is true regarding CGI?
    (a) Compiled programs are faster than interpreted programs
    (b) Compiled programs are slower than interpreted programs
    (c) The source code is necessary for compiled programs
    (d) The source code is not necessary for interpreted programs

11. Which of the following is a compiled language?
    (a) C++
    (b) Perl
    (c) Python
    (d) Shell

12. Which of the following is an interpreted language?
    (a) Tcl
    (b) Ruby
    (c) Visual Basic
    (d) All of the above

13. Which of the following headers should be set by a CGI script?
    (a) Value-type
    (b) Content
    (c) Content-type
    (d) Return-type

14. Which of the following CGI environment variables represents the port number used by the web server?
    (a) REMOTE_PORT
    (b) SERVER_PORT
    (c) PORT
    (d) HOST_PORT

15. Which of the following functions in C/C++ is used to retrieve the value of an environment variable?
    (a) env()
    (b) environment()
    (c) getenv()
    (d) getvariable()

16. Which of the following lines should be printed by a CGI program to generate an HTML document?
    (a) Content-type: html/text
    (b) Content-type: text/html
    (c) Return-type: text/html
    (d) Return-type: html/text

17. The first line of any CGI script should start with
    (a) $!
    (b) &!
    (c) @!
    (d) #!

18. Which of the following characters is used to separate a parameter from a URL?
    (a) ?
    (b) &
    (c) $
    (d) %

19. Which of the following characters is used to separate a parameter name and its value?
    (a) ^
    (b) $
    (c) =
    (d) &

20. Which of the following functions is used in Perl to print a string on the standard output?
    (a) print
    (b) show
    (c) display
    (d) out

20 SERVLET

KEY OBJECTIVES

After completing this chapter readers will be able to—

- get an idea about server-side technology
- understand the advantages of Java servlet technology over other similar technologies
- get an idea about the basic execution philosophy of servlets
- understand how to write, deploy, and invoke servlets
- learn how to install and configure Tomcat servlet engine
- learn how to use important concepts such as filters, session tracking, etc.

20.1 SERVER-SIDE JAVA

Servlet is a Java technology for server-side programming. It is a Java module that runs inside a Java-enabled web server and services requests obtained from the web server. Servlets are not tied to a specific client–server protocol, but are most commonly used with HTTP. The word "servlet" is often used to mean "HTTP servlet". Execution of a servlet consists of four steps [Figure 20.1]:

- The client sends a request to the web server.
- The web server interprets it and forwards it to the corresponding servlet.
- The servlet processes the request, generates the output (if any), and sends it back to the web server.

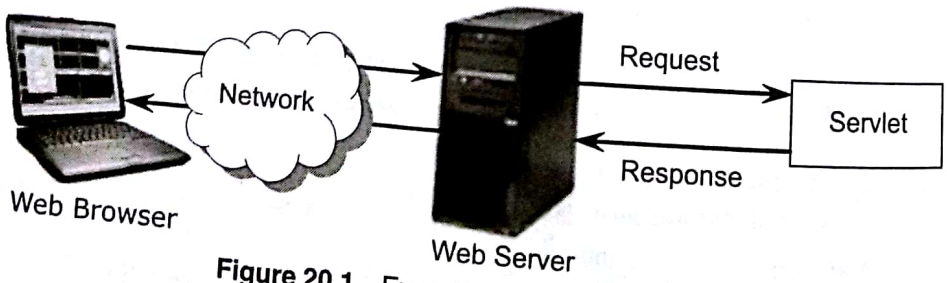


Figure 20.1 Execution of a Java servlet

- The web server sends the response back to the client. The browser then displays it on the screen.

It runs entirely within the Java Virtual Machine (JVM). Since it runs on the server and generates simple output, which even older versions of browsers can interpret, there is no browser incompatibility.

20.2 ADVANTAGES OVER APPLETS

Applets, upon download from the server, run in the client's browser. They will function properly, provided a proper Java Runtime Environment (JRE) is installed in the client's browser. If the client uses old fashioned browsers and the applets use advanced features, the browsers may fail to execute them.

Servlets run on the server machine and usually generate simple HTML codes. So, even an older version of the browser can display these HTML pages correctly.

Applets cannot access local resources such as files and databases. Servlets, if configured properly, have full access to system resources. Therefore, servlets are more powerful than applets.

20.3 SERVLET ALTERNATIVES

There are several alternatives to servlets. However, each has its own set of problems. Some of the other server-side technologies are discussed briefly here.

20.3.1 Common Gateway Interface (CGI)

CGI is one of the most common server-side solutions. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. As described in Chapter 19, a CGI application is an independent program that receives requests from the web server and sends it back to the web server. Some of the common problems of this technology are as follows:

- A new process is created every time the web server receives a CGI request. Since a new process is created and initialized, it results in the delay of response time. Servers may also run out of memory if not configured properly.
- As mentioned earlier, a CGI application can be written in almost every language and most of them are not platform-independent. The common platform-independent language is Perl. Although Perl is a very powerful text processing language, for every request it requires a new interpreter to be started. This makes the overall response time longer.
- Since a CGI application is a completely separate process, if it terminates abnormally before responding, the web server has no way to identify what happened there.

20.3.2 Proprietary APIs

Many proprietary web servers have built-in support for server-side programming. Examples include Netscape's NSAPI, Microsoft's ISAPI, and O'Reilly's WSAPI. None of these APIs is free and the newer versions may not be backward compatible. Most of these are developed in C/C++ languages and hence can contain memory leaks and core dumps that can crash the web server.

20.3.3 Active Server Pages (ASP)

Microsoft's Active Server Pages (ASP) is another technology that supports server-side programming. Unfortunately, the only web server that supports this technology is Microsoft's Internet Information Server (IIS), which is not free. Although some third-party products support ASP, they are not free either.

20.3.4 Server-side JavaScript

Server-side JavaScript is another alternative to servlets. However, the only known servers that support it are Netscape's Enterprise and FastTrack servers. This ties you to a particular vendor.

20.4 SERVLET STRENGTHS

Java servlet technology has some distinct advantages over other competent technologies. Truly speaking, except JSP, no technology can compete with servlet technology. Following are some advantages of Java servlet technology.

20.4.1 Efficient

When a servlet gets loaded in the server, it remains in the server's memory as a single object instance. Each new request is then served by a lightweight Java thread spawned from the servlet. This is a much more efficient technique than creating a new process for every request, as done in CGI. Servlets also have more alternatives for optimizations, such as caching the previous computation and keeping database connections open.

20.4.2 Persistent

Servlets can maintain the session by using the session tracking/cookies, a mechanism that helps them track information from request to request. Critical information can also be saved to a persistent storage and can be retrieved from it when the servlet is loaded next time.

20.4.3 Portable

Servlets are written in Java and so they are portable across operating systems and server implementations. This allows servlets to be moved across new operating systems seamlessly. We can develop a servlet on a Windows machine running the Tomcat or any other server and later we can deploy that servlet effortlessly on any other operating system such as a Unix server running an iPlanet/ Netscape application server. So, servlets are **W**rite **O**nce **R**un **A**nywhere (WORA) programs.

20.4.4 Robust

Since servlets can use the entire JDK, they can provide extremely robust solutions. Java has a very powerful exception handling mechanism and provides a garbage collector to handle memory leaks. It also has a very large and rich class library with network and file support, database access, distributed object components, DOM support, security, utility packages, etc.

20.4.5 Extensible

Servlets are nothing but Java technology, which is popular for its well-known features such as platform independence, garbage collection, exception handling, and extremely powerful utility packages. Servlets can also make use of Java's object orientation features, especially inheritance. A sub class inherits all the features of its super class. Additional features and methods can be added and hence it is easily extendable.

20.4.6 Secure

Servlets run in a Java-enabled web server. So, they can use security mechanisms from both the web server and the Java Security Manager.

20.4.7 Cost-effective

There are a number of free web servers available [Table 20.1] for personal and commercial use. Java Development Kit (JDK) is also free and can be downloaded from `http://java.sun.com`. So, applications can be developed practically without any cost.

Table 20.1 Standalone web server

Product	Source	Product	Source
Apache Web Server	Apache	Web Server	ZEUS
WebSphere Application Server	IBM	iPlanet	Netscape
Weblogic Application Server	BEA	J Application Server	GenStone
Resin	Caucho	Netscape Enterprise Server	Netscape
JRUN	Adobe	LiteWebServer	Gefion
Orion Application Server	Orion	Java Web Server	Sun
Oracle Application Server	Oracle	CtO-JStar	JavaOne
Dynamo Application Server	ATG	iServer	ServerTec
J2EE Server	Pramati	Domino Go WebServer	Lotus
AppServer	Borland	Java Servlet Server 2.0	Paperclips
Jetty	Eclipse Foundation	KonaSoft Enterprise Server	KonaSoft
Jigsaw Server	W3C	Enhydra	ObjectWeb

Table 20.2 also shows some other add-on software that support Java servlet technology.

Table 20.2 Add-on Servlet engine

Product	Source

servlets. The second package is provided for servlets that can handle HTTP requests. Figure 20.2 shows the Servlet architecture.

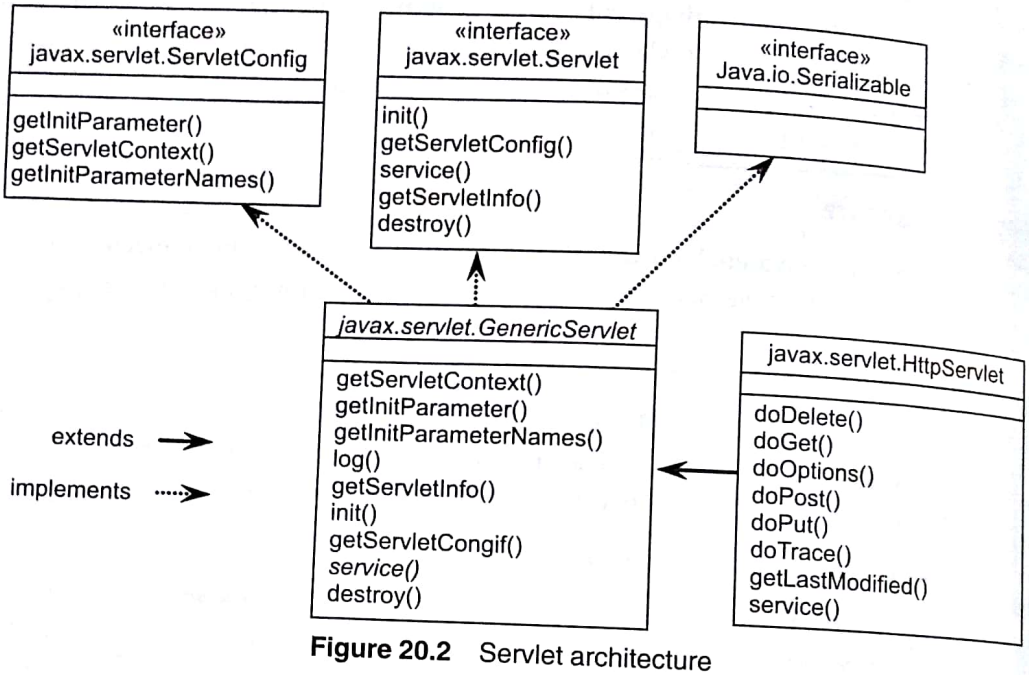


Figure 20.2 Servlet architecture

The top-level interface of the servlet architecture is `javax.servlet.Servlet`. It provides the basic functionalities of all the servlets that are created by implementing this interface directly or indirectly. For example, the `init()` method initializes the servlet, the `service()` method serves the client request, and the `destroy()` method shuts the servlet down.

20.6 SERVLET LIFE CYCLE

The container in which the servlet has been deployed supervises and controls the life cycle of a servlet. Typically, this container is nothing but a web server. When the container receives a request from a client and determines that the request should be handled by a servlet, it performs the following steps.

- If an instance of the target servlet does not exist, the container does the following:
  - o Finds and loads the servlet class
  - o Creates an instance of the servlet class
  - o Calls the `init()` method on this servlet instance to initialize it
- Otherwise (i.e., if the target servlet exists), it invokes the `service()` method on it, passing a `ServletRequest` type object and a `ServletResponse` type object.
- If the container decides that the servlet is no longer needed, it removes and finalizes the servlet by calling the servlet's `destroy()` method.

20.6.1 init()

A servlet's life begins here. This method is called *only once* by the web container, just after it instantiates and loads the servlet. So, it is a good idea to read the persistent configuration data, initialize resources that will be used during the rest of the time, and perform any other one-time activity by overriding the `init()` method. Once the servlet is initialized, it becomes ready to handle the client's request. The prototype of the `init()` method is as follows:

```
void init(ServletConfig)
```

The web container passes a `ServletConfig` object, which contains the startup configuration for this servlet such as initialization parameters.

20.6.2 service()

This is the most important method, whose signature is as follows:

```
void service(ServletRequest request, ServletResponse response)
```

This method gets called each time the web container receives a request intended for this servlet. The web container calls the `service()` method on a servlet with two objects: a `ServletRequest` type object `request` and a `ServletResponse` type object `response`. The `request` object encapsulates the communication from the client to the server, while the `response` object encapsulates the communication from the servlet back to the client.

The `ServletRequest` interface provides methods to retrieve information sent by the clients. Similarly, the `ServletResponse` interface provides methods to send data to the clients.

20.6.3 destroy()

The method signature is as follows:

```
void destroy();
```

This method gets called when the web container uninstalls the servlet. This method is overridden to clean up the resources that were allocated to this servlet. It also makes sure that any persistent state is synchronized with the servlet's current in-memory state.

20.6.4 Other Methods

The `javax.servlet.Servlet` interface also provides the following methods to inspect the properties of a servlet at runtime.

```
ServletConfig getServletConfig();
```

Returns a `ServletConfig` object, which contains the initialization parameters and startup configuration used for this servlet.

```
String getServletInfo();
```

Returns a `String` object containing information about the servlet, such as its version, author, copyright, etc.

20.7 GENERICSERVLET AND HTTPSERVLET

The class `javax.servlet.GenericServlet` implements the `javax.servlet.Servlet` interface and, for convenience, the `javax.servlet.ServletConfig` interface. A servlet class is usually created by extending either the `GenericServlet` class or its descendant `javax.servlet.http.HttpServlet` class unless the servlet needs another class as a parent. If a servlet does need to be a subclass of another class, it must implement the `Servlet` interface directly. This would be necessary when, for example, RMI or CORBA objects act as servlets. In such a case, the servlet class must implement all the methods of the `Servlet` interface.

The `GenericServlet` class defines a generic protocol-independent servlet, in the sense that it can be extended to provide implementation of any protocol, such as HTTP, FTP, and SMTP.

The `GenericServlet` class was created to make writing servlets easier. It provides implementations of the life cycle methods `init()` and `destroy()`, as well as methods in the `ServletConfig` interface. The servlet writer has to implement only the `service()` method. This is required because it is the method that is called every time a client requests for the servlet. The prototype of this method is shown as follows.

```
public void service(javax.servlet.ServletRequest request,
        javax.servlet.ServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException;
```

This method expects two arguments. `request` encapsulates the client request and is used to extract information from the client's request. On the other hand, `response` contains information to be sent back to the client.

The `HttpServlet` class extends `GenericServlet` and provides a framework for handling HTTP requests. Servlets that want to handle only HTTP requests are usually created by extending this class. It provides an implementation for the `service()` method. So, you do not have to implement the `service()` method if you extend the `HttpServlet` class. The `service()` method in the `HttpServlet` class reads the method type stored in the request and invokes a specific method based on this value. Specifically, if the method type is GET, it calls `doGet()`; if the method type is POST, it calls `doPost()`; and so on These are the methods that we need to override. The list of all such methods is given in Figure 20.3.

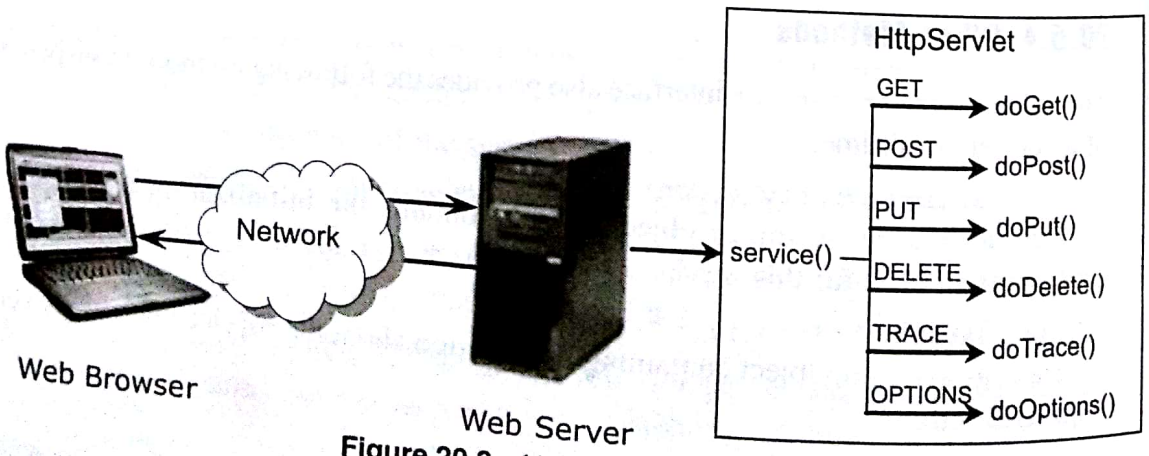


Figure 20.3 HttpServlet in action

Each of these methods takes two arguments: an `HttpServletRequest` type object and an `HttpServletResponse` type object. Following is the prototype of the `doGet()` method:

```
void doGet(HttpServletRequest request, HttpServletResponse response)
```

The `HttpServletRequest` interface extends the `ServletRequest` interface and provides all the functionalities of the `ServletRequest` interface. Additionally, it provides methods to retrieve HTTP headers, cookies sent, and other HTTP-specific information.

The `HttpServletResponse` interface extends the `ServletResponse` interface and provides all the functionalities of the `ServletResponse` interface. Additionally, it provides methods to send HTTP-specific information back to the client.

Servlets created by extending the `HttpServlet` class can handle multiple simultaneous requests. For each request, a thread is created, which runs its `service()` method. If you want to create a single-threaded servlet, your servlet must also implement the `SingleThreadModel` interface as follows:

```
public class SimpleServlet extends HttpServlet implements SingleThreadModel { }
```

The interface `SingleThreadModel` does not define any method. It is used to merely declare that the servlet should use a single thread.

20.8 FIRST SERVLET

Our first servlet is a simple servlet designed to handle the HTTP GET method. Create the following servlet `HelloWorldServlet.java` using any text editor.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorldServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><head><title>Hello World Servlet</title></head>");
      out.println("<body>");
      out.println("<h1>Hello World!</h1>");
      out.println("</body></html>");
      out.close();
   }
}
```

The class `HelloWorldServlet` extends the `HttpServlet` class. So, it can handle HTTP requests. It is written to handle only the GET method. Therefore, only the `doGet()` method is overridden. The output of this servlet is an HTML document. This is mentioned using the `setContentType()` method. To send the data back to the client, a `PrintWriter` object is obtained by calling the `getWriter()` method on the `response` object. Finally, it sends an HTML document as follows:

```
<html><head><title>Hello World Servlet</title></head>
<body>
<h1>Hello World!</h1>
</body></html>
```

At the end, we close the `PrintWriter` object. It is not mandatory as the web container closes the `PrintWriter` or `ServletOutputStream` automatically, when the `service()` method returns. An explicit call to `close()` is useful when you want to perform some processing after responding to the client's request. This also tells the web container that the response is completed and the connection to the client may be closed as well.

20.8.1 Installing Apache Tomcat Web Server

Note that the procedure for installing servlets varies from web server to web server. Please refer to the documentation of your web server for definitive instructions. All the servlets are tested in this book using Apache's Tomcat web server. Tomcat is a lightweight, easily configurable free web server and is widely used by users who are new to servlets.

Let us now discuss how to install and configure the Tomcat web server.

Before installing Tomcat, make sure that you have installed either Java 6 Standard Edition (JDK 1.6) or Java 5 Standard Edition (JDK 1.5) in your computer. Download and install Java from `http://java.sun.com/javase/downloads/index.jsp`. Once you have installed, include the Java "bin" directory in the PATH environment variable. Additionally, create an environment variable JAVA_HOME and set it by the Java installation directory.

Download the necessary files from `http://tomcat.apache.org/download-60.cgi`. If you have downloaded an installer file, install it. If it is a `.zip` file or `.zip.tar` file, simply unzip it in a folder. We shall assume that you have installed Tomcat in the `D:\apache-tomcat-6.0.16` directory.

Tomcat has a predefined directory structure. A typical directory structure is shown in Figure 20.4.

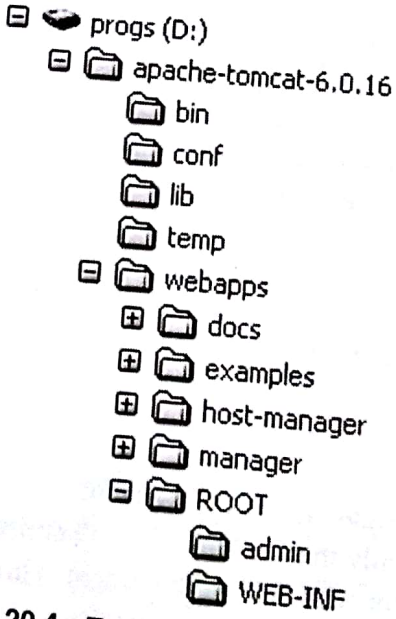


Figure 20.4 Tomcat directory structure

Suppose `$TOMCAT_HOME` represents the root of the Tomcat installation directory. According to Figure 20.4, it represents the `D:\apache-tomcat-6.0.16` directory. Following are some key Tomcat directories under `$TOMCAT_HOME`:

- `\bin` — Startup, shutdown, and other scripts. The `*.bat` (for Windows) files and Unix/Linux counterparts `*.sh` files.
- `\conf` — Configuration XML files. The primary configuration file is `server.xml`, which describes how the server starts, behaves and finds other information. All the information in the configuration files is loaded whenever the web server starts up. So, any change to the files needs a restart of the web server.
- `\lib` — Binary jar files are needed for the server to function. You should put your custom jar files in this directory. Tomcat can find them at runtime.
- `\webapps` — It contains web applications. The directory `ROOT` under it contains the default web application, which can be accessed using the URL `http://192.168.1.2:8080`, where `192.168.1.2` is the IP address of the computer where this web server is running. You should create your own directory under it. We shall create a directory `wt` (short for web technology) to put our servlet files. So, the document root of our website will be `http://192.168.1.2:8080/wt`.

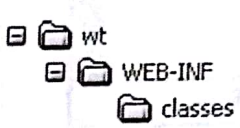
20.8.2 Building and Installing Servlet

To compile our `HelloWorldServlet.java` file, servlet class files are required. Tomcat comes with a .jar file (`servlet-api.jar` for version 6.0.16) that contains necessary class files. You can find this jar file usually in the `$TOMCAT_HOME\lib` directory.

Compile the `HelloWorldServlet.java` using the following command:

```
javac -cp d:\apache-tomcat-6.0.16\lib\servlet-api.jar HelloWorldServlet.java
```

This generates the file `HelloWorldServlet.class` that contains the byte code for our servlet. Create the following directory structure.



Put this `HelloWorldServlet.class` file in the `$TOMCAT_HOME\webapps\wt\WEB-INF\classes` directory.

The servlet class file is now ready to use. However, we have to inform the web server about the existence of this servlet and the URL that will be used to refer to this servlet. This is specified in the `$TOMCAT_HOME\wt\WEB-INF\web.xml` file, which is the configuration XML file for this website. If the file does not exist, create this file first.

Now, insert the following lines in this file. Some IDE provides interfaces to generate the same information. However, it is recommended to the beginners to do it manually. When you have sufficient knowledge about how to install servlets and how they work, you can use IDEs.

```
<servlet>
    <servlet-name>HelloWorld</servlet-name>
   <servlet-class>HelloWorldServlet</servlet-class>
</servlet>
```

This code maps the servlet class (`HelloWorldServlet.class`) file to a servlet name (`HelloWorld`). You then need to map this servlet name to a URL to be used to invoke this servlet. This is a URL relative to the document root for this website.

```
<servlet-mapping>
   <servlet-name>HelloWorld</servlet-name>
   <url-pattern>/servlet/HelloWorld</url-pattern>
</servlet-mapping>
```

In our case, the document root is `http://192.168.1.2:8080/wt`. So, the complete URL of this servlet will be `http://192.168.1.2:8080/wt/servlet/HelloWorld`.

20.8.3 Invoking Servlet

Invoking this servlet is very easy. Start the Tomcat web server. Type the following URL in the address bar of your web browser and press enter. Make sure that the computer where the web server runs is accessible from your computer.

```
http://192.168.1.2:8080/wt/servlet/HelloWorld
```

It generates the result shown in Figure 20.5.

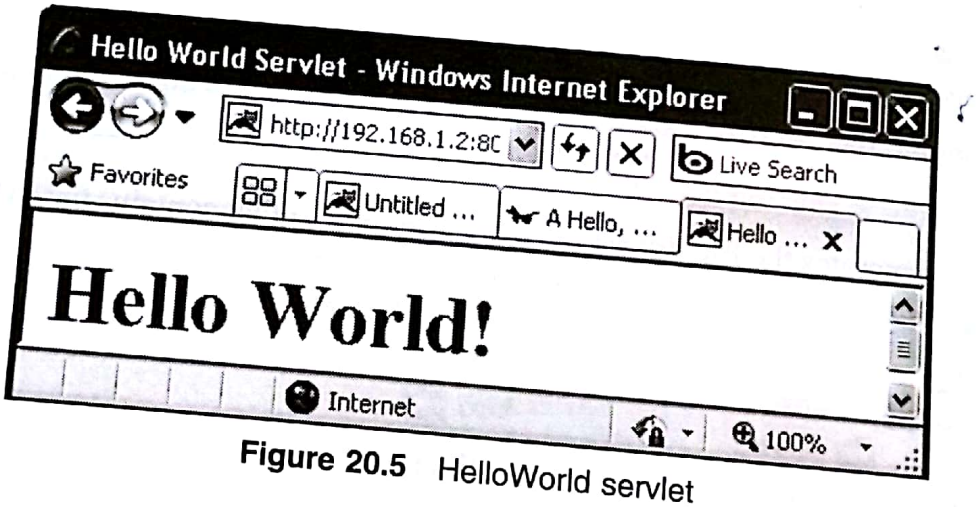


Figure 20.5 HelloWorld servlet

20.9 PASSING PARAMETERS TO SERVLETS

There are two ways in which you can pass parameters to a servlet: using URL and using HTML form.

Passing parameters directly to a Servlet

Parameters can be passed to servlets directly using URL. In this case parameters and their values are attached directly to the URL when you use the URL to call the servlet. The URL and parameters are separated by '?' character. The character ? says "here are the parameters". Each parameter takes the form `name=value`. Multiple parameters are separated by the '&' character Consider the following URL.

```
http://192.168.1.2:8080/servlet/check?login=abc&password=xyz
```

In this example, two parameters are passed `login` and `password` whose values are `abc` and `xyz`, respectively.

Certain special characters such as space and quote (") are not allowed directly in the URL. These characters are encoded in the `%hh` format where `hh` is the hex code of the character. For example, `%20`, `%22`, and `%7E` represent the space, quote, and tilde (~) characters, respectively.

Passing parameters directly to a Servlet

One of the disadvantages of passing parameters using URLs is that users can view the parameter names with their values passed. So, it is not safe to send sensitive information using this method. Moreover, the number of parameters that we can pass is also limited. HTML forms can be used to pass parameters in a convenient way. The following HTML code can be used to pass the same information as the previous example.

```
<form method='post' action='http://192.168.1.2:8080/servlet/check'>
  Login:<input type='text' name='login'><br />
  Password:<input type='password' name='password'><br/>
  <input type='submit' value='Login'>
</form>
```

The POST method is used here. In this case, the values of all form fields are embedded in the body of the HTTP request message and hence cannot be viewed directly. However, servlets can retrieve them using the usual way.

20.10 RETRIEVING PARAMETERS

The three methods of the `ServletRequest` interface may be used to retrieve parameters. Their signatures are as follows:

```
public abstract java.lang.String getParameter(java.lang.String);
public abstract java.lang.String[] getParameterValues(java.lang.String);
public abstract java.util.Enumeration getParameterNames();
```

The `getParameter()` method returns the value of the specified parameter. Make sure that the parameter has exactly one value. If there are multiple parameters with the same name, use the `getParameterValues()` method, which returns an array of string values of the specified parameter.

The `getParameterNames()` method returns all the parameters as an enumeration. The values of the individual parameters can be obtained by iterating through the enumeration and using the `getParameter()` method for each of these parameters. The following example shows how to retrieve values of two parameters, `login` and `password`, and return back to the client.

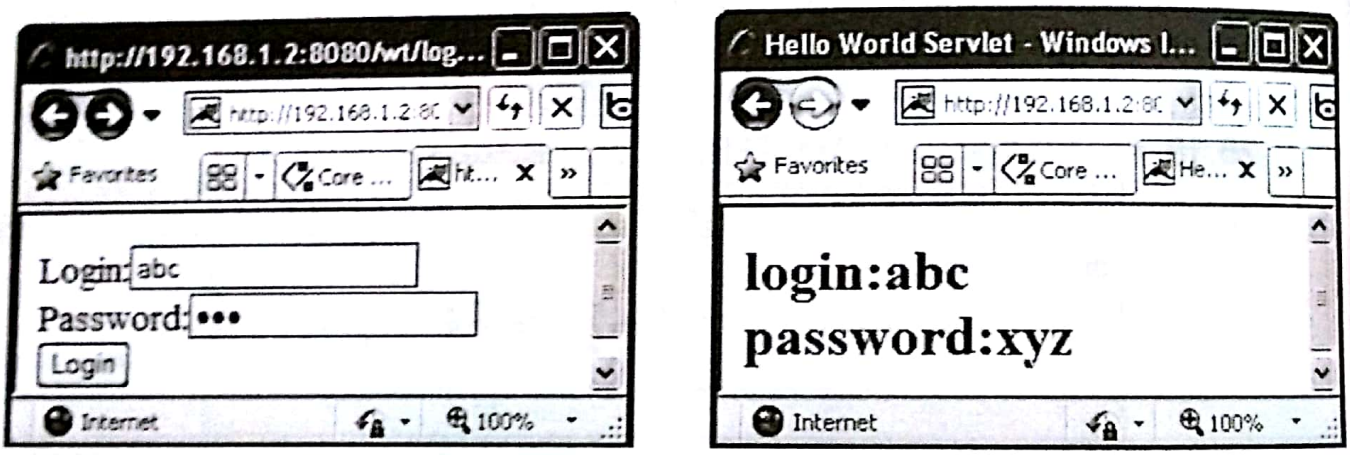
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetParameterServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Hello World Servlet</title></head>");
        out.println("<body>");
        String login = request.getParameter("login");
        String password = request.getParameter("password");
        out.println("<h2> login:" + login + "<br>password:" + password + "</h2>");
        out.println("</body></html>");
    }
}
```

Compile the servlet, make the following entry in the `web.xml` file, and restart the web server.

```
<servlet>
   <servlet-name>GetParameter</servlet-name>
   <servlet-class>GetParameterServlet</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>GetParameter</servlet-name>
   <url-pattern>/servlet/check</url-pattern>
</servlet-mapping>
```

Now, call this servlet from the form described before. The form output is shown in Figure 20.6 (i), while the response from the servlet is shown in Figure 20.6 (ii).



(i) (ii)

Figure 20.6 (i) Passing parameter (ii) Retrieving parameter

In practice, servlets hardly return the parameter with their values back to the client. Typically, they consult the database for further processing. In our case, the servlet will possibly verify the login name and password information against the information stored in the database and depending upon the result, either allow or disallow the user to login.

20.11 SERVER-SIDE INCLUDE

Server-Side Include (SSI) allows us to embed (include) servlets within HTML pages. The HTML files containing servlets typically have the extension `.shtml`. The web server understands this extension and forwards it to an internal servlet. This internal servlet parses the HTML document and executes the referenced servlets and sends the result back to the web server after merging it

with the original HTML document. This lets us include dynamically generated content in an existing HTML file, without having to serve the entire page via a CGI program or other dynamic technologies.

Tomcat uses `org.apache.catalina.ssi.SSIServlet` as its internal servlet. To enable SSI support, remove the XML comments from around the SSI servlet and servlet-mapping configuration in `$TOMCAT_HOME/conf/web.xml`.

The syntax for including servlets in HTML documents varies widely from web server to web server. Tomcat SSI directives have the following syntax:

```
<!--#directive attribute="value" attribute="value" ... -->
```

Tomcat SSI directives are formatted like an HTML comment. So, if SSI is not correctly configured, the browser will ignore it. Otherwise, the directive will be replaced with its results.

echo

It just inserts the value of a variable. For example, the following directive inserts the value of the local date.

```
<!--#echo var="DATE_LOCAL" -->
```

This generates the following result:

```
Friday, 20-Nov-2009 20:50:28 IST
```

There are a number of standard variables, including all the environment variables that are available to servlets. Additionally, you can define your own variables with the `set` directive.

printenv

It returns the list of all defined variables.

```
<!--#printenv -->
```

It results in the following:

```
HTTP_USER_AGENT=Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
HTTP_ACCEPT_LANGUAGE=en-us HTTP_ACCEPT=*/* LAST_MODIFIED=11/20/09
DOCUMENT_URI=/wt/SSI.shtml HTTP_ACCEPT_ENCODING=gzip, deflate REMOTE_PORT=1482
SERVER_NAME=192.168.1.2 SERVER_SOFTWARE=Apache Tomcat/6.0.16 Java HotSpot(TM)
Client VM/11.0-b12 Windows XP SCRIPT_FILENAME=D:\ApacheTomcat6.0.16 on
Thinkpad\webapps\wt\SSI.shtml DATE_LOCAL=11/20/09 SERVER_ADDR=192.168.1.2
SERVER_PROTOCOL=HTTP/1.1 REQUEST_METHOD=GET DOCUMENT_NAME=SSI.shtml
SERVER_PORT=8080 SCRIPT_NAME=/SSI.shtml REMOTE_ADDR=192.168.1.2
DATE_GMT=11/20/09 REMOTE_HOST=192.168.1.2 HTTP_HOST=192.168.1.2:8080
HTTP_CONNECTION=Keep-Alive UNIQUE_ID=2BD24DA4115256FAA76B7E55168AFC28
QUERY_STRING= GATEWAY_INTERFACE=CGI/1.1 org.apache.catalina.ssi.SSIServlet=true
REQUEST_URI=/wt/SSI.shtml
```

config

We can use the `config` directive, with a `timefmt` attribute, to modify the formatting of the date and time as follows:

```
<!--#config timefmt="%A %B %d, %Y" -->
<!--#echo var="DATE_LOCAL" -->
```

This generates the following result:

```
Friday November 20, 2009
```

fsize

This directive returns the size of the file specified by the `file` attribute.

```
<!--#fsize file="SSI.shtml" -->
```

This directive returns the size of the file `SSI.shtml` in kilobytes.

flastmod

The last modification time of a file is inserted using `flastmod` directive whose `file` attribute specifies the file name. The following code inserts the last modification time of the file named `SSI.shtml`.

```
<!--#flastmod file="SSI.shtml" -->
```

You need to replace the `SSI.shtml` with the actual file name you want to refer to. Note that this piece of code does not give the last modification time of an arbitrary file. It is better to use the `echo` directive to print the value of the `LAST_MODIFIED` environment variable, which specifies the last modification time of the file where this directive is included.

```
<!--#echo var="LAST_MODIFIED" -->
```

include

The most common use of SSI is to include the result of a server-side program such as servlet or CGI. The following directive includes the result of our `HelloWorld` servlet in the place of the directive.

```
<!--#include virtual="servlet/HelloWorld" -->
```

The attribute `virtual` specifies the server-side script to be used. Note that the servlet URL used here is relative to the `.shtml` file containing this directive.

So, create a file `SSI.shtml` containing a single line as follows and put in the `$TOMCAT_HOME\webapps\wt` directory.

```
<!--#include virtual="servlet/HelloWorld" -->
```

It generates the same output as our `HelloWorld` servlet, as shown in Figure 20.7.

Figure 20.7 Using SSI

The `include` directive is useful if you want to insert the result of the same servlet in different pages. Use the following to insert a header in every page.

```
<!--#include virtual="/header.html" -->
```

Note that the `include` directive can insert the result of any type of file.

set

It is used for creating variables and setting their values for later use. The following directive creates a variable `SSI` with the value `"Server Side Include"`.

```
<!--#set var="SSI" value="Server Side Include" -->
```

Similarly, the following code creates a variable `sum` with the value `"0"`.

```
<!--#set var="sum" value="0" -->
```

20.12 COOKIES

HTTP is stateless. This means that every HTTP request is different from others. Sometimes, it is necessary to keep track of a sequence of related requests sent by a client to perform some designated task. This is called *session tracking*.

Cookies are one of the solutions to session tracking. A *cookie* is key–value pair created by the server and is installed in the client's browser when the client makes a request for the first time. Browsers also maintain a list of cookies installed in them and send them to the server as a part of subsequent HTTP requests. The server can then easily identify that this request is a part of a sequence of related requests. This way, cookies provide an elegant solution to session tracking.

The servlet API supports cookies. A cookie is represented using the `javax.servlet.http.Cookie` class and is created using the following constructor:

```
Cookie(String key, String value)
```

A cookie is added by the `addCookie()` method of the `HttpServletResponse` class. Similarly, the server can get all cookies sent by the web browser using the `getCookies()` method of the `HttpServletRequest` class.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieDemo extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {
      PrintWriter out = response.getWriter();
      Cookie[] cookies = request.getCookies();
      boolean found = false;
      if (cookies != null) {
         for (int i = 0; i < cookies.length; i++) {
            if (cookies[i].getName().equals("session_started")) {
               found = true;
               out.print("You started this session on : " );
               out.println(cookies[i].getValue());        }
         }
      }
```

```
            if (!found) {
                String dt = (new java.util.Date()).toString();
                response.addCookie(new Cookie("session_started", dt));
                out.println("Welcome to out site...");
            }
        }
    }
```

When a client makes the call for the first time, the web browser does not send any cookie. The servlet looks for the cookie with the name `"session_started"`. Naturally, it cannot find the cookie. It creates, installs, and sends a cookie with the name `"session_started"` using the following code.

```
response.addCookie(new Cookie("session_started", dt));
```

For subsequent calls, web browsers send this cookie with a request. Our servlet finds it, and retrieves the start time of this session and finally sends this value. So, when you access this servlet for the first time, you will see an output as shown in Figure 20.8 (i). Any subsequent request generates the output shown in Figure 20.8 (ii).

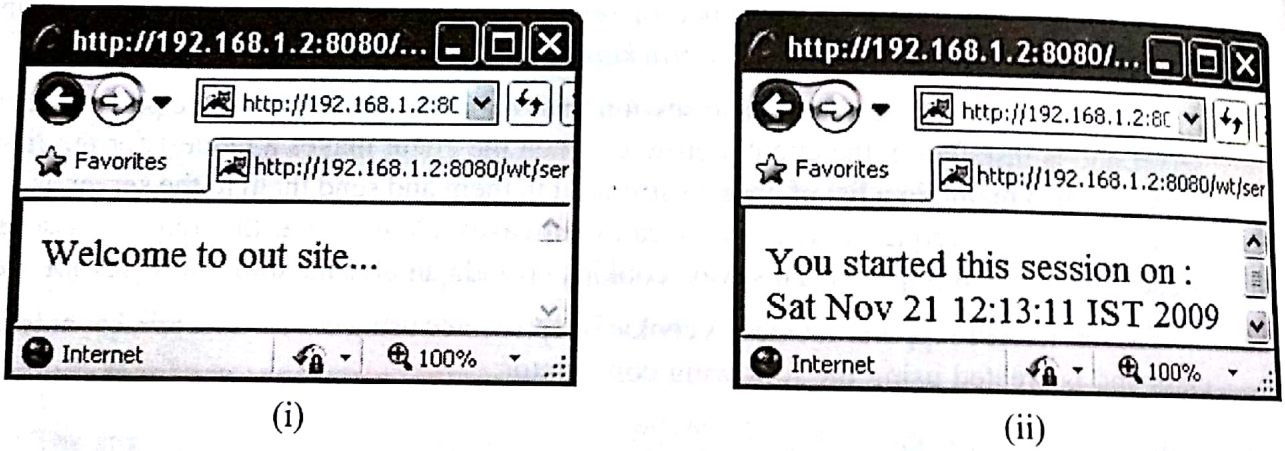


Figure 20.8 (i) First access (ii) Subsequent access

20.12.1 Limitations of Cookies

Cookies work correctly, provided that the web browsers have enabled cookie support. In addition to the security concerns, there are some technical limitations:

- Cookies can carry small pieces of information and are not a standard way of communication.
- Some web browsers limit the number of cookies (typically 20 per web server) that can be installed. To avoid this problem, more than one block of information may be sent per cookie.
- The value of a cookie should never exceed 4 KB. If the value of a cookie is larger than 4 KB, it should be trimmed to fit.
- Cookies cannot identify a particular user. A user can be identified by a combination of user account, browser, and computer. So, users who have multiple accounts and use multiple computers/browsers have multiple sets of cookies. Consequently, cookies cannot differentiate between multiple browsers running in a single computer.
- Intruders can snoop, steal cookies, and attack sessions. This is called *session hijacking*.

20.13 FILTERS

Filters are objects that are installed between the client and the server to inspect requests and responses. They can transform the request or modify the response. Note that filters are not servlets and hence cannot create actual responses. Filters process requests before they reach a servlet and/or process responses after leaving a servlet. In general, a filter can do the following:

- Intercept and inspect requests before dispatching them to the servlets
- Modify requests' headers and data and discard or filter requests
- Intercept and inspect responses before dispatching them to the clients
- Modify requests' headers and data and discard or filter responses

A filter can work on behalf of a single servlet or a group of servlets. Similarly, zero or more servlets can be installed for a servlet.

A filter class must implement the `javax.servlet.Filter` interface, which provides a framework for the filtering mechanism. It defines the following method to be implemented by each filter class:

```
void init(FilterConfig config)
```

The web container calls this method once to install it and to set its configuration object. The filter can then start servicing. The `FilterConfig` interface defines methods to retrieve the filter's name, its `init` parameters and the underlying servlet context.

```
void defiler(ServletRequest request, ServletResponse response, FilterChain chain)
```

This method is called before dispatching the request to the servlet. It receives the current request as well as the `FilterChain` containing filters that will handle this request after it. In this method, the filter should process the request and take necessary actions. The filter should then call the `chain.doFilter()` method to handover the control to the next filter in this chain. Whenever the response comes back, this filter can do additional tasks on the response.

If the filter decides, it may not forward the request to the next filter; in this case, the request will not reach the servlet at all. In this way, filters can drop malicious requests intended for the servlet.

```
void destroy()
```

The method is called to uninstall the filter.

The following filter forwards only those requests that come from the host having the IP address "192.168.1.2". Requests from all other hosts are discarded.

```
import java.io.*;
import javax.servlet.*;
public class Firewall implements Filter {
   private FilterConfig config = null;
   public void init(FilterConfig config) throws ServletException {
      this.config = config;
   }
   public void doFilter(ServletRequest request, ServletResponse response,
      FilterChain chain) throws IOException, ServletException {
```

```
            if (request.getRemoteAddr().equals("192.168.1.2"))
               chain.doFilter(request, response);
        }
        public void destroy() {
            config = null;
        }
    }
```

20.13.1 Deploying Filter

To use this filter, you must specify it in the `web.xml` file using the `<filter>` tag.

```
<filter>
      <filter-name>myFirewall</filter-name>
      <filter-class>Firewall</filter-class>
</filter>
```

This notifies the server that a filter named `myFirewall` is implemented in the `Firewall` class. Now, add the following lines in the `web.xml` file.

```
<filter-mapping>
      <filter-name>myFirewall</filter-name>
      <url-pattern>/HelloWorld</url-pattern>
</filter-mapping>
```

This code states that a filter has to be installed for the URL pattern `/HelloWorld` that corresponds to our `HelloWorld` servlet. Now, try accessing the `HelloWorld` servlet from the host 192.168.1.2. A response will be received as before. But, if the servlet is accessed from any machine other than 192.168.1.2, no response will be received.

20.14 PROBLEMS WITH SERVLET

Servlets are not useful for generating presentation content such as HTML. Consider the following piece of code.

```
out.println("<html>");
out.println(" <head>");
out.println("    <title>Hello World Servlet</title>");
out.println(" </head>");
out.println(" <body>");
out.println("    <h1>Hello World!</h1>");
out.println(" </body>");
out.println("</html>");
```

The purpose of the code is to generate the following HTML code.

```
<html>
   <head>
      <title>Hello World Servlet</title>
   </head>
   <body>
      <h1>Hello World!</h1>
   </body>
</html>
```

This is acceptable if you want to generate smaller HTML files. However, if the size of the HTML document is large, it is a tedious process to generate it.

Another drawback of the servlet over Java Server Pages (JSP) is that you have to recompile the source yourself after any modification is done. Moreover, the web server has to be restarted to get the effect of the modified code. However, JSP can perform these tasks automatically on our behalf.

Servlets often contain presentation logic as well as processing logic, which makes the code difficult to read, understand and extend. If the presentation logic changes, the servlet code has to be modified, recompiled, and redeployed.

20.15 SECURITY ISSUES

Different authors write servlets that are run using the resources of the server computer under the supervision of a web server. So, before installing servlets in the web server, make sure that they come from trusted sources. Certain access constraints should be imposed on the servlets depending on their sources.

The concept of *sandbox* may be incorporated. A *sandbox* is a container of servlets where restrictions are imposed. The administrator of the web server decides which servlets are given which permissions, so that they cannot compromise with the system's security.

In addition to these security issues, the author of the servlet should consider the following points:

- Take sufficient care while writing the file upload code. If not implanted carefully, users may fill the hard disk of the server by uploading large files.
- Review the code that accesses files/database based on the user input. For example, do not allow users to execute arbitrary SQL commands. If allowed, users may fire some harmful SQL commands that can delete database tables.
- Make sure that the request comes from an authorized user. Do not rely on the existence of a session variable.
- Make sure that you have not used the `System.exit()` method anywhere in your program. This will terminate the web server.
- Do not display sensitive parameter values in the web page.

KEYWORDS

Cookies: A cookie is a key–value pair used for session tracking.

Filters: Filters are objects that are installed between the client and the server to inspect requests and responses.

GenericServlet: The class `GenericServlet` implements the Servlet and ServletConfig interfaces and defines a generic protocol-independent servlet.

HTTP tunneling: A procedure for using other protocols through the HTTP protocol.

HttpServletRequest: An interface that represents an HTTP request from the client.

HttpServletResponse: An interface that represents an HTTP response from the servlet.

HttpServlet: The `HttpServlet` class extends `GenericServlet` and provides a framework for handling HTTP requests.

Life Cycle methods: The methods such as `init()`, `service()`, `destroy()` that run during the life cycle of a servlet.

Multi-threaded servlet: A servlet that can handle multiple simultaneous requests.

Server Side Include: Server-Side Include (SSI) allows us to embed (include) servlets within HTML pages, using SSI directives.

Servlet Chain: A sequence of servlets called in that order to accomplish some specific task.

Servlet Interface: Top-level interface of the servlet architecture, which provides the basic functionalities of all servlets that are created by implementing this interface directly or indirectly.

Servlet life cycle: The set of states that a servlet goes through, from creation to destruction.

ServletConfig Interface: Represents the configuration of a servlet.

ServletRequest: An interface that represents a request from the client.

ServletResponse: An interface that represents a response from the servlet.

Session Tracking: Keeping track of a sequence of requests sent by a client to perform some designated task.

Single-threaded servlet: A servlet that can service one request at a time.

Web container: The environment/engine where servlets or other server-side programs run.

SUMMARY

Servlet is a Java technology for server-side programming. A servlet is a Java module that runs inside a Java-enabled web server and services requests obtained from a web server.

Java servlet technology has some distinct advantages over other competent technologies in terms of efficiency, persistency, portability, robustness, extensibility, etc. Servlets run in the server machine and usually generate simple HTML codes. So, even a very old browser can display generated HTML pages correctly.

The basic servlet architecture consists of two packages: `javax.servlet` and `javax.servlet.http`. The first package contains top-level interfaces and classes that are used and extended by all other servlets. The second package is provided to use for servlets that can handle HTTP requests.

The container in which the servlet is deployed, supervises and controls the life cycle of a servlet. During its lifetime, methods such as `init()`, `service()`, and `destroy()` are called in some specific order.

The `GenericServlet` class defines a generic protocol-independent servlet in the sense that it can be extended to provide implementation of any protocol such as HTTP, FTP, and SMTP. On the other hand, `HttpServlet` class extends GenericServlet and provides a framework for handling HTTP requests.

There are two ways in which you can pass parameters to a servlet: using URL and using HTML form. One of the disadvantages of passing parameters using URLs is that users can view the parameter names with their values passed. So, it is not safe to send sensitive information using this method. Moreover, the number of parameters that we can pass is also limited. HTML forms can be used to pass parameters in a convenient way. The methods of the ServletRequest interface are used to retrieve parameters.

Server-Side Include (SSI) allows us to embed (include) servlets within HTML pages, using SSI directives. This lets us include dynamically generated content in an existing HTML file, without having to serve the entire page via a CGI program or other dynamic technologies.

Sometimes, it is necessary to keep track of a sequence of related requests sent by a client to perform some designated task. This is called session tracking. Cookies are one of the solutions to session tracking.

Filters are objects that are installed between clients and servers to inspect requests and responses. They can transform request or modify responses.

WEB RESOURCES

`http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html`
Java Servlet Technology

EXERCISES

Multiple Choice Questions

1. Which one of the following methods is called first on a servlet?
   (a) start() (b) initialize()
   (c) init() (d) doInit()

2. Which of the following is the name of a web server?
   (a) Netscape (b) Borland
   (c) W3C (d) Tomcat

3. Which is the top-level class/interface in the servlet class hierarchy?
   (a) javax.servlet.Servlet
   (b) javax.servlet.HttpServlet
   (c) javax.servlet.GenericServlet
   (d) javax.servlet.TopServlet

4. Which of the following statements is false regarding servlets?
   (a) The init() function is called only once during the instantiation of the servlet
   (b) The HttpServlet class implements the Servlet interface
   (c) The HttpServlet can handle HTTP requests
   (d) The GenericServlet class implements the Servlet interface

5. A servlet is instantiated when
   (a) a client makes a request for the first time.
   (b) the web server starts up
   (c) the client makes a request
   (d) the source code is compiled

6. Which of the following interfaces a single threaded servlet must implement?
   (a) NoMultiThread
   (b) SingleThread
   (c) OneThreadModel
   (d) SingleThreadModel

7. How many arguments does the service() method take?
   (a) 1 (b) 2 (c) 3 (d) 4

8. Which of the following methods is used to retrieve the value of a specified parameter?
   (a) getParameter() (b) retrieveParameter()
   (c) parameter() (d) getParam()

9. Which of the following tags is used to make a named servlet?
   (a) <servlet-name> (b) <servlet-class>
   (c) <servlet> (d) <servlet-mapping>

10. Which of the following tags maps a named servlet to a URL pattern?
    (a) <servlet> (b) <servlet-name>
    (c) <servlet-class> (d) <servlet-mapping>

11. Which of the following SSI directives displays the value of a variable?
    (a) display (b) print
    (c) echo (d) show

12. What is the full form of SSI?
    (a) Server Side Include
    (b) Server Side Integration
    (c) Same Side Include
    (d) Same Side Integration

13. Which of the following SSI directive is used insert the content of a server-side program?
    (a) insert (b) include
    (c) paste (d) import

14. Which of the following methods is used to add a cookie to the response?
    (a) addCookie() (b) sendCookie()
    (c) installCookie() (d) insertCookie()

15. What is the name of the servlet configuration file?
    (a) servlet.xml (b) config.xml
    (c) web.xml (d) server.xml

16. Which of the following packages contains Servlet interface?
    (a) javax.server.servlet (b) javax.servlet
    (c) java.server.servlet (d) java.servlet

17. The servlet initialization parameters are specified in the _____ deployment descriptor file as part of a servlet element.
    (a) web.xml (b) web.doc
    (c) web.txt (d) web.ini

18. Which of the following identifies the correct method a servlet developer should use to retrieve data provided by the client?
    (a) getParameter() against the HttpServletRequest object
    (b) getInputStream() against the HttpServletRequest object
    (c) getBytes() against the HttpServletRequest object
    (d) getQueryString() against the HttpServletRequest object

19. Which package provides interfaces and classes for writing servlets?
    (a) javax.servlet (b) javax.java.servlet
    (c) javax.awt.servlet (d) javax.swing.servlet

20. The Web server that executes the servlet creates an _____ object and passes this to the servlet's service method (which, in turn, passes it to doGet or doPost).
    (a) HttpRequest (b) HttpResponse
    (c) Request (d) HttpServletResponse

21. Which method can be used to access the ServletConfig object?
    (a) getServletInfo()
    (b) getServletConfig()
    (c) getInitParameters()
    (d) getConfig()

Review Questions

1. What is a servlet?
2. What are the advantages of servlets over CGI?
3. What is the task of the javax.servlet.Servlet interface?
4. What are the two objects that a servlet receives when it accepts a call from a client?
5. What information does ServletRequest allow access to?
6. What type of constraints can ServletResponse interface set on the client?
7. Describe the life cycle of a servlet.
8. Describe how an HTTP Servlet handles its client requests.
9. Differentiate between the single-threaded and multi-threaded servlet model.
10. What are the differences between servlets and applets?
11. Mention some uses of Servlets.
12. How can you invoke other web resources such as servlets /JSPs?
13. How can you include other Resources in the Response?
14. What information does the ServletResponse interface give to the servlet methods for replying to the client?
15. What do you understand by servlet mapping?
16. What are the advantages of Cookies over URL rewriting?
17. What is session hijacking?

21 JAVA SERVER PAGES (JSP)

KEY OBJECTIVES

After completing this chapter readers will be able to—

- understand the advantages of JSP technology over other competitive technologies
- learn the architecture of JSP pages and their execution philosophy
- get an idea about the different components of JSP pages
- get an overview about the methods used to track sessions
- create and use JavaBean components
- create and use custom tags
- learn how to use JDBC technology

21.1 INTRODUCTION AND MARKETPLACE

Java **S**erver **P**ages (JSP) is a server-side technology that enables web programmers to generate web pages dynamically in response to client requests. There are many server-side technologies for building dynamic web applications. However, JSP is the one that has pulled the attention of the web developers. There are several reasons for it.

JSP is nothing but high-level abstraction of Java servlet technology. It allows us to directly embed pure Java code in an HTML page. JSP pages run under the supervision of an environment called *web container*. The web container compiles the page on the server and generates a servlet, which is loaded in the Java Runtime Environment (JRE). The servlet serves the client requests in the usual way. This makes the development of web applications convenient, simple, and fast. Maintenance also becomes very easy.

JSP technology provides excellent server-side programming for web applications that need database access. JSP not only provides cross-web-server and cross-platform support, but also integrates the WYSIWYG (What You See Is What You Get) features of static HTML pages with the extreme power of Java technology.

JSP also allows us to separate the dynamic content of our web pages from the static HTML content. We can write regular HTML files in the usual way. We can then insert Java code for the dynamic parts using special tags, which usually start with <% and end with %>.

Usually, JSP files have the extension .jsp. They are also installed in a place where we can place our normal web pages. Many web servers let us define aliases for JSP pages or servlets. So, a URL that appears to reference an HTML file may actually point to a JSP page or servlet.

21.2 JSP AND HTTP

Java Server Pages specification extends the idea of Java servlet API, to provide a robust framework to developers of web applications for creating dynamic web content. Currently, JSP or servlet technology supports only HTTP. However, a developer may extend the idea of servlet or JSP to implement other protocols such as FTP or SMTP. Since JSP uses HTML, XML, and Java code, the applications are secure, fast, and independent of server platforms. It allows us to embed pure Java code in an HTML document. It is important to note that JSP specification has been defined on top of the Java servlet API. Consequently, it follows all servlet semantics and has all powers that the servlet has.

The life cycle and many of the capabilities, especially the dynamic aspects of JSP pages, are exactly the same as Java servlet technology. So, much of the discussion in this chapter refers to Chapter 20.

21.3 JSP ENGINES

To process JSP pages, a JSP engine is needed. It is interesting to note that what we know as the "JSP engine" is nothing but a specialized servlet, which runs under the supervision of the servlet engine. This JSP engine is typically connected with a web server or can be integrated inside a web server or an application server. Many such servers are freely available and can be downloaded for evaluation and/or development of web applications. Some of them are Tomcat, Java Web Server, WebLogic, and WebSphere.

Once you have downloaded and installed a JSP-capable web-server or application server in your machine, you need to know where to place the JSP files and how to access them from the web browser using the URL. We shall use the Tomcat JSP engine from Apache to test our JSP pages.

21.3.1 Tomcat

Tomcat implements servlet 2.2 and JSP 1.1 specifications. It is very easy to install and can be used as a small stand-alone server for developing and testing servlets and JSP pages. For large applications, it is integrated into the Apache Web server.

In this book, we shall use tomcat JSP engine to test our JSP pages. Following is a brief description of how to install Tomcat in your machine.

- Download the appropriate version of Tomcat from the Apache site `http://tomcat.apache.org/`
- Uncompress the file in a directory. If it is an installer file in Windows, install it in a directory by double clicking on the installer file.
- Set the environment variable JAVA_HOME to point to the root directory of your JDK hierarchy. For example, if the root directory of your JDK is `D:\Java\jdk1.6.0_10`, the JAVA_HOME environment variable should point to this directory. Also make sure that the directory (usually bin) containing the Java compiler and interpreter is in your PATH environment variable.
- Go to the bin directory under the tomcat installation directory and start Tomcat using the command-line command `startup.bat` (in Windows) or `startup.sh` (in Unix/Linux). If everything goes well, you will see an output in Windows as shown in Figure 21.1

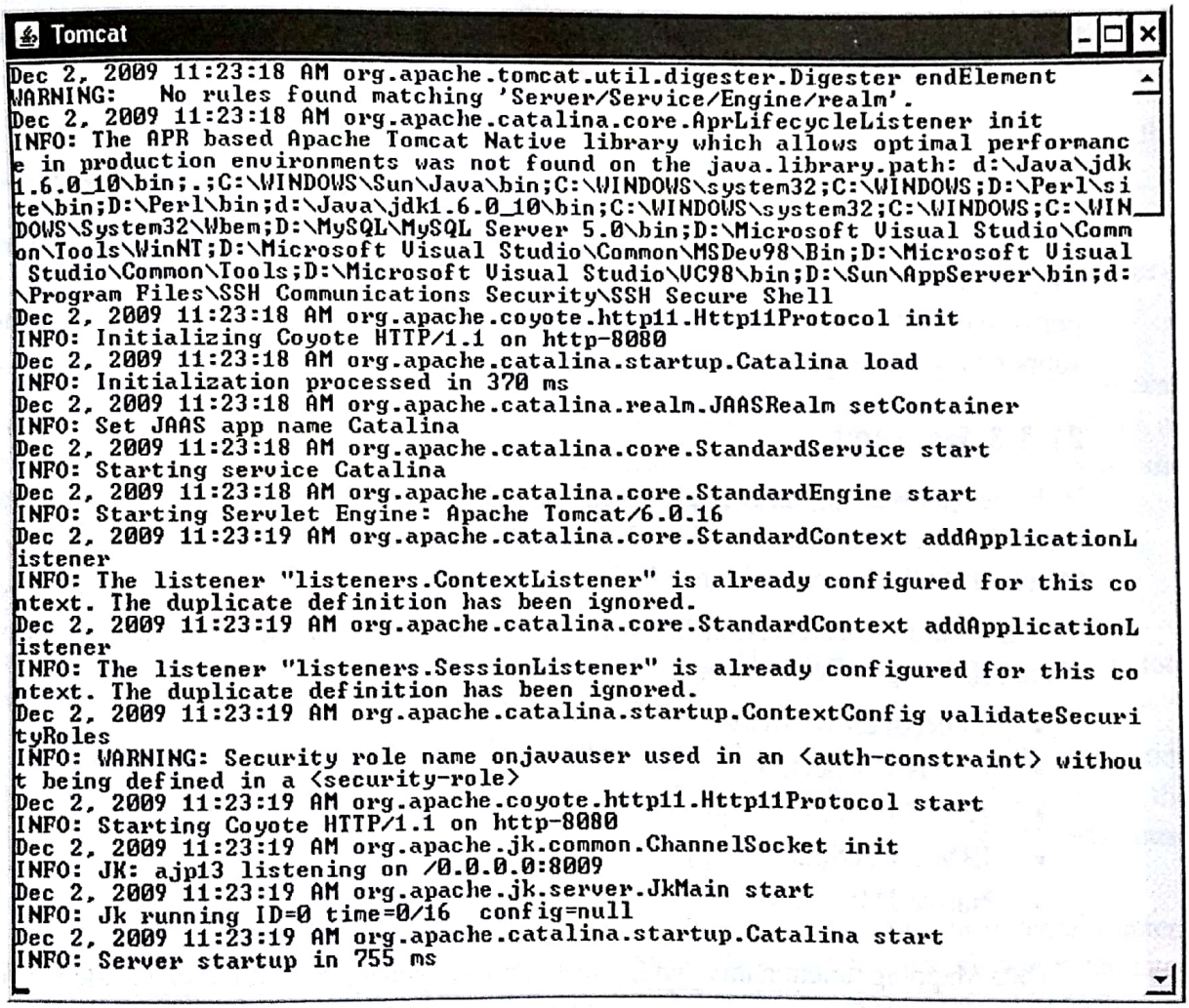


Figure 21.1 Starting tomcat server

- Tomcat is now running on port 8080 by default. You can test it by using the URL `http://127.0.0.1:8080/` . The page shown in Figure 21.2 appears.

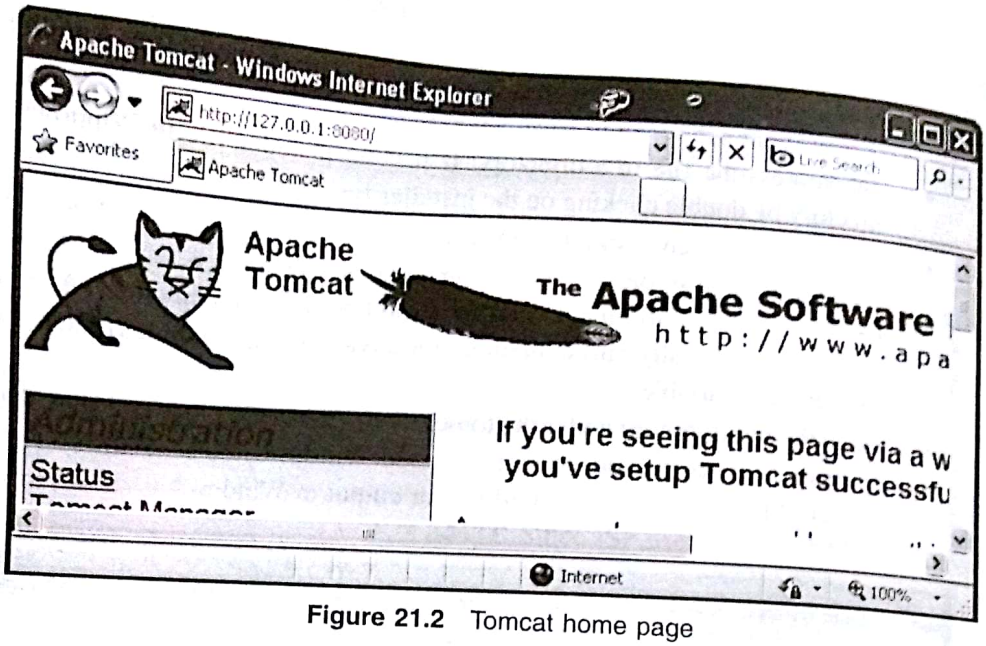


Figure 21.2 Tomcat home page

21.3.2 Java Web Server

The Java System Web Server by Sun is a leading web server that delivers secure infrastructure for medium and large business technologies and applications. Sun claims that it delivers 8x better performance than Apache 2.0 with Tomcat. It is available on all major operating systems and supports a wide range of technologies such as JSP and Java Servlet technologies, PHP, and CGI.

21.3.3 WebLogic

WebLogic by BEA Systems of San Jose, California, is a J2EE application server and also an HTTP server for Microsoft Windows, Unix/Linux, and other platforms. WebLogic supports DB2, Oracle, Microsoft SQL Server, and other JDBC-compliant databases.

The WebLogic server also includes the .NET framework for interoperability and allows integration of the following native components:

- CORBA connectivity
- COM+ Connectivity
- J2EE Connector Architecture
- IBM's WebSphere MQ connectivity
- Native JMS messaging for enterprise

Data Mapping functionality and Business Process Management are also included in WebLogic Server Process Edition.

21.3.4 WebSphere

IBM attempted to develop a software to set up, operate, and integrate electronic business applications that can work across multiple computing platforms, using Java-based web technologies. The result

is WebSphere Application Server (WAS). It includes both the run-time components and the tools that can be used to develop robust and versatile applications that will run on WAS.

21.4 HOW JSP WORKS

When a client such as a web browser sends a request to a web server for a JSP file using a URL, the web server identifies the .jsp file extension in the URL and figures out that the requested resource is a Java Server Page. The web server hands over the request to a special servlet. This servlet checks whether the servlet corresponding to this JSP page exists or not. If the servlet does not exist, or exists but it is older than the JSP page, it performs the following:

- Translates the JSP source code into the servlet source code
- Compiles the servlet source code to generate a class file
- Loads the class file and creates an instance
- Initializes the servlet instance by calling the `jspInit()` method
- Invokes the `_jspService()` method, passing request and response objects

If the servlet exists and is not older than the corresponding JSP page requested, it does the following:

- If an instance is already running, it simply forwards the request to this instance.
- Otherwise, it loads the class file, creates an instance, initializes it, and forwards the request to this instance.

During development, one of the advantages of JSP pages over servlets is that the build process is automatically performed by the JSP engine. When a JSP file is requested for the first time, or if it is changed, the translation and compilation phase occurs. The subsequent requests for the JSP page directly go to the servlet byte code, which is already in the memory.

21.5 JSP AND SERVLET

Java servlet technology is an extremely powerful technology. However, when it is used to generate large, complex HTML code, it becomes a bit cumbersome.

In most servlets, a small piece of code is written to handle application logic and a large code is written using several `out.println()` statements that handle output formatting. Since the codes for application logic and formatting are closely tied, it is difficult to separate and reuse portions of the code when a different logic or output format is required.

JSP separates the static presentation templates from the logic, to generate the dynamic content by encapsulating it within external JavaBean components. These components are then instantiated and used in a JSP page using special tags and scriptlets. When the presentation template is changed by the web designer, the JSP engine recompiles the JSP page and reloads it in the Java Runtime Environment (JRE).

Another problem of servlets is that each time the servlet code is modified, it needs to be recompiled and the web server also needs to be restarted. The JSP engine takes care of all these issues automatically.

Whenever a JSP code is modified, the JSP engine identifies it and translates it into a new servlet. The servlet code is then compiled, loaded, and instantiated automatically. The servlet remains in the memory and serves requests subsequently without any delay.

JSP uses technologies based on reusable component engineering such as JavaBean component architecture and Enterprise JavaBean technology. So, JSP does not require significant developer expertise, like Java servlets. Consequently, in the application development life cycle, page designers can now play a major role. JSP pages can be moved easily across web servers and platforms, without any changes or significant effort.

For these reasons, web application developers turn towards JSP technology as an alternative to servlet technology.

21.5.1 Translation and Compilation

Every JSP page gets converted to a normal servlet behind the scene by the web container automatically. Figure 21.3 shows one such translation.

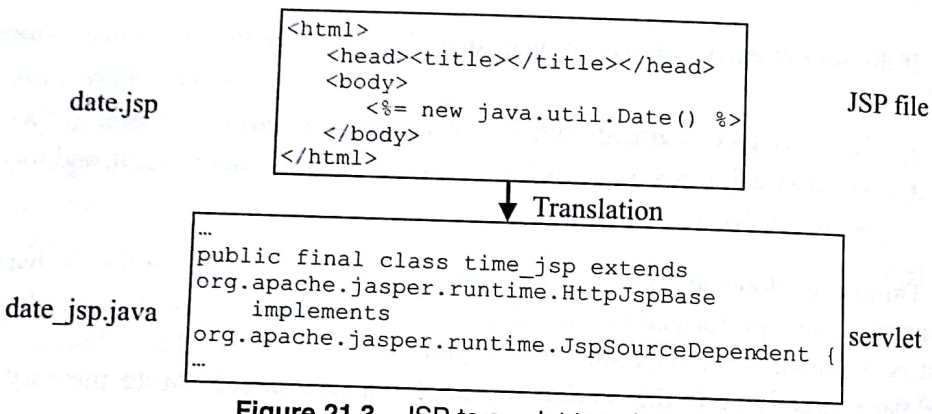


Figure 21.3 JSP to servlet translation

Each type of data in a JSP page is processed differently during the translation phase. The translation and compilation phases may result in errors. These errors (if any) are generated when a user requests the page for the first time.

The JSP engine returns a `ParseException`, if an error occurs during the translation phase. In such a case, the servlet source file will be empty or incomplete. The last incomplete line describes the cause of error in the JSP page. If an error occurs during the compilation phase, the JSP engine returns a `JasperException` and a message that describes the name of the JSP page's servlet and the line that caused the error.

Now, let us take a specific example to understand this translation procedure. Consider the following JSP page. You may not understand the syntax used in this file. We shall discuss it in the rest of this chapter.

```
<html>
  <head><title>Square table</title></head>
  <body>
    <table border="1">
```

```
            <caption>Temperature Conversion chart</caption>
            <tr><th>Celsius</th><th>Fahrenheit</th></tr>
        <%
            for(int c = 0; c <= 100; c+=20) {
               double f = (c*9)/5.0 + 32;
               out.println("<tr><td>" + c + "</td><td>" + f + "</td></tr>");
            }
        %>
        </table>
    </body>
</html>
```

This is a simple JSP page that prints temperature conversion (from Celsius to Fahrenheit) chart. If you open this page in the Internet Explorer, it looks as shown in Figure 21.4.

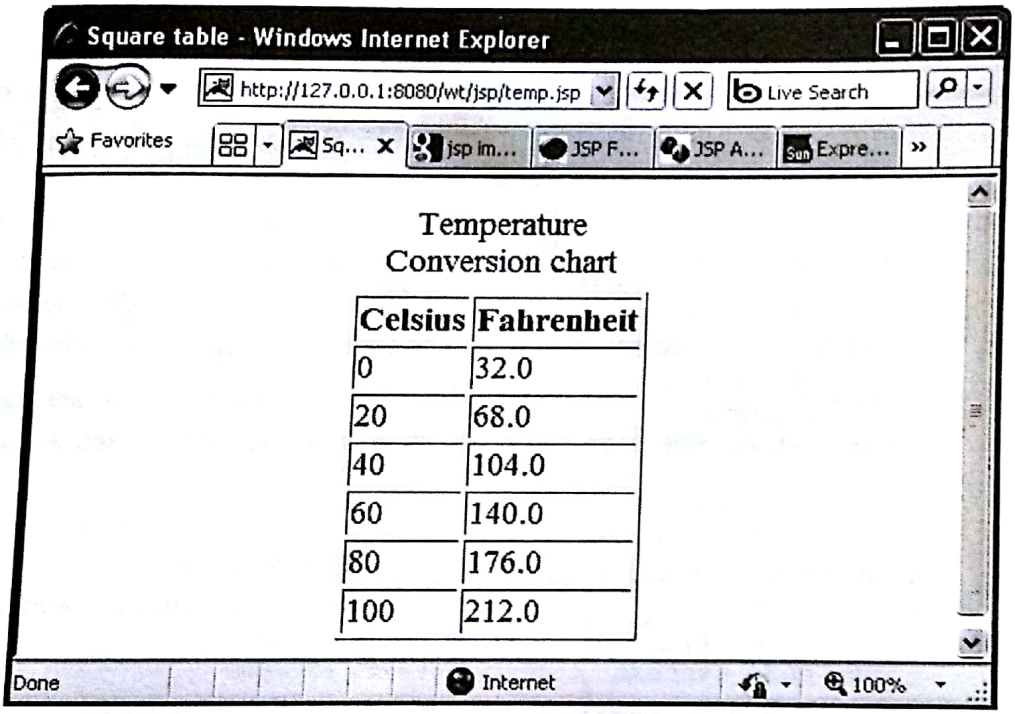


Figure 21.4 Temperature conversion chart

When you request this page for the first time, the JSP engine translates the JSP page into the following servlet source code.

```
package org.apache.jsp.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class temp_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

  private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();

  private static java.util.List _jspx_dependants;
```

```
    private javax.el.ExpressionFactory _el_expressionfactory;
    private org.apache.AnnotationProcessor _jsp_annotationprocessor;

    public Object getDependants() {
      return _jspx_dependants;
    }

    public void _jspInit() {
      _el_expressionfactory =
_jspxFactory.getJspApplicationContext(getServletConfig().getServletContext()).
getExpressionFactory();
      _jsp_annotationprocessor = (org.apache.AnnotationProcessor)
getServletConfig().getServletContext().getAttribute(org.apache.AnnotationProcessor.
class.getName());
    }

    public void _jspDestroy() {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {

      PageContext pageContext = null;
      HttpSession session = null;
      ServletContext application = null;
      ServletConfig config = null;
      JspWriter out = null;
      Object page = this;
      JspWriter _jspx_out = null;
      PageContext _jspx_page_context = null;


      try {
        response.setContentType("text/html");
        pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;

        out.write("<html>\r\n");
        out.write("\t<head><title>Square table</title></head>\r\n");
        out.write("\t<body>\r\n");
        out.write("\t\t<table border=\"1\">\r\n");
        out.write("\t\t\t<caption>Temperature Conversion chart</caption>\r\n");
        out.write("\t\t\t<tr><th>Celsius</th><th>Fahrenheit</th></tr>\r\n");
        out.write("\t\t");

          for(int c = 0; c <= 100; c+=20) {
            double f = (c*9)/5.0 + 32;
            out.println("<tr><td>" + c + "</td><td>" + f + "</td></tr>");
          }
```

```
            out.write("\r\n");
            out.write("\t\t</table>\r\n");
            out.write("\t</body>\r\n");
            out.write("</html>");
          } catch (Throwable t) {
            if (!(t instanceof SkipPageException)){
              out = _jspx_out;
              if (out != null && out.getBufferSize() != 0)
                try { out.clearBuffer(); } catch (java.io.IOException e) {}
              if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
            }
          } finally {
            _jspxFactory.releasePageContext(_jspx_page_context);
          }
        }
      }
```

The JSP page's servlet class extends `HttpJspBase`, which in turn implements the `Servlet` interface. In the generated servlet class, the methods `jspInit()`, `_jspService()`, and `jspDestroy()` correspond to the servlet life cycle methods `init()`, `service()`, and `destroy()`, respectively which were discussed in Chapter 20. The `_jspService()` method is responsible for serving client's requests. JSP specification prohibits the overriding of the `_jspService()` method. However, the developers are allowed to override the `jspInit()` and `jspDestroy()` methods within their JSP pages, to initialize and shut down servlets, respectively.

By default, the servlet container dispatches the `_jspService()` method using a separate thread to process concurrent client requests, as shown in Figure 21.5.

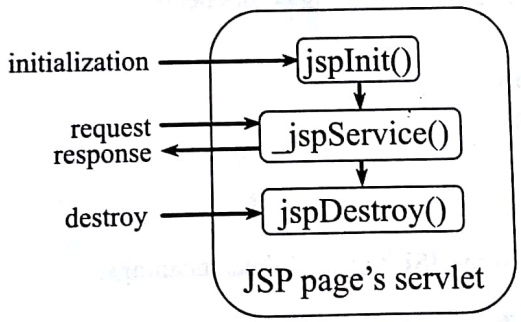


Figure 21.5 Life cycle of JSP's servlet

The static HTML part is simply printed to the output stream associated with the servlet's `service()` method. The code in <% and %> is copied in the `_jspService()` method.

21.6 ANATOMY OF A JSP PAGE

A JSP page basically consists of two parts: HTML/XML markups and JSP constructs. A large percent of your JSP page, in many cases, just consists of static HTML/XML components, known as *template text*. Consequently, we can create and maintain JSP pages using traditional HTML/XML tools.

We use three primary types of JSP constructs in a typical JSP page: *scripting elements*, *directives*, and *actions*.

Java code that will become an integral part of the resultant servlet is inserted using *scripting elements*. *Directives* let us control the overall structure and behavior of the generated servlet. *Actions* allow us to use existing components, and otherwise control the behavior of the JSP engine.

There are three kinds of scripting elements: *scriptlets*, *declarations*, and *expressions*. Scriptlets allow us to insert any Java-server-relevant API in the HTML or XML page provided that the syntax is correct. Declarations allow us to declare variable and methods. Expressions are used to print the value of a Java expression.

21.7 JSP SYNTAX

The syntax of JSP is almost similar to that of XML. ALL JSP tags must conform to the following general rules:

- Tags must have their matching end tags.
- Attributes must appear in the start tag.
- Attribute values in the tag must be quoted.

White spaces within the body text of a JSP page are preserved during the translation phase. To use special characters such as '%', add a '\' character before it. To use the '\' character, add another '\' character before it.

21.8 JSP COMPONENTS

A JSP page consists of the following components:

- Directives
- Declarations
- Expressions
- Scriptlets
- Actions

Table 21.1 lists some JSP tags and their meanings.

Table 21.1 JSP tags

JSP tag	Meaning
<%@...%>	Used for JSP directives such as page and include
<%=...%>	Used for JSP expressions
<%...%>	Used for JSP scriptlets that can contain arbitrary Java statements
<%!...%>	Used for variables, methods, and inner class declarations

21.8.1 Directives

A JSP page may contain instructions to be used by the JSP container to indicate how this page is interpreted and executed. Those instructions are called *directives*. Directives do not generate any

output directly, but tell the JSP engine how to handle the JSP page. They are enclosed within the `<%@` and `%>` tags. The commonly used directives are `page`, `include`, and `taglib`.

21.8.1.1 Page directive

Following is the syntax of the `page` directive:

```
<%@ page
    [ language="java" ]
    [ extends="package.class" ]
    [ import="{package.class | package.*}, ..." ]
    [ session="true | false" ]
    [ buffer="none | 8kb | sizekb" ]
    [ autoFlush="true | false" ]
    [ isThreadSafe="true | false" ]
    [ info="text" ]
    [ errorPage="relativeURL" ]
    [ contentType="MIMEType [ ;charset=characterSet ]" | "text/html ;
charset=ISO-8859-1" ]
          [ isErrorPage="true | false" ]
%>
```

The `page` directive has many attributes, which can be specified in any order. Multiple `page` directives may also be used, but in this case, except for the `import` attribute, no attribute should appear more than once. The code in boldface indicates the default values. Let us discuss the function of some attributes.

import

The value of this attribute is a list of fully qualified names of classes separated by commas (,), to be imported by the JSP file. To import all the classes of a package, use ".*" at the end of the package name. The following directive imports all the classes in the `java.io` and `java.reflect` packages and the class `Vector` that belongs to the `java.util` package.

```
<%@ page import="java.io.*, java.reflect.*, java.util.Vector" %>
```

The classes can then be referred from declarations, expressions, and scriptlets within the JSP document, without using the package prefix. You must import a class before using it. You can use `import` as many times as you wish in a JSP file. For example, the line of code we have just discussed can be written as three directives, as follows:

```
<%@ page import="java.io.*" %>
<%@ page import="java.reflect.*" %>
<%@ page import="java.util.Vector" %>
```

Those directives are converted to the corresponding import statement in the generated servlet file. You need not import the classes of the packages `java.lang`, `javax.servlet`, `javax.servlet.http`, and `javax.servlet.jsp`, as they are imported implicitly.

session

This attribute indicates whether the JSP file requires a HTTP session. The following syntax is used:

```
session="true | false"
```

If `true` is specified (this is the default value), the JSP file has a `session` object that refers to the current or new session. If the value is `false`, no `session` object is created.

If your JSP file does not require a session, the value should be set to `false` for performance consideration.

buffer

Syntax:

```
buffer="none | sizekb"
```

The `buffer` attribute indicates the size in kilobytes (default is 8kb) of the output buffer to be used by the JSP file. The value `none` indicates a buffer with zero size. In such a case, the output is written directly to the output stream of the response object of the servlet.

autoFlush

Syntax:

```
autoFlush="true | false"
```

The `autoFlush` attribute specifies whether the buffer should be flushed automatically (`true`) when it is full. If set to `false`, a buffer overflow exception is thrown when the buffer becomes full. The default value is `true`. The value of `autoFlush` can never be set to false when the buffer is set to none.

isThreadSafe

Syntax:

```
isThreadSafe="true | false"
```

This attribute indicates whether the JSP page can handle multiple threads simultaneously. If true (default value) is specified, the JSP container is allowed to send multiple concurrent client requests to this JSP page. In this case, we must ensure that the access to shared resources by multiple concurrent threads is effectively synchronized. If false is specified, the JSP container sends clients requests one by one using a single thread. This attribute basically provides information to the underlying servlet on whether it should implement the `SingleThreadModel` interface or not. Tomcat generates the following servlet class declaration if `false` is specified.

```
public final class ThreadDemo_jsp extends org.apache.jasper.runtime.HttpJspBase
```

```
public final class ThreadDemo_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent
```

info

Syntax:

```
info="text"
```

It allows us to specify a descriptive information about the JSP page. This information can be retrieved using the `Servlet.getServletInfo()` method.

contentType

Syntax:

```
contentType="MIMEType [ ;charset=characterSet ]"
```

It specifies the MIME type and encoding used in the generated response to be sent to the client. MIME types and encoding supported by the JSP container can only be specified. The default MIME type and character encoding are `text/html` and `ISO-8859-1`, respectively.

errorPage and isErrorPage

Syntax:

```
errorPage="relativeURL"
isErrorPage="true | false"
```

When an uncaught exception occurs in a JSP page, the JSP container sends information explaining the exception, which is undesirable in commercial sites. The `errorPage` attribute specifies the relative path name of another JSP page to be displayed, in case an error occurs in the current page. In the error page, the `isErrorPage` attribute must be set to `true`. The error page can access the implicit `exception` variable that represents the exception that has occurred as well as other implicit variables. It can then send some message understandable by the clients, by consulting those objects.

21.8.1.2 Include directive

This directive inserts the content of a file in a JSP page, during the translation phase when the JSP page is compiled.

```
<%@ include file="relativeURL" %>
```

If the file to be included is an HTML or text file, its content is directly included in the place of the `include` directive. The following example includes an HTML file `header.html` in a JSP page.

```
<%@ include file="header.html" %>
```

If the included file is a JSP file, it is first translated and then included in the JSP page.

```
<%@ include file="login.jsp" %>
```

Make sure that the tags in the included file do not conflict with the tags used in including the JSP page. For example, if the including JSP page contains tags such as `<html>` and `<body>`, the included file must not contain those tags.

The include process is said to be static as the file is included at compilation time. JSP 1.1 specifies that once the JSP file is compiled any changes in the included file will not be reflected. However, some JSP containers, such as tomcat, recompile the JSP page if the included file changes.

21.8.2 Comments

Comments are used to document the JSP pages and can be inserted anywhere in the JSP page. The general syntax is as follows:

```
<%-- JSP comment --%>
```

Anything between `<%--` and `--%>` is ignored by the JSP engine and is not even added to the servlet's source code. Here is an example:

```
<%-- Prints current date and time --%>
<%= new java.util.Date(); %>
```

The Java-like comments may also be used in scriptlets and declarations. They are added to the servlet, but not interpreted and hence are not sent to the client.

```
<%
   //get all cookies
   Cookie[] cookies = request.getCookies();
   /*
   Following code checks whether the request contains
   a cookie having name "user"
   */
   String user = null;
   for(int i = 0; i < cookies.length; i++)
      if(cookies[i].getName().equals("user"))
         user = (String)cookies[i].getValue();
   if(user != null) {
     //proceed
   }
%>
```

The HTML comments are enclosed in `<!--` and `-->` and added to the servlet source code as well as to the response, but are not displayed on the client's screen. Consider the following code:

```
<!-- This page was generated at server on
<%= (new java.util.Date()) %>
-->
```

This generates the following HTML comment. This is sent to the client, but not displayed on the screen.

```
<!-- This page was generated at server on
Sat Dec 05 13:17:45 IST 2009
-->
```

21.8.3 Expressions

JSP 2.0 specification includes a new feature, "Expression". It is used to insert usually a small piece of data in a JSP page, without using the `out.print()` or `out.write()` statements. It is a faster,

easier, and clearer way to display the values of variables/parameters/expressions in a JSP page. The general syntax of the JSP expression is as follows:

```
<%= expressions %>
```

The expression is embedded within the tag pair `<%=` and `%>`. Note that no semicolon is used at the end of the expression. The expression is evaluated, converted to a `String` and inserted in the place of the expression using an `out.print()` statement. For example, if we want to add 3 and 4 and display the result, we write the JSP expression as follows:

```
3 + 4 = <%= 3+4 %>
```

The expression is translated into the following Java statements in servlet source code.

```
out.write("3 + 4 = ");
out.print( 3+4 );
```

If everything works, you should see the following output:

```
3 + 4 = 7
```

The expression can be anything, as long as it can be converted to a string. For example, the following expression uses a `java.util.Date` object.

```
Date and time is : <%= new java.util.Date() %>
```

JSP expressions can be used as attribute values as well as tag names of JSP elements. Consider the following JSP code.

```
<%
   java.util.Date dt = new java.util.Date();
   String dateStr = dt.toString();
   String time = "currentTime";
%>
<<%= time %> value="<%= dateStr %>" />
```

This generates the following tag:

```
<currentTime value="Mon Nov 30 19:26:17 IST 2009" />
```

JSP also has the following XML equivalent for expression:

```
<jsp:expression>
   expressions
</jsp:expression>
```

21.8.4 Scriptlets

You can insert an arbitrary piece of Java code using the JSP *scriptlets* construct in a JSP page. This code will be inserted in the servlet's `_jspService()` method. Scriptlets are useful if you want to insert complex code which would otherwise be difficult using expressions. A scriptlet can contain any number of variables, class declarations, expressions, or other language statements. Scriptlets have the following form:

```
<% scriptlets %>
```

For example, the following scriptlet inserts the current date and time in the JSP page.

```
<%
   java.util.Date d = new java.util.Date();
   out.println("Date and time is : " + d);
%>
```

The resultant servlet code looks like this:

```
…
public final class scriptlet_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
…
  public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws java.io.IOException, ServletException {
…

   java.util.Date d = new java.util.Date();
   out.println("Date and time is : " + d);
…
```

This results in the following:

```
Date and time is : Mon Nov 30 17:01:09 IST 2009
```

JSP also has the following XML equivalent for scriptlet:

```
<jsp:scriptlet>
   scriptlets
</jsp:scriptlet>
```

21.8.4.1 Conditional processing

Several scriptlets and templates can be merged, to do some designated task. The following example illustrates this:

```
<%
int no = (int)(Math.random()*10);
if(no % 2 == 0) {
%>
Even
<% } else { %>
Odd
<% } %>
```

This code gets converted into something like

```
int no = (int)(Math.random()*10);
if(no % 2 == 0) {

     out.write("\r\n");
     out.write("Even\r\n");
} else {
     out.write("\r\n");
     out.write("Odd\r\n");
}
```

21.8.5 Declarations

JSP *declarations* are used to declare one or more variables, methods, or inner classes, which can be used later in the JSP page. The syntax is as follows:

```
<%! declarations %>
```

Examples are

```
<%! int sum = 0; %>
<%! int x, y , z; %>
<%! java.util.Hashtable table = new java.util.Hashtable(); %>
```

These variable declarations are inserted into the body of the servlet class, i.e., outside the `_jspService()` method that processes the client requests. So, variables declared in this way become instance variables of the underlying servlet.

```
<%! int sum = 0; %>
```

This is translated in the servlet as follows:

```
…
public final class test_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
  int sum = 0;
  …
  public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws java.io.IOException, ServletException {
…
```

Note that variables created using *scriptlets* are local to the `_jspService()` method. Instance variables are created and initialized once the servlet is instantiated. This is sometimes useful, but sometimes not desirable. The following declarations create an instance variable, `lastLoaded`.

```
<%! java.util.Date lastLoaded =  new java.util.Date();%>
The servlet was last loaded on <b><%=lastLoaded%></b>
```

This generates the following result:

```
The servlet was last loaded on Sun Nov 29 00:01:43 IST 2009
```

Similarly, the following piece of code displays the number of times the JSP page is referred after the page is loaded.

```
<%! int count = 0;%>
This page is referred <%= ++count %> times after last modification
```

Variables generated using declarations are available in scriptlets. A declaration has the scope of entire translation unit. It is valid in the JSP file as well as in all the files included statically. Declarations are not valid in dynamically included files.

JSP also has the following XML equivalent for declaration:

```
<jsp:declaration>
   declarations
</jsp:declaration>
```

21.8.6 Scope of JSP Objects

In a JSP page, objects may be created using directives, actions, or scriptlets. Every object created in a JSP page has a scope. The scope of a JSP object is defined as the availability of that object for use from a particular place of the web application. There are four object scope: *page*, *request*, *session*, and *application*.

page

Objects having *page* scope can be accessed only from within the same page where they were created. JSP implicit objects `out`, `exception`, `response`, `pageContext`, `config`, and `page` have 'page' scope. We have discussed about implicit objects in the next section. JSP objects created using the `<jsp:useBean>` tag also have page scope.

request

A request can be served by more than one page. Objects having *request* scope can be accessed from any page that serves that request. The implicit object `request` has the *request* scope.

session

Objects having *session* scope are accessible from pages that belong to the same session from where they were created. The `session` implicit object has the *session* scope.

application

JSP objects that have *application* scope can be accessed from any page that belong to the same application. An example is `application` implicit object.

21.8.7 Implicit Objects

Web container allows us to directly access many useful objects defined in the `_jspService()` method of the JSP page's underlying servlet. These objects are called *implicit objects* as they are instantiated automatically. The *implicit objects* contain information about request, response, session, configuration, etc. Some implicit objects are described in Table 21.2.

Table 21.2 JSP Implicit Objects

Variable	Class	Description
out	javax.servlet.jsp.JspWriter	Output stream of the JSP page's servlet.
request	Subtype of javax.servlet.ServletRequest	Current client request being handled by the JSP page
response	Subtype of javax.servlet.ServletResponse	Response generated by the JSP page to be returned to the client
config	javax.servlet.ServletConfig	Initialization information of the JSP page's servlet
session	javax.servlet.http.HttpSession	Session object for the client
application	javax.servlet.ServletContext	Context of the JSP page's servlet and other web components contained in the same application.

(Contd.)

(Contd.)

exception	java.lang.Throwable	Represents error. Accessible only from an error page
page	java.lang.Object	Refers to JSP page's servlet processing the current request
pageContext	javax.servlet.jsp.PageContext	The context of the JSP page that provides APIs to manage the various scoped attributes. It is extensively used by the tag handlers.

request

This object refers to the `javax.servlet.http.HttpServletRequest` type object that is passed to the `_jspService()` method of the generated servlet. It represents the HTTP request made by a client to this JSP page. It is used to retrieve information sent by the client such as parameters, (using the `getParameter()` method), HTTP request type (GET, POST, HEAD, etc), and HTTP request headers (cookies, referrer, etc). This implicit object has request scope.

```
<%
   String name =request.getParameter("name");
   out.println("Hello " + name);
%>
```

For the URL `http://127.0.0.1:8080/wt/jsp/getParameterDemo.jsp?name=Monali`, it displays the following:

```
Hello Monali
```

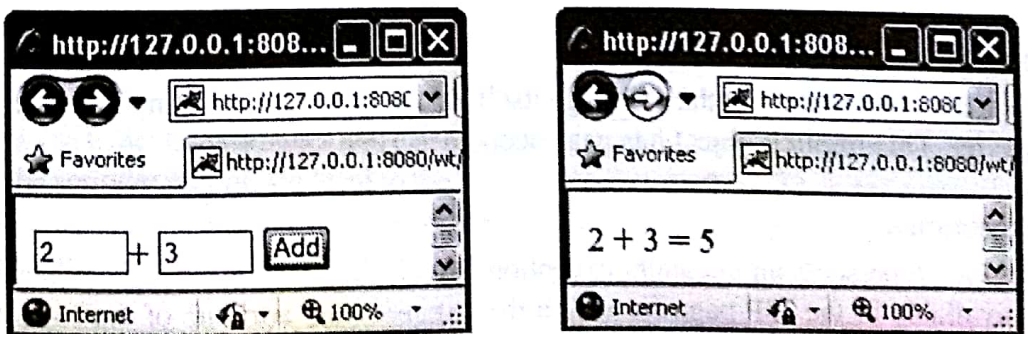
A JSP page can retrieve parameters sent through an HTML form. Consider the following form:

```
<form method='post' action='jsp/add.jsp'>
<input type='text' name='a' size='4'>+
<input type='text' name='b' size='4'>
<input type='submit' value='Add'>
</form>
```

Here is the code for the `add.jsp` file.

```
<%
   int a = Integer.parseInt(request.getParameter("a"));
   int b = Integer.parseInt(request.getParameter("b"));
   out.println(a + " + " + b + " = " + (a + b));
%>
```

The result is shown in Figure 21.6.



response

This object refers to the `javax.servlet.http.HttpServletResponse` type object that is passed to the `_jspService()` method of the generated servlet. It represents the HTTP response to the client. This object is used to send information such as HTTP response header and cookies. This implicit object has page scope.

pageContext

This `javax.servlet.jsp.PageContext` type object refers to the current JSP page context. It is used to access information related to this page such as request, response, session, application, and underlying servlet configuration. This implicit object has page scope.

session

This `javax.servlet.http.HttpSession` type object refers to the session (if any) used by the JSP page. Note that no `session` object is created if the `session` attribute of the `page` directive is turned off and any attempt to refer to this object causes an error. It is used to access session-related information such as creation time and ID associated to this session. This implicit object has session scope.

application

This `javax.servlet.ServletContext` object refers to the underlying application and is used to share data among all pages under this application. This implicit object has application scope.

out

It denotes the character output stream that is used to send data back to the client. It is a buffered version of `java.io.PrintWriter` called `javax.servlet.jsp.JspWriter` type object. The object `out` is used only in scriptlets. This implicit object has page scope.

```
<% out.println("Hello World!"); %>
```

JSP expressions are placed in the output stream automatically and hence we do not use this `out` object there explicitly.

config

This `javax.servlet.ServletConfig` type object refers to the configuration of the underlying servlet. It is used to retrieve initial parameters, servlet name, etc. This implicit object has page scope.

page

This object refers to the JSP page itself. It can be used to call any instance of the JSP page's servlet. This implicit object has page scope.

exception

This represents an uncaught exception, which causes an error page to be called. This object is available in the JSP page for which the `isErrorPage` attribute of the `page` directive is set to `true`. This implicit object has page scope.

21.8.8 Variables, Methods, and Classes

Variables, methods, and classes can be declared in a JSP page. If they are declared in the declaration section, they become part of the class. For example, variables declared in the declaration section become instance variables and are accessible from any method of the JSP page's servlet class.

Consider the following variable declaration:

```
<%! double PI = 22/7.0; %>
```

The resultant servlet code will look like this:

```
…
public final class method_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
double PI = 22/7.0;
…
```

These variables are also shared among multiple threads. Classes and methods can be declared in a similar way as follows:

```
<%!
int add(int a, int b) {
   return a + b;
}
class AnInnerClass {
}
%>
```

The resultant servlet source code will look like this:

```
…
public final class method_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
int add(int a, int b) {
   return a + b;
}
class AnInnerClass {
}
…
```

Variables, methods, and classes declared in the declaration section are available in scriptlets. So, the following scriptlet is valid for this declaration.

```
<%out.println(add(2, 3));%>
```

It displays the following:

```
5
```

JSP allows us to declare variables and classes, but not methods, in the scriptlet section. Variables declared in the scriptlet section are local to the `_jspService()` method. They are created each time the client makes a request and destroyed after the request is over. Similarly, classes declared in the scriptlet section become inner classes of the `_jspService()` method. Consider the following code:

```
<%
double temp = 0;
```

```
class TempClass {
}
%>
```

The resultant servlet code looks like this:

```
…
public final class method_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
…
  public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
…
double temp = 0;
class TempClass {
}
```

21.8.8.1 Synchronization

Note that variables declared in the declaration section become instance variables. Instance variables become shared automatically among all request handling threads. You must write the code to access these variables synchronously. Consider the following piece of code:

```
<%!int n = 1;%>
<%
for (int i = 0; i < 5; i++) {
   out.println("Next integer: " + n++ + "<br>");
   Thread.sleep(500);
}
%>
```

The code is intended to print five consecutive integers. Each time this JSP page is invoked, it increments the instance variable `n` five times and displays the value. The code segment works fine if only one request is dispatched to this JSP page at a time. However, if two or more requests are sent to this JSP page, the result is not displayed correctly. To demonstrate this and to allow collision, an artificial delay is given. A sample output is shown in Figure 21.7, if two requests are sent simultaneously.

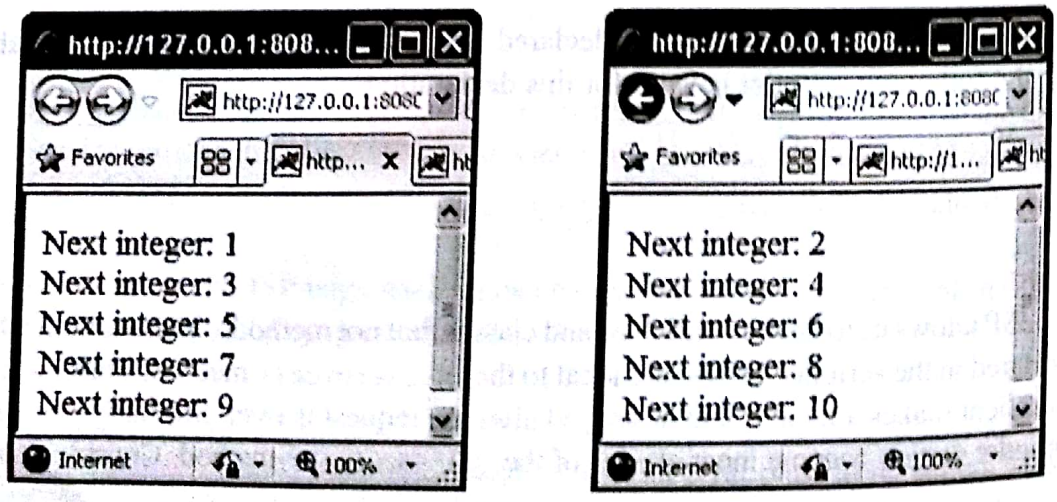


Figure 21.7 Accessing JSP page without synchronization

One of the solutions of this problem is to make the `for` loop atomic so that only one request can access it at a time.

```
<%!
   int n = 1;
   Object o = new Object();
%>
<%
synchronized(o)  {
   for (int i = 0; i < 5; i++) {
      out.println("Next integer: " + n++ + "<br>");
      Thread.sleep(500);
   }
}
%>
```

Another way to solve this problem is to inform the JSP container that this page is not thread-safe, so that the container dispatches requests one by one.

```
<%@ page isThreadSafe="false" %>
<%!int n = 1; %>
<%
for (int i = 0; i < 5; i++) {
   out.println("Next integer: " + n++ + "<br>");
   Thread.sleep(500);
}
%>
```

In either case, the result will be correct.

21.8.9 Standard Actions

The JSP engine provides many built-in sophisticated functions to the programmers for easy development of the web applications. These functions include instantiating objects in a JSP page, communicating with other JSP pages and servlets, etc. JSP *actions* are nothing but XML tags that can be used in a JSP page to use these functions.

Note that the same thing can be achieved by writing Java code within scriptlets. However, JSP action tags are a convenient way to use those functions. They also promote component framework as well as application maintainability. The following section describes commonly used JSP action tags.

include

This action tag provides an alternative way to include a file in a JSP page. The general syntax of the `include` action tag is

```
<jsp:include page="relativeURL | <%=expression%>" flush="true" />
```

For example, the following code includes the file `header.jsp` in the current page:

```
<jsp:include page="header.jsp" />
```

It is similar to the `include` directive but instead of inserting the text of the included file in the original file at compilation time, it actually includes the target at run-time. It acts like a subroutine where the control is passed temporarily to the target. The control is then returned back to the original JSP page. The result of the included file is inserted in the place of `<jsp:include>` action in the original JSP page. Needless to say, the included file should be a JSP file, servlet, or any other dynamic program that can process the parameters. The optional attribute `flush` can only take the value `true`.

Let us illustrate with an example. Suppose the content of `date.jsp` is as follows:

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

Now, consider the file `include.jsp`, which has included the `date.jsp` file using the `<jsp:include>` action tag as follows:

```
Before<br>
<jsp:include page="date.jsp" />
<br>After
```

This generates the following result:

```
Before
Date and time: Wed Dec 02 19:56:54 IST 2009
After
```

The value of the `page` attribute may be a dynamically generated file name. The following example illustrates this:

```
<%
String fileName = "sortByName.jsp";
String criteria = request.getParameter("sortCriteria");
if(criteria != null) fileName = criteria + ".jsp";
%>
<jsp:include page="<%=fileName%>" />
```

So, if the `include.jsp` page is invoked using the URL `http://127.0.0.1:8080/wt/jsp/include.jsp?sortCriteria=sortByRoll`, the include action effectively becomes:

```
<jsp:include page="sortByRoll.jsp" />
```

Note that the file `sortByRoll.jsp` must exist. Otherwise, an exception will be thrown. If the `include.jsp` page is invoked using the URL `http://127.0.0.1:8080/wt/jsp/include.jsp`, the include action effectively becomes

```
<jsp:include page="sortByName.jsp" />
```

param

The JSP `<jsp:param>` action allows us to append additional parameters to the current request. The general syntax is as follows:

```
<jsp:param name="parameterName" value="parameterValue | <%=expression%>" />
```

The `name` and `value` attributes of the `<jsp:param>` tag specify the case-sensitive name and value of the parameter, respectively.

It is typically used with the <jsp:include> and <jsp:forward> action tags. For example, the following code passes the control to the JSP page process.jsp temporarily, with two additional parameters, user and sessionId.

```
<jsp:include page="process.jsp">
   <jsp:param name="user" value="monali" />
   <jsp:param name="sessionId" value="12D43F3Q436N43" />
</jsp:include>
```

The value of the value attribute may be a dynamic value, but the value of the name attribute must be static. The following example illustrates this:

```
<% String user = request.getParameter("user"); %>
<jsp:include page="process.jsp">
   <jsp:param name="user" value="<%=user%>" />
   <jsp:param name="sessionId" value="<%=System.currentTimeMillis()%>" />
</jsp:include>
```

forward

This action tag hands over the current request to the specified page internally at the server side. If the current page has already generated any output, it is suppressed. The output will only be caused by the page that has handled the request last in the forward chain. The control is never returned to the original page. The general syntax of the JSP forward action tag is

```
<jsp:forward page="relativeURL | <%=expression%>" />
```

Consider the forward.jsp file, which forwards the current request to the date.jsp file using the <jsp:forward> action as follows:

```
Before<br>
<jsp:forward page="date.jsp" />
<br>After
```

This transfers the control to the date.jsp page. The result the file forward.jsp has generated so far (Before
 in our case) is cleared. The output is due to the page date.jsp only, as follows.

```
Date and time: Wed Dec 02 20:03:00 IST 2009
```

Like the <jsp:include> action tag, the value of the page attribute of <jsp:forward> may be a dynamic file name. The following example illustrates this:

```
<% String mailbox = request.getParameter("mailbox") + ".jsp";%>
<jsp:forward page="<%=fileName%>" />
```

Additional parameters may be specified using the <jsp:param> action as follows:

```
<%
   String mailbox = request.getParameter("mailbox") + ".jsp";
   String user = request.getParameter("user");
%>
<jsp:forward page="<%=fileName%>" >
   <jsp:param name="user" value="<%=user%>" />
<jsp:forward>
```

plugin

The <jsp:plugin> action is used to generate an HTML file that can download the Java plug-in on demand and execute applets or JavaBeans. It is an alternative way to deploy applets through Java plug-in. Since JSP pages are dynamic in nature, the developers of web applications can make use of Java plug-in to generate browser-specific tags to insert applets on the fly in a much easier and flexible way.

The <jsp:plugin> action generates the embed or object tag for the applet to be executed, depending upon the browser used. When the JSP is translated, the <jsp:plugin> action element is substituted by either an <embed> or an <object> HTML tag. The following code specifies an applet.

```
<jsp:plugin type="applet" code="Message" >
<jsp:params>
    <jsp:param name="message" value="Hello World!"/>
</jsp:params>
<jsp:fallback>
        <p> Unable to start Plug-in. </p>
</jsp:fallback>
</jsp:plugin>
```

The type attribute of the <jsp:plugin> tag specifies the type (bean or applet) of plug-in to be used and the code attribute specifies the name of the file (without extension) containing the byte code for this plug-in. For example, the <jsp:plugin> action inserts an applet whose code can be found in the class file Message.class.

The <jsp:plugin> action tag has a similar set of attributes as the <applet> tag. The <jsp:params> and <jsp:param> actions are used to pass parameters to the plug-in. The <jsp:fallback> tag specifies the message to be displayed if the Java plug-in fails to start due to some unavoidable reason.

Here is the source code for the Message applet (Message.java).

```
public class Message extends java.applet.Applet {
   String msg;
   public void init() {
      msg = getParameter("message");
   }
   public void paint(java.awt.Graphics g) {
      g.drawString(msg, 20, 20);
   }
}
```

This applet takes a single parameter message and displays its value at location (20, 20) on the browser's screen, relative to the applet window.

As mentioned earlier, the web server generates either an <embed> or an <object> tag, depending upon the browser that requested the JSP page. For example, the following code is generated if Tomcat 6.0 is used as the web server and Internet Explorer 8.0 makes the request.

```
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-
win.cab#Version=1,2,2,0">
```

```
<param name="java_code" value="Message">
<param name="type" value="application/x-java-applet;">
<param name="message" value="Hello World!">
<comment>
<EMBED type="application/x-java-applet;"
pluginspage="http://java.sun.com/products/plugin/" java_code="Message"
message="Hello World!"/>
<noembed>

      <p> Unable to start Plug-in. </p>

</noembed>
</comment>
</object>
```

Though JSP specification includes the `<jsp:plugin>` action tag, different vendors may implement it differently. For detailed information, read the documentation of the web server.

useBean

It is used to instantiate a JavaBean object for later use. We shall discuss this action in Section 21.9.1 in detail.

setProperty

It is used to set a specified property of a JavaBean object. A detailed description of this action may be found in Section 21.9.2.

getProperty

It is used to get a specified property from a JavaBean object. This action has been discussed in Section 21.9.3 in detail.

21.8.10 Tag Extensions

One significant feature added in the JSP 1.1 specification is *tag extension*. It allows us to define and use custom tags in a JSP page exactly like other HTML/XML tags, using Java code. The JSP engine interprets them and invokes their functionality. So, Java programmers can provide application functionality in terms of custom tags, while web designers can use them as building blocks without any knowledge of Java programming. This way, JSP tag extensions provide a higher-level application-specific approach to the authors of HTML pages.

Though a similar functionality can be achieved using JavaBean technology, it lacks many features such as nesting, iteration, and cooperative actions. JSP tag extension allows us to express complex functionalities in a simple and convenient way.

21.8.10.1 Tag type

JSP specification defines the following tags, which we can create and use in a JSP file.

Simple Tags

Tags with no body or no attribute. Example:

```
<ukr:copyright>
```

Tags with attributes

Tags that have attributes but no body. Example:

```
<ukr:hello name="Monali" />
```

Tags with body

Tags that can contain other tags, scripting elements, and text between the start and end tags. Example:

```
<ukr:hello>
  <%= name%>
</ukr:hello>
```

Tags Defining Scripting Variables

Tags that can define scripting variables, which can be used in the scripts within the page. Example:

```
<ukr:animation id="logo" />
<% logo.start(); %>
```

Cooperating Tags

Tags that cooperate with each other by means of named shared objects. Example:

```
<ukr:tag1 id="obj1" />
<ukr:tag2 name="obj1" />
```

In this example, `tag1` creates a named object called `obj1`, which is later used by `tag2`.

21.8.10.2 Writing tags

In this section, we shall discuss how to define and use simple tags. The design and use of a custom tag has the following basic steps:

- Tag Definition
- Provide Tag Handler
- Deploy the tag
- Use the tag in the JSP file

Tag Definition

Before writing the functionality of a tag, you need to consider the following points:

- Tag Name—The name (and prefix) that will be used for the tag we are going to write
- Attributes—Whether the tag has any attribute and whether it is mandatory
- Nesting—Whether the tag contains any other child tag. A tag directly enclosing another tag is called the *parent* of the tag it encloses
- Body Content—Whether the tag contains anything (such as text) other than child tags

Provide Tag Handler

The functionality of a tag is implemented using a Java class called *tag handler*. Every tag handler must be created by directly or indirectly implementing the `javax.servlet.jsp.tagext.Tag` interface, which provides the framework of the JSP tag extension. In this section, we shall develop a simple tag named `<ukr:hello>`, which has a single optional attribute, `name`. The `<ukr:hello>` tag prints "Hello <name>" or "Hello World!" depending upon whether the attribute `name` is specified or not.

Note that the tag `<ukr:hello>` does not do much. It helps us understand the basic steps that we must follow to write a custom tag. In practice, custom tags will do complex tasks, but the procedure for developing such tags is exactly the same as the `<ukr:hello>` tag. After creating the `<ukr:hello>` tag, it is used in either of the following ways:

```
<ukr:hello name="Monali" />
```

This prints "Hello Monali".

```
<ukr:hello />
```

This prints "Hello World!". Here is the complete source code for the hello tag handler (`HelloTag.java`).

```
package tag;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import java.io.IOException;
public class HelloTag extends SimpleTagSupport {
   String name = "World!" ;
   public void doTag() throws JspException, IOException {
      JspWriter out = getJspContext().getOut();
      out.println("Hello " + name);
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Though each tag handler must implement the `javax.servlet.jsp.tagext.Tag` interface, it is convenient to use one of the default implementations, the `TagSupport` or `SimpleTagSuport` class. Our tag is one without any body and hence we have used the `SimpleTagSupport` class. Each tag handler must define some predefined methods [Table 21.3] that are called by the JSP engine. For example, the JSP engine calls the `doStartTag()` and `doEndTag()` methods when the start tag and the end tag, respectively, are encountered. The class `SimpleTagSupport` implements most of the work of a tag handler. The implementation of our class is simple, as it extends the `SimpleTagSupport` class. We have only implemented the `doTag()` method, which is called when the JSP engine encounters the start tag.

Table 21.3 Tag handler methods

Tag Handler Type	Methods
Simple	doStartTag, doEndTag, release
Attributes	doStartTag, doEndTag, set/getAttribute1...N
Body, No Interaction	doStartTag, doEndTag, release
Body, Interaction	doStartTag, doEndTag, release, doInitBody, doAfterBody

The hello tag has one attribute, `name`, whose value the handler must access. The JSP engine sets the value of an attribute `xxx` using the method `setXxx()` of the handler. This `setXxx()` method is invoked after the start tag, but before the body of the tag. So, we have inserted one method `setName()`, which will by used by the JSP engine to pass the value of the attribute `name`. Inside the handler, we have stored this value in a variable `name`. If a tag has multiple attributes, the corresponding tag handler must have separate set functions, one for each of these attributes.

Finally, we write the string "Hello" appended with the value of the `name` attribute to the servlet output stream. A reference to this output stream is obtained using the `getOut()` method on `javax.servlet.jsp.JspContext`, which is returned by the `getJspContext()` method.

Deploy the tag

Save this code in file `HelloTag.java` and put it in the application's `/WEB-INF/classes/tag` directory. To compile this file, we need the necessary tag classes that are provided as the jar file `jsp-api.jar`. This jar file can be found in the `lib` directory of Tomcat's installation directory. Make sure that this jar file is in your classpath during compilation. You can use the following command in the `/WEB-INF/classes/tag` directory to compile the `HelloTag.java` file.

```
javac -classpath ../../../../../lib/jsp-api.jar HelloTag.java
```

This will generate the `HelloTag.class` file in the `/WEB-INF/classes/tag` directory. If everything goes fine, restart the web server. The tag class is now ready to use.

Now, we have to map the tag `hello` with this tag handler class, `HelloTag`. This is done in an XML file called **T**ag **L**ibrary **D**escriptor (TLD). A TLD contains information about a tag library as well as each tag contained in the library. The JSP engine uses TLDs to validate tags used in the JSP pages. The Tag Library Descriptor is an XML document conforming to a DTD specified by Sun Microsystems.

We shall name our tag library file as `tags.tld` and put it in the application's `/WEB-INF/taglib` directory. For example, if the application's root directory is `wt`, put the `tags.tld` file in the `/wt/WEB-INF/taglib` directory. Make the following entry in the `tags.tld` file.

```
<?xml version="1.0"?>
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>Simple tag library</short-name>
  <tag>
    <description>Prints Hello 'name'</description>
    <name>hello</name>
```

```
        <tag-class>tag.HelloTag</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <name>name</name>
            <required>false</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>
</taglib>
```

Each `<tag>` element in the TLD file describes a tag. The `<name>` element specifies the name of the tag that will be used in the JSP file. The `<tag-class>` element associates the handler class with this tag name. In this case, the `tag.HelloTag` class is the handler of the tag `hello`. Each `<attribute>` element specifies the attribute that can be used for this tag. Our tag can have only one attribute, `name`. The `<required>` element specifies whether the attribute is mandatory (`true`) or optional (`false`). A tag without a body must specify that its `body-content` is empty.

Use the tag in the JSP file

The tag `hello` is now ready to use. Use the following entry in the JSP page `tag.jsp` and put it in the applications root directory, i.e., `/wt` in our case.

```
<%@ taglib prefix="ukr" uri="/taglib/tags.tld" %>
<ukr:hello name="Monali"/>
```

The `taglib` directive includes a tag library whose tags will be used by the JSP page. It must appear before using any custom tags it refers to. The `uri` attribute specifies a URI that uniquely identifies the TLD. The `prefix` attribute specifies the prefix to be used for every tag defined in this TLD. The prefix distinguishes tags in one library from those in others, if they contain tags with the same name. Now, write the following URL in the address bar of your web browser and press enter.

```
http://127.0.0.1:8080/wt/tag.jsp
```

It displays

```
Hello Monali
```

For the following code

```
<%@ taglib prefix="ukr" uri="/taglib/tags.tld" %>
<ukr:hello />
```

The code

```
Hello World!

<%@ taglib prefix="ukr" uri="/taglib/tags.tld" %>
<%!String name="Kutu"; %>
<ukr:hello name="<%=name%>"/>
```

displays

```
Hello Kutu
```

21.8.11 Iterating a Tag Body

Sometimes, it is necessary to iterate a specific code several times. For example, suppose we want to generate a table containing integers and their factorial value. We can write scripts to do this, as follows:

```
for(int i = 2; i <= 6; i++) {
%>
<%=i%>!=<ukr:fact no="<%=i%>" /><br>
<%
}
%>
```

We have assumed that there exists a tag handler for `<ukr:fact>` as follows:

```
package tag;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import javax.servlet.jsp.JspException;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
public class FactTag extends SimpleTagSupport {
   int no;
   public void doTag() throws JspException, IOException {
      int prod = 1;
      for(int j = 1;j <= no; j++)
         prod *= j;
      JspWriter out = getJspContext().getOut();
      out.println(prod);
   }
   public void setNo(int no) {
      this.no = no;
   }
}
```

It would be better, if the same table uses the following code:

```
<ukr:FactTable start="2" end="6">
   ${count}! = ${fact}<br>
</ukr:FactTable>
```

We can reduce the amount of code in the scripting element by moving the flow control to the tag handlers. The basic idea is to write a tag called iteration tags that iterates its body.

The iteration tag retrieves two parameters start and end. It calculates the factorial of each number between these two numbers (both inclusive) and assigns them to a scripting variable. The body of the tag retrieves the numbers and their factorials from scripting variables. Here is the handler for the iteration tag.

```
package tag;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import javax.servlet.jsp.JspException;
import java.io.IOException;
public class FactorialTag extends SimpleTagSupport {
   int start, end;
   public void doTag() throws JspException, IOException {
```

```
        for(int i = start; i <= end; i++) {
            int prod = 1;
            for(int j = 1; j <= i; j++)
                prod *= j;
            getJspContext().setAttribute("count", String.valueOf(i) );
            getJspContext().setAttribute("fact", String.valueOf(prod) );
            getJspBody().invoke(null);
        }
    }
    public void setStart(int start) {
        this.start = start;
    }
    public void setEnd(int end) {
        this.end = end;
    }
}
```

This handler, in each iteration, stores the value of the integer and its factorial in the `count` and `fact` variable, respectively, using the following code.

```
getJspContext().setAttribute("count", String.valueOf(i) );
getJspContext().setAttribute("fact", String.valueOf(prod) );
```

The body of the tag is iterated as follows:

```
getJspBody().invoke(null);
```

Now, make the following entry in the `tags.tld` file.

```
<?xml version="1.0"?>
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>Simple tag library</short-name>
    <tag>
        <description>Calculates factorial from 'start' to 'end'</description>
        <name>FactTable</name>
        <tag-class>tag.FactorialTag</tag-class>
                <body-content>scriptless</body-content>
        <attribute>
            <name>start</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
        <attribute>
            <name>end</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>
</taglib>
```

Obtaining the factorial tables is now very simple. Consider the following code.

```
<%@ taglib prefix="ukr" uri="/taglib/tags.tld" %>
Factorial table<br>
```

```
<ukr:FactTable start="2" end="6">
   ${count}! = ${fact}<br>
</ukr:FactTable>
```

In the body of the tag, the values of the count and fact variables are retrieved using expression language. Note that the JSP page does not contain a single piece of script. It is displayed as follows:

```
Factorial table
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

21.8.12 Sharing Data Between JSP Pages

All JSP pages participate in an HTTP session unless the `session` attribute of the `page` directive is set to `false`. An HTTP session is represented by the implicit object `session`. This `session` object has session scope and is thus shared among all the pages within the session.

This session object can be used as a shared repository of information such as beans and objects among JSP pages of the same session. For example, `login.jsp` page may store the user name in the session, while subsequent pages such as `home.jsp` can use it. Consider the code segment in `login.jsp`:

```
String user = request.getParameter("user");
session.setAttribute("user", user);
```

Now, see the code segment in `home.jsp`.

```
String user = (String)session.getAttribute("user");
```

21.9 BEANS

JavaBeans are reusable Java components that allow us to separate business logic from presentation logic. Technically, a JavaBean class is a Java class that meets the following requirements:

- It has a public, no argument constructor.
- It implements the `java.io.Serializable` or `java.io.Externalizable` interface.
- Its properties are accessible using methods that are written following a standard naming convention.

With this simple definition, properly designed beans can be used virtually in all Java environments such as JSP, servlet, applet, or even in Java applications. Note that most of the existing classes are already bean classes or they can be converted to bean classes very easily.

The methods of a bean must follow a naming convention. If the name of a bean property is *xxx*, the associate reader and writer method must have the name `getXxx()` and `setXxx()`, respectively. Following is a sample class declaration for JavaBean `Factorial`:

```
package bean;
public class Factorial implements java.io.Serializable {
    int n;
    public int getValue() {
        int prod = 1;
        for(int i = 2; i <= n; i++)
            prod *= i;
        return prod;
    }
    public void setValue(int v) {
        n = v;
    }
}
```

Since no constructor is defined explicitly in this class, a zero argument constructor is inserted in the `Factorial` class. The `Factorial` bean class has one property, `value`. So, the name of the reader method is `getValue()`. Similarly, the name of the writer method is `setValue()`. The name of the member variable need not be `value`; it is the property that we want to manipulate on a `Factorial` bean.

Save this code in the file `Factorial.java` and store it in the application's `/WEB-INF/classes/bean` directory. Compile the class exactly like other Java classes. Use the following command in the `/WEB-INF/classes/bean` directory.

```
javac Factorial.java
```

This generates a class file, `Factorial.class`. If everything goes fine, restart the tomcat web server. Tomcat loads all the class files under `/classes` directory and add its subdirectories to the CLASSPATH of the Java Runtime Environment (JRE). Those class files can now be used exactly like other Java classes.

There are three action elements that are used to work with beans.

21.9.1 useBean

A JSP action element `<jsp:useBean>` instantiates a JavaBean object into the JSP page. The syntax is

```
<jsp:useBean id="object_name" class="class_name" scope="page | request |
session |application" />
```

Here, the `id` attribute refers to the name of the object to be created and the attribute `class` specifies the name of the JavaBean class from which the object will be instantiated. The attribute `scope` specifies the area of visibility of the loaded bean. The effect of the `<jsp:useBean>` element is equivalent to instantiating an object as follows:

```
<% class_name object_name = new class_name(); %>
```

For example, to instantiate a Factorial bean in a JSP page, the following action is used:

```
<jsp:useBean id="fact" scope="page" class="bean.Factorial" />
```

This is equivalent to the following scriptlet:

```
<% bean.Factorial fact = new bean.Factorial(); %>
```

Once a bean object is loaded into a JSP page, we can use two other action elements to manipulate it.

21.9.2 setProperty

The `<jsp:setProperty>` action tag assigns a new value to the specified property of the specified bean object. It takes the following form:

```
<jsp:setProperty name="obj_name" property="prop_name" value="prop_value"/>
```

The object name, property name, and its value are specified by the `name`, `property`, and `value` attributes, respectively. This is equivalent to calling the `setProp_name()` method on the specified object `obj_name` as follows:

```
<% obj_name.setProp_name(prop_value); %>
```

To set a property of our bean object `fact`, we use the following:

```
<jsp:setProperty name="fact" property="value" value="5" />
```

The equivalent scriptlet is as follows:

```
<% fact.setValue(5); %>
```

21.9.3 getProperty

The `<jsp:getProperty>` action element retrieves the value of the specified property of the specified bean object. The value is converted to a string. It takes the following form:

```
<jsp:getProperty name="obj_name" property="prop_name"/>
```

The object name and its property name are specified by the `name` and `property` attributes, respectively. This is equivalent to calling the `getProp_name()` method on the specified object `obj_name` as follows:

```
<%= obj_name.getProp_name() %>
```

To get the value of the property `value` of our `fact` bean object, we use the following:

```
<jsp:getProperty name="fact" property="value" />
```

The equivalent scriptlet is as follows:

```
<%= fact.getValue() %>
```

21.9.4 Complete Example

Following is a complete JSP page:

```
<table border="1">
  <caption>Factorial table</caption>
  <tr><th width="50">n</th><th width="100">n!</th></tr>
```

```
<jsp:useBean id="fact" scope="page" class="bean.Factorial" />
<%
for(int i = 2; i < 6; i++) {
%>
<jsp:setProperty name="fact" property="value" value="<%=i%>" />
<tr><td><%=i%></td><td><jsp:getProperty name="fact" property="value"
/></td></tr>
<%
}
%>
</table>
```

It generates the output shown in Figure 21.8.

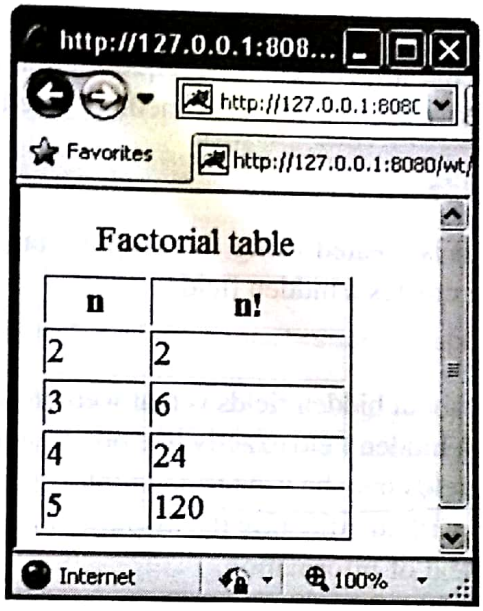


Figure 21.8 Using beans in JSP

21.9.5 Other Usage

Once a bean object is loaded into the page, it can be used exactly like other objects in scripting elements in the same JSP page. Consider the following code:

```
<jsp:useBean id="fact" scope="page" class="bean.Factorial" />
```

This is equivalent to the following instantiation:

```
<% bean.Factorial fact = new bean.Factorial(); %>
```

Now, the bean object can be accessed using its name, as follows:

```
<%fact.setValue(6);%>
<%=fact.getValue()%>
```

This displays the following output:

```
720
```

21.10 SESSION TRACKING

Since HTTP is a stateless protocol, the web server cannot remember previous requests. Consequently, the web server cannot relate the current request with the previous one. This creates a problem for some applications that require a sequence of related request–response cycles. Examples include online examination systems, email checking systems, and banking applications. How does the server know how many questions you have answered so far or when did you start the examination?

We shall use a simple application to demonstrate session tracking using different methods. In this application, the server initially dynamically generates an HTML file to display the integer 0, with two buttons captioned "prev" and "next". If the user clicks on the *next* button, the server sends the integer next to the one displayed. Similarly, when *prev* button is clicked, the server sends the integer previous to the one displayed.

Note that each time the server receives a request, it has to remember the number it sent in the previous step. Let us now discuss different methods used for session tracking.

21.10.1 Hidden Fields

An HTML hidden field is created using the `<input>` tag, with the `type` attribute `hidden`. For example, the following creates a hidden field.

```
<input type="hidden" name="user" value="monali" >
```

The interesting part about hidden fields is that web browsers do not display them, but send the name–value pair of each hidden field exactly like other input elements when the enclosing form is submitted. So, hidden fields may be used to send information back and forth between server and client to track sessions, without affecting the display. Hidden fields are ideal for applications that do not require a great deal of information.

Let us now implement the application mentioned, using hidden fields.

```
<%@page import="java.util.*"%>
<html>
  <head><title>Hidden field demo</title></head>
  <body>
    <%
      int current = 0;
      String last = request.getParameter("int");
      String button = request.getParameter("button");
      if(button != null) {
        if(button.equals("next"))
          current = Integer.parseInt(last) + 1;
        else
          current = Integer.parseInt(last) - 1;
      }
      out.println(current);
    %>
    <br>
    <form name="myForm" method="post">
      <input type="hidden" name="int" value="<%=current%>">
```

```
            <input type="submit" name="button" value="prev">
            <input type="submit" name="button" value="next">
        </form>
    </body>
</html>
```

In this method, the JSP page uses a single hidden field in the generated HTML file named "int". The value of this hidden field is the integer being sent currently. The hidden field, together with the two submit buttons captioned "prev" and "next", are put in a form. So, the web browser receives an HTML file like this:

```
<html>
    <head><title>Hidden field demo</title></head>
    <body>

        0

        <br>
        <form name="myForm" method="post">
            <input type="hidden" name="int" value="0">
            <input type="submit" name="button" value="prev">
            <input type="submit" name="button" value="next">
        </form>
    </body>
</html>
```

The HTTP `POST` method is used in the form and no `action` attribute is specified. When the user clicks any of the two submit buttons, the value of this hidden field (which is the integer currently displayed) is sent to the same JSP page behind the scene. The JSP page extracts this value and understands that this value was sent in response to the previous request. This way, the hidden field helps the JSP page to remember information sent earlier. It also determines the button whose click event generates this request. The JSP page can then easily calculate the value to be sent next, depending on the button clicked.

The advantage of this method is that it does not require any special support from the client's side. Hidden fields are supported by all browsers and underlying information propagation is absolutely transparent to the user.

Note that users can view the source code of the HTML page and consequently see the value of the hidden field. So, the JSP page should not pass sensitive information such as passwords in the hidden field.

21.10.2 URL Rewriting

This is another simple but elegant method to track sessions and is widely used. It does not require any special support from the web browser. Remember that HTTP allows us to pass information using HTTP URL. This method makes use of that concept. The session information is appended to the URL. Here is the solution using URL rewriting.

```
<%@page import="java.util.*"%>
<html>
   <head><title>URL rewriting demo</title></head>
   <body>
      <%
         int last = 0;
         String param = request.getParameter("int");
         if(param != null) last = Integer.parseInt(param);
         out.println(last);
      %>
      <br>
      <a href="intUrl.jsp?int=<%=last-1%>">prev</a>
      <a href="intUrl.jsp?int=<%=last+1%>">next</a>
   </body>
</html>
```

In this case, the JSP page generates and sets the URL to be called for the hyperlinks "prev" and "next". For example, the web gets this HTML page for the first time:

```
<html>
   <head><title>URL rewriting demo</title></head>
   <body>

0

<br>
<a href="intUrl.jsp?int=-1">prev</a>
<a href="intUrl.jsp?int=1">next</a>
   </body>
</html>
```

When a user clicks on the "prev" hyperlink, the integer that should be sent by the JSP page for this request is appended to the URL. The JSP page simply extracts this information and sends it back with the new URLs for those two hyperlinks. A similar sequence of events happen when a user clicks on the "next" button.

21.10.3 Cookies

This method requires support from the web browsers. In this method, session information is represented by a named token called *cookie*. The JSP page sends this cookie to the web browser. The browser must be configured properly to accept it. Upon receiving a cookie, the web browser installs it. This cookie is then sent back to the server with the subsequent requests. The JSP page can then extract session information from this cookie. This way, the server can identify a session.

By default, most browsers support cookies. However, due to security reasons, very often cookie support is disabled and in that case session management will not work correctly. Following is the JSP page for the application we discussed earlier.

```
<html>
   <head><title>Cookie demo</title></head>
   <body>
      <%
         int current = 0;
         Cookie cookie = null;
```

```
            Cookie[]  cookies = request.getCookies();
            if(cookies != null)
               for(int i = 0; i < cookies.length; i++)
                  if(cookies[i].getName().equals("last"))
                     cookie = cookies[i];
            if(cookie != null) {
               String button = request.getParameter("button");
               if(button != null) {
                  if(button.equals("next"))
                     current = Integer.parseInt(cookie.getValue()) + 1;
                  else
                     current = Integer.parseInt(cookie.getValue()) - 1;
               }
            }
            response.addCookie(new Cookie("last", String.valueOf(current)));
            out.println(current);
         %>
         <br>
         <form name="myForm" method="post">
            <input type="submit" name="button" value="prev">
            <input type="submit" name="button" value="next">
         </form>
      </body>
   </html>
```

The JSP page first looks for a cookie named "last". If it finds the cookie, the value of the cookie is obtained using the `getValue()` method. Depending on the button clicked, the integer to be sent next is calculated and stored in the variable `current`. This integer, together with a new cookie with the name "last" having the current value set is sent to the web browser. However, if the JSP page does not find any cookie with the name "last", which happens during the first time, it sends a new cookie having the value 0, which is the initial value of the variable `current`.

21.10.4 Session API

JSP technology (and its underlying servlet technology) provides a higher level approach for session tracking. The basic building block of this session API is the `javax.servlet.http.HttpSession` interface. The `HttpSession` objects are automatically associated with the client by the cookie mechanism. An `HttpSession` object is the one that persists during a session, until it times out or is shutdown by the JSP page participating in this session.

In a JSP page, an HTTP session is created by default if it is not suppressed by setting the `session` attribute of the `page` directive to `false` as follows.

```
<%@ page session="false" %>
```

This session can be accessed by the implicit object `session`. A JSP page may also create an `HttpSession` object explicitly, using the following method in `HttpServletRequest`.

```
HttpSession getSession(boolean create);
HttpSession getSession();
```

The first version takes a boolean parameter that indicates whether a new session has to be created if one does not exist. If the parameter is `true`, a new session is created if it does not exist.

Otherwise, the existing session object is returned. If the parameter is false, it returns the existing session object if it exists, and null otherwise. The second version simply calls the first version with the parameter `true`.

In the `HttpSession` object, we can store and retrieve key–value pair using `setAttribute()`, `getAttribute()`, and `getAttributes()` methods.

```
Object getAttribute(String key);
Enumeration getAttributeNames();
void setAttribute(String key, Object value);
```

The return type of the `getAttribute()` method is `Object`. So you must typecast it to the required type of data that was stored with a key in the session object. If the key specified in the `setAttribute()` method does not already exist in the session, the specified value is stored with this key. Otherwise, the old value is overwritten with the specified value.

The JSP engine uses the cookie mechanism to keep track of sessions. A session object is associated with usually a long alphanumeric ID. This session ID is sent to the client as a cookie with the name `JSESSIONID`, when the client makes a request for the first time using the HTTP response header `Set-Cookie`, as follows:

```
Set-Cookie: JSESSIONID=g52d15acu325dlw532h234
```

For subsequent requests, the client sends this cookie back to the server using the HTTP request header `Cookie` as follows:

```
Cookie: JSESSIONID=g52d15acu325dlw532h234
```

The server can then identify that this request has come from the same client. However, if the client does not accept cookies, this mechanism fails. In that case, programmers may use the URL rewriting mechanism, where each URL must have the session ID appended. The JSP engine, on behalf of programmers, provides a straightforward mechanism to implement URL rewriting. The `HttpServletResponse` object provides methods to append this session ID automatically. These methods identify whether the client is configured to accept cookies. They append the session ID only if the client does not accept cookies.

```
java.lang.String encodeURL(java.lang.String);
java.lang.String encodeRedirectURL(java.lang.String);
```

The `encodeRedirectURL()` method is used for the `sendRedirect()` and `encodeURL()` method is used for the rest to create such URLs.

Here is the solution of our application using `HttpSession` API:

```
<%@page import="java.util.*"%>
<html>
   <head><title>Hidden field demo</title></head>
   <body>
      <%
int current = 0;
      String last = (String)session.getAttribute("last");
      if(last != null) {
```

```
            String button = request.getParameter("button");
               if(button != null) {
                  if(button.equals("next"))
                     current = Integer.parseInt(last) + 1;
                  else
                     current = Integer.parseInt(last) - 1;
               }
            }
            session.setAttribute("last", String.valueOf(current));
            out.println(current);
         %>
         <br>
         <form name="myForm" method="post">
            <input type="submit" name="button" value="prev">
            <input type="submit" name="button" value="next">
         </form>
      </body>
   </html>
```

21.11 USERS PASSING CONTROL AND DATA BETWEEN PAGES

Sometimes, we need to hand over the control to another page, with the necessary data passed to it. In this section, we shall discuss how to pass control across pages with data passed to them.

21.11.1 Passing Control

Sometimes, a JSP page wants to pass the control to another server-side program for further processing. For example, in an e-mail application, the `login.jsp` page on user verification may want to forward the request to the `home.jsp` page. This is done using the `<jsp:forward>` action in the `login.jsp` page as follows:

```
String user = request.getParameter("user");
String password = request.getParameter("password");
//verify the user here
<jsp:forward page="home.jsp" />
```

Once the JSP engine encounters the `<jsp:forward>` action, it stops the processing of the current page and starts the processing of the page (called target page) specified by the `page` attribute. The rest of the original page is never processed further.

The `HttpServletRequest` and `HttpServletResponse` objects are passed to the target page. So, the target page can access all information passed to the original page. You can pass additional parameters to the target page using the `<jsp:param>` action as follows.

```
String user = request.getParameter("user");
String password = request.getParameter("password");
//verify the user here
double startTime = System.currentTimeMillis();
<jsp:forward page="home.jsp">
   <jsp:param name="startTime" value="<%=startTime%>" />
</jsp:forward>
```

The target page can access these parameters in the same way as the original parameters, using the `getParameter()` and/or `getParameterNames()` methods.

21.11.2 Passing Data

The scope of an object indicates from where the object is visible. JSP specification defines four types of scopes: `page`, `request`, `session`, and `application`. The objects having `page` scope are only available within the page. So, objects that are created with `page` scope in a page are not available in another page where the control is transferred using the `<jsp:forward>` action. If you make an object visible across multiple pages across the same request, create the object with `request` scope. The following example illustrates this.

```
<!--original.jsp-->
<jsp:useBean id="fact" scope="request" class="bean.Factorial" />
<%fact.setValue(5);%>
<jsp:forward page="new.jsp" />
```

This page creates a bean object with the `request` scope and sets the value property with 5. It then passes the control to the `new.jsp` page, which looks like this.

```
<!--new.jsp-->
<jsp:useBean id="fact" scope="request" class="bean.Factorial" />
<%=fact.getValue()%>
```

Since the bean was saved as the `request` scope in the `original.jsp` page, the `<jsp:useBean>` action in the `new.jsp` page finds and uses it.

For the URL `http://127.0.0.1:8080/wt/original.jsp`, the following is displayed:

```
120
```

Depending upon your requirement, you may create an object having any one of the four scopes.

21.12 SHARING SESSION AND APPLICATION DATA

The objects having `request` scope are available in all pages across a single request. However, sometimes objects should be shared among multiple requests.

Think about on online examination system. JSP pages requested from the same browser must share the same user name that was provided during the login procedure. This is required because different users have different sets of data such as expiry time, number of questions answered so far, number of correct answers, and so on. This type of information can be shared through `session` scope.

The objects having `session` scope are available to all pages requested by the same browser. The implicit object `session` is one such object. This session object can be used to store and retrieve other objects. So, objects may be shared across pages in the same session using this `session` object. Following is a code fragment used in `login.jsp`.

```
<!-login.jsp-->
<%
String user = request.getParameter("user");
String password = request.getParameter("password");
```

```
//verify the user here
double startTime = System.currentTimeMillis();
session.setAttribute(user, String.valueOf(startTime));

<jsp:forward page="exam.jsp" />
%>
```

It stores the time when the user starts its session in the session object and forwards the request to `exam.jsp`. The `exam.jsp` will be called many times by the user. The `exam.jsp` retrieves the start time for this user, checks whether the user has exhausted its time limit, and takes necessary actions.

```
String user1 = (String)session.getAttribute("user");

double startTime = Double.parseDouble((String)session.getAttribute("user"));
double currentTime = System.currentTimeMillis();
if(currentTime - startTime > 600000) {  //duration is 10 minutes
   //the time is over, forward the request to the page logout.jsp
%>
<jsp:forward page="logout.jsp" />
<%
}
%>
```

Now, think about the same online examination system where different users are giving the examination using different browser windows, but using the same database. For efficiency purposes, information should be shared among pages even if the pages are requested by different users. This type of information can be shared through application scope.

The objects that have application scope are shared by all pages requested by any browser. One such implicit object is `application`. Consider the following code fragment in the `login.jsp` page:

```
Object obj = application.getAttribute("config");
if(obj == null) {
   ExamConfig conf = new ExamConfig();
   application.setAttribute("config", conf);
}
```

It looks for an object having the name "config" in the `application` object. If no such object exists, it creates an `ExamConfig` object and stores one such object in the `application` object. When the `ExamConfig` object is created, connection to the database is established. This database connection can now be shared by all other pages related to this application.

Here is the code fragment used in the `exam.jsp` page.

```
ExamConfig conf = (ExamConfig)application.getAttribute("config");
Statement stmt = conf.getConnection().createStatement();
//use stmt to fire SQL query
```

All requests that use the `exam.jsp` file can simply use the same object stored in the application object to get the `Statement` object and fire the SQL query.

Application program Interface.

21.13 DATABASE CONNECTIVITY

Most of the web applications need access to databases in the backend. Java DataBase Connectivity (JDBC) allows us to access databases through Java programs. It provides Java classes and interfaces to fire SQL and PL/SQL statements, process results (if any), and perform other operations common to databases. Since Java Server Pages can contain Java code embedded in them, it is also possible to access databases from Java Server Pages. The classes and interfaces for database connectivity are provided as a separate package, `java.sql`.

21.14 JDBC DRIVERS

A Java application can access almost all types of databases such as relational, object, and object-relational. The access to a specific database is accomplished using a set of Java interfaces, each of which is implemented by different vendors differently. A Java class that provides interfaces to a specific database is called JDBC driver. Each database has its own set of JDBC drivers. Users need not bother about the implementation of those Java classes. They can concentrate on developing database applications. Those drivers are provided (generally freely) by database vendors. This way, JDBC hides the underlying database architecture. JDBC drivers provided by database vendors convert database access requests to database-specific APIs.

JDBC drivers are classified into four categories depending upon the way they work.

21.14.1 JDBC-ODBC bridge (Type 1)

This is the *Type 1* driver. This type of drivers cannot talk to the database directly. It needs an intermediate ODBC (**O**pen **D**ata**B**ase **C**onnectivity) driver, with which it forms a kind of bridge. The driver translates JDBC function calls to ODBC method calls. ODBC makes use of native libraries of the operating system and is hence platform-dependent. For this mechanism to function correctly, the ODBC driver must be available in the client machine and must also be configured correctly, which is generally a long and tedious process. For this reason, the Type 1 driver is used for experimental purposes or when no other JDBC driver is available. Sun provides a Type 1 JDBC driver with JDK 1.1 or later.

21.14.2 Native-API, Partly Java (Type 2)

This is very similar to the Type 1 driver. However, it does not forward the JDBC call to the ODBC driver. Instead, it translates JDBC calls to database-specific native API calls. This driver is not a pure Java driver, as it interfaces with non-Java APIs that communicates with the database. This approach is a little bit faster than the earlier one, as it interfaces directly with the database through the native APIs. However, it has limitations similar to the previous one. This means that the client must have vendor-specific native APIs installed and configured in it.

21.14.3 Middleware, Pure Java (Type 3)

In this case, the JDBC driver forwards the JDBC calls to some middleware server using a database-independent network protocol. The middleware server acts as a *gateway* for multiple (possibly

different) database servers and can use different database-specific protocols to connect to different database servers. This intermediate server sends each client request to a specific database. The results are then sent back to the intermediate server, which in turn sends the result back to the client. This approach hides the details of connections to the database servers and makes it possible to change the database servers without affecting the client.

21.14.4 Pure Java Driver (Type 4)

These types of drivers communicate with the database directly by making socket connections. It has distinct advantages over other mechanisms, in terms of performance and development time. Since it talks with the database server directly, no other subsidiary driver is needed. In this book, we shall use only the Type 4 driver. Figures 21.9 (i) and (ii) show the JDBC two-tier and three-tier architectures.

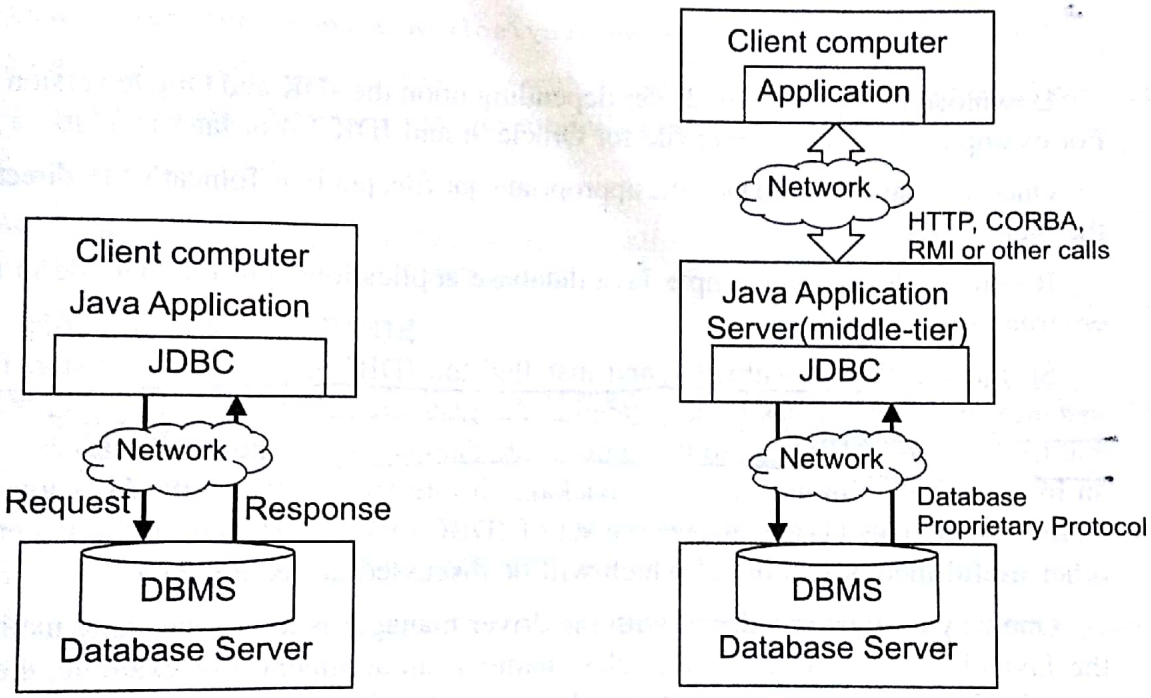


Figure 21.9 JDBC architecture (i) two tier (ii) three-tier

Table 21.4 gives the Java JDBC classes and interfaces.

Table 21.4 Java JDBC classes and interfaces

DriverManager	Connection	Statement
PreparedStatement	ResultSet	ResultSetMetaData
CallableStatement	DatabaseMetaData	

21.15 BASIC STEPS

The following basic steps are followed to work with JDBC:

- Loading a Driver

- Making a connection
- Executing an SQL statement

21.16 LOADING A DRIVER

You have to first download an appropriate driver depending upon the database you want to connect. Sun provides a Type 1 driver bundled with the JDK 1.1 or later. Other types of drivers are database-specific and must be downloaded.

The latest version, Type 4 MySQL JDBC driver, can be downloaded from the following site:

```
http://dev.mysql.com/downloads/connector/j/5.1.html
```

Download the .zip or .tar.gz file containing the binary file `mysql-connector-java-5.1.7-bin.jar`.

The latest version Type 4 JDBC driver for Oracle can be downloaded from the following site:

```
http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html
```

Download the appropriate driver depending upon the JDK and Oracle version you are using. For example, the binary driver file for Oracle 9i and JDK 1.4 or later is `ojdbc14.jar`.

Once you have downloaded the appropriate .jar file, put it in Tomcat's `lib` directory and restart the web server.

If you are developing simple Java database applications, put this .jar file in the `CLASSPATH` environment variable.

So far, we have downloaded and installed the JDBC driver. For it to start functioning, an instance of the driver has to be created and registered with the `DriverManager` class so that it can translate the JDBC call to the appropriate database call. The JDBC class `DriverManager` is an important class in the `java.sql` package. It interfaces between the Java application and the JDBC driver. This class manages the set of JDBC drivers installed on the system. It has many other useful methods, some of which will be discussed in Section 21.17.

One way to register a driver with the driver manager is to use the static method `forName` of the Java class `Class` with a driver class name as an argument. For example, the Type 1 driver provided by sun can be instantiated and registered with the driver manager as follows:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

The method `forName` creates an instance of the class whose name is specified as an argument using its default constructor. The instance created in this fashion must register itself with the `DriverManager` class. The .jar file for MySQL contains two driver class files with the name `Driver.class`, one in the `com.mysql.jdbc` package and the other in the `org.gjt.mm.mysql` package. So, you may use any one of the following:

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("org.gjt.mm.mysql.Driver");
```

One can perform this registration procedure by explicitly creating an instance and passing it to the static `registerDriver` method of the `DriverManager` class. The method `registerDriver()`

in turn registers the driver with the driver manager. Some JDBC vendors such as Oracle recommend the latter mechanism.

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

A similar procedure can be followed for other drivers as well. The implementation of the MySQL driver file `com.mysql.jdbc.Driver` looks like this.

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
   static {
      try {
         java.sql.DriverManager.registerDriver(new Driver());
      } catch (SQLException E) {
         throw new RuntimeException("Can't register driver!");
      }
   }
   public Driver() throws SQLException {}
}
```

Since the static block registers the driver with the driver manager automatically, only creating an instance is sufficient. So, one might use the following code as well:

```
new com.mysql.jdbc.Driver();
```

Now, the driver is ready to translate the JDBC call.

21.17 MAKING A CONNECTION

Once a driver is instantiated and registered with the driver manager, the connection to the database can be established using methods provided by the `DriverManager` class. For each connection created, `DriverManager` makes use of the appropriate driver registered to it. The following methods are available on the `DriverManager` class to establish a connection. All methods return a `Connection` object on successful creation of the connection.

```
public static Connection getConnection (String url, String login, String passwd)
public static Connection getConnection (String url)
public static Connection getConnection (String url, Properties)
```

The `Connection` object encapsulates the session/connection to a specific database. It is used to fire SQL statements as well as commit or roll back database transactions. It also allows us to collect useful information about the database dynamically and to write custom applications. Many connections can be established to a single database server or different database servers.

The primary argument that the `getConnection()` method takes is a database URL. This argument identifies a database uniquely. `DriverManager` uses this URL to find a suitable JDBC driver installed earlier, which recognizes the URL and uses this driver to connect to the corresponding database.

The URL always starts with `jdbc:`. The format of the rest of the JDBC URL varies widely for different databases. Some are mentioned in Table 21.4. The format of the MySQL JDBC URL is as follows:

```
jdbc:mysql://[host]:[port]/[database]
```

Here, `host` is the name (or IP address) of the machine running the database at the port number `port` and `database` is a name of a database. Suppose a MySQL database, `test`, is running in a machine, `thinkpad`, at port `3306`, the corresponding URL will be

```
jdbc:mysql://thinkpad:3306/test
```

A database connection can be established using this URL as follows:

```
Connection con = DriverManager.getConnection("jdbc:mysql://thinkpad:3306/test",
"root",  "nbuser");
```

Similarly, the following code segment creates a database connection to the Oracle database `mirora` running in the machine `miroracle` at port `1521`.

```
Connection  con =
DriverManager.getConnection("jdbc:oracle:thin:@miroracle:1521:mirora", "scott",
"tiger");
```

Table 21.5 shows the JDBC URL formats.

Table 21.5 JDBC URL format

MySQL		
Jar file	mysql-connector-java-nn-bin.jar	
Download URL	http://dev.mysql.com/downloads/connector/j/5.1.html	
Driver	com.mysql.jdbc.Driver	
URL format	jdbc:mysql://[host]:[port]/[database]	
Sample URL	jdbc:mysql://thinkpad:3306/test	
	jdbc:mysql://localhost:3306/sample	
Oracle		
Jar file	ojdbc14.jar (Java 1.4) ojdbc15.jar (Java 1.5) ojdbc16.jar (Java 1.6)	
Download URL	http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html	
Driver	oracle.jdbc.driver.OracleDriver	
URL format	jdbc:oracle:[type]:@[host]:[port]:[service] jdbc:oracle:[type]:[host]:[port]:[SID] jdbc:oracle:[type]:[TNSName]	
Sample URL	thin	jdbc:oracle:thin:@miroracle:1521: ORCL_SVC jdbc:oracle:thin:@172.16.4.243:1521: ORCL_SID jdbc:oracle:thin:@(description=(address=(host=localhost)(protocol=tcp)(port=1521))(connect_data=(sid=ORCL))) jdbc:oracle:thin:@TNS-NAME
	Oci	jdbc:oracle:oci:@miroracle:1521: ORCL_SVC jdbc:oracle:oci:@172.16.4.243:1521: ORCL_SID jdbc:oracle:oci:@(description=(address=(host=localhost)(protocol=tcp)(port=1521))(connect_data=(sid=ORCL))) jdbc:oracle:oci:@TNS-NAME

(Contd.)

(Contd.)

Sun JDBC–ODBC Bridge	
Jar file	Bundled with JDK
Driver	Sun.jdbc.odbc.JdbcOdbcDriver
URL format	JDBC:ODBC:[data source name]
Sample URL	JDBC:ODBC:test
DB2	
Jar file	db2jcc.jar
Download URL	http://www-01.ibm.com/software/data/db2/ad/java.html
Driver	Com.ibm.db2.jdbc.net.DB2Driver
URL format	Jdbc:db2://[host]:[port]/[database]
Sample URL	jdbc:db2://172.16.4.243:50000/test
Pervasive	
Jar file	pvjdbc2.jar
Driver	com.pervasive.jdbc.v2.Driver
Download URL	http://www.pervasive.com/developerzone/access_methods/jdbc.asp
URL format	jdbc:pervasive://[host]:[port]/[database]
Sample URL	jdbc:pervasive://thinkpad:1583/sample
PostgreSQL	
Jar file	postgresql-nn.jdbc3.jar
Download URL	http://jdbc.postgresql.org/download.html
Driver	org.postgresql.Driver
URL format	jdbc:postgresql://[host]:[port]/[database]
Sample URL	jdbc:postgresql://[localhost]:[5432]/[test]
JavaDB/Derby	
Jar file	derbyclient.jar
Download URL	http://db.apache.org/derby/derby_downloads.html
Driver	org.apache.derby.jdbc.ClientDriver
URL format	jdbc:derby:net://[host]:[port]/[database]
Sample URL	jdbc:derby:net://[172.16.4.243]:[1527]/[sample]

The second overloaded version of the `getConnection()` method takes only a string argument. This argument must contain URL information, together with other parameters such as user name and password. The parameters are passed as a name–value pair separated by "&" using the same syntax as the HTTP URL. The general syntax of such a URL is as follows:

```
basicURL?param1=value1&param2=value2…
```

Following is an example of such a string argument for the MySQL database.

```
jdbc:mysql://thinkpad:3306/test?user=root&password=nbuser
```

Alternatively, parameters can be put in a `java.util.Properties` object and the object can be passed to the `getConnection()` method. Following is an example using Properties.

```
Connection con = DriverManager.getConnection(url, p);
```

21.18 EXECUTE AN SQL STATEMENT

Once a connection to the database is established, we can interact with the database. The `Connection` interface provides methods for obtaining different statement objects that are used to fire SQL statements via the established connection. The `Connection` object can be used for other purposes such as gathering database information, and committing or rolling back a transaction. The following section describes different types of statement objects and their functionality.

21.19 SQL STATEMENTS

The `Connection` interface defines the following methods to obtain statement objects.

```
Statement createStatement()
Statement createStatement(int resultSetType,
                          int resultSetConcurrency)
Statement createStatement(int resultSetType,
                          int resultSetConcurrency,
                          int resultSetHoldability)

PreparedStatement prepareStatement(java.lang.String)
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                   int resultSetConcurrency)
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                   int resultSetConcurrency,
                                   int resultSetHoldability)

CallableStatement prepareCall(java.lang.String)
CallableStatement prepareCall(String sql,
                              int resultSetType,
                              int resultSetConcurrency)
CallableStatement prepareCall(String sql,
                              int resultSetType,
                              int resultSetConcurrency,
                              int resultSetHoldability)
```

The JDBC `Statement`, `CallableStatement`, and `PreparedStatement` interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

21.19.1 Simple Statement

The `Statement` interface is used to execute static SQL statements. A `Statement` object is instantiated using the `createStatement()` method on the `Connection` object as follows:

```
Statement stmt = con.createStatement();
```

This `Statement` object defines the following methods to fire different types of SQL commands on the database.

`executeUpdate()`

This method is used to execute DDL (CREATE, ALTER, and DROP), DML (INSERT, DELETE, UPDATE, etc.) and DCL statements. In general, if an SQL command changes the database, the `executeUpdate()` method is used. The return value of this method is the number of rows affected.

Assume that `stmt` is a `Statement` object. The following code segment first creates a table named `accounts`.

```
String create = "CREATE TABLE accounts (              "+
                "  accNum      integer  primary key,  "+
                "  holderName  varchar(20),           "+
                "  balance     integer                "+
                ")";
stmt.executeUpdate(create);
```

Once the table is created, data can be inserted into it using the following code segment.

```
String insert = "INSERT INTO accounts VALUES(1,'Uttam K. Roy', 10000)";
stmt.executeUpdate(insert);
insert = "INSERT INTO accounts VALUES(2,'Bibhas Ch. Dhara', 20000)";
stmt.executeUpdate(insert);
```

`executeQuery()`

This is used for DQL statements such as SELECT. Remember, DQL statements only read data from database tables; it cannot change database tables. So, the return value of this method is a set of rows that is represented as a `ResultSet` object.

The result of the `executeQuery` method is stored in an object of type `ResultSet`. This result set object looks very much similar to a table and hence has a number of rows. A particular row is selected by setting a cursor associated with this result set. A cursor is something like a pointer to the rows. Once the cursor is set to a particular row, individual columns are retrieved using the methods provided by the `ResultSet` interface. The cursor is placed before the first row of result set when it is created first. JDBC 1.0 allows us to move the cursor only in the forward direction using the method `next()`. JDBC 2.0 allows us to move the cursor both forward and backward as well as to move to a specified row relative to the current row. These types of result sets are called scrollable result sets, which will be discussed in Section 21.22.

Since the cursor does not point to any row (it points to a position before the first row), users have the responsibility to set the cursor to a valid row to retrieve data from it. To retrieve data from a column, methods of the form `getX()` are used, where `X` is the data type of the column. Following is an example that retrieves information from the `accounts` table created earlier.

```
String query = "SELECT * FROM accounts";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
```

```
        out.println(rs.getString("accNum"));
        out.println(rs.getString("holderName"));
        out.println(rs.getString("balance"));
    }
```

`execute()`

Sometimes, users want to execute SQL statements whose type (DDL, DML, DCL, or DQL) is not known in advance. This may happen particularly when statements are obtained from another program. In that case, users cannot decide which method they should use. In such cases, the `execute()` method is used. It can be used to execute any SQL commands. Since it allows us to execute any SQL commands, the result can either be a `ResultSet` object or an integer. However, how does a user know it? Fortunately, this method returns a Boolean value, which indicates the return type. The return value `true` indicates that the result is a `ResultSet` object, which can be obtained by calling its `getResultSet()` method. On the other hand, if the return value is `false`, the result is an update count, which can be obtained by calling the `getUpdateCount()` method.

The following JSP page takes an arbitrary SQL statement as a parameter and fires this SQL statement on the database. Make sure that the MySQL JDBC driver is in the `/lib` directory under the Tomcat installation directory.

```
<%@page import="java.sql.*"%>
<%
   response.setHeader("Pragma", "no-cache");
   response.setHeader("Cache-Control", "no-cache");
   response.setDateHeader("Expires", -1);
   try {
      String query = request.getParameter("sql");
      if (query != null) {
         new com.mysql.jdbc.Driver();
         String url = "jdbc:mysql://thinkpad:3306/test";
         Connection con = DriverManager.getConnection(url, "root", "nbuser");
         Statement stmt = con.createStatement();
         if (stmt.execute(query) == false) {
            out.println(stmt.getUpdateCount() + " rows affected");
         }
         else {
            ResultSet rs = stmt.getResultSet();
            ResultSetMetaData md = rs.getMetaData();
            out.println("<table border=\"1\"><tr>");
            for (int i = 1; i <= md.getColumnCount(); i++) {
               out.print("<th>" + md.getColumnName(i) + "</th>");
            }
            out.println("</tr>");
            while (rs.next()) {
               out.println("<tr>");
               for (int i = 1; i <= md.getColumnCount(); i++) {
                  out.print("<td>" + rs.getString(i) + "</td>");
               }
               out.println("</tr>");
            }
            out.println("</table>");
            rs.close();
         }
```

```
            stmt.close();
            con.close();
        }
    }
    catch (Exception e) {
        out.println(e);
    }
%>
<form name="sqlForm" method="post">
    SQL statement:<br><input type="text" name="sql" size="50"><br />
    <input type="reset"><input type="submit" value="Execute">
</form>
```

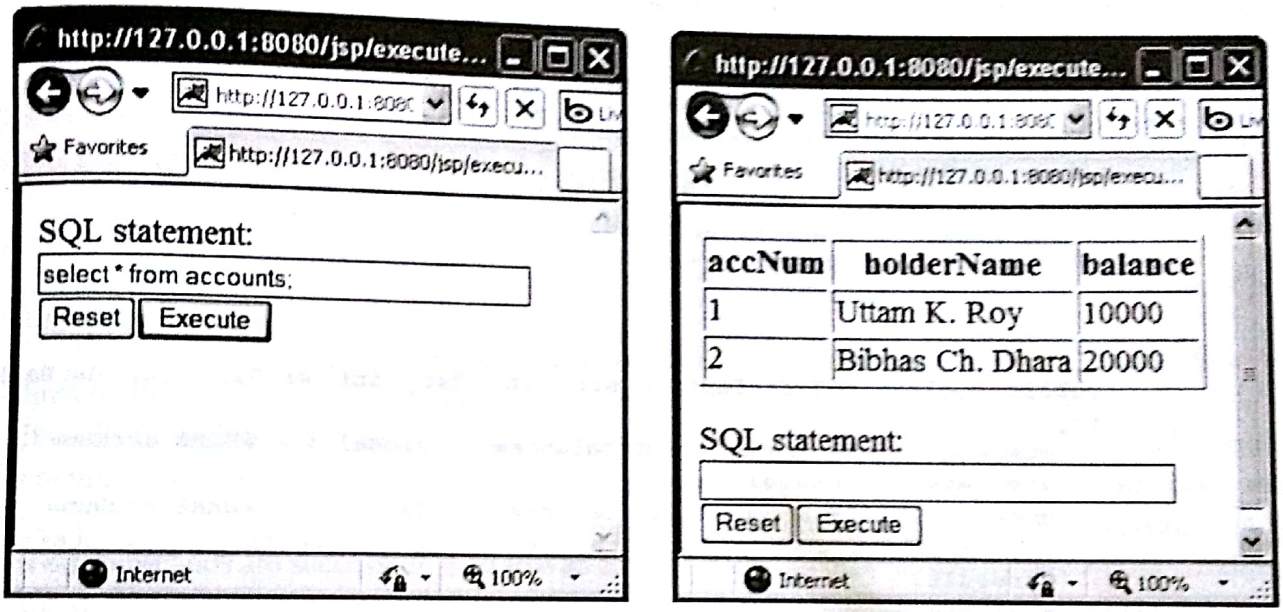


Figure 21.10 Executing SQL statements

Note that this JSP page allows you to fire any valid SQL query, including DML. So, the JSP page must perform user verification before providing this interface.

21.19.2 Atomic Transaction

The database transaction made by the `executeUpdate()` method is committed automatically. This may lead to data inconsistency if a series of related statements are executed. Consider the following simple table for a banking application.

```
accounts(accNum, holderName, balance)
```

The bank manager wants to write a java program to transfer some amount of money `amount` from the source account `src` to the destination account `dst`. The basic task of this program will be to subtract `amount` from the source account balance and add `amount` to the destination account balance. A sample JSP page looks like this:

```
<%@page import="java.sql.*"%>
<%!
    Connection con;
    Statement stmt;
```

```
String query;
public void jspInit() {
   try {
      new com.mysql.jdbc.Driver();
      String url = "jdbc:mysql://thinkpad:3306/test";
      con = DriverManager.getConnection(url, "root", "nbuser");
      stmt = con.createStatement();
   }catch(Exception e) {}
}
public boolean transfer(int src, int dst, int amount) {
   try {
      query = "SELECT balance FROM accounts WHERE accNum=" + src;
      ResultSet rs = stmt.executeQuery(query);
      rs.next();
      int srcBal = rs.getInt("balance") - amount;
      query = "SELECT balance FROM accounts WHERE accNum=" + dst;
      rs = stmt.executeQuery(query);
      rs.next();
      int dstBal = rs.getInt("balance") + amount;
      return doTransfer(src, dst, srcBal, dstBal);
   } catch (SQLException e) {
      return false;
   }
}
public boolean doTransfer(int src, int dst, int srcBal, int dstBal) {
try {
   query = "UPDATE accounts SET balance=" + srcBal + " WHERE accNum=" + src;
   stmt.executeUpdate(query);
   query = "UPDATE accounts SET balance=" + dstBal + " WHERE accNum=" + dst;
   //If anything goes wrong here, destination account will contain wrong
   //result.
   stmt.executeUpdate(query);
   return true;
} catch (SQLException e) {
   return false;
}
}
%>
<%
try {
   int src = Integer.parseInt(request.getParameter("src"));
   int dst = Integer.parseInt(request.getParameter("dst"));
   int amount = Integer.parseInt(request.getParameter("amount"));
   transfer(src, dst, amount);
}catch(Exception e) {out.println(e);}
%>
```

Note that source and destination accounts must be updated atomically. However, if anything goes wrong after updating the source account and before updating the destination account in the `doTransfer()` method, the destination account will hold an incorrect balance.

This problem can be solved using the `autoCommit()` method available on the `Connection` object. First the `autoCommit` flag of the `Connection` object is set to false. At the end of execution of all related statements, the transaction is committed. If anything goes wrong during the execution

of those statements, it can be caught and the transaction gets rolled back accordingly. This way a set of related operations can be performed atomically.

The correct `doTransfer()` method looks like this:

```
public boolean doTransfer(int src, int dst, int srcBal, int dstBal) {
   try {
      con.setAutoCommit(false);
      query = "UPDATE accounts SET balance=" + srcBal + " WHERE accNum=" + src;
      stmt.executeUpdate(query);
      query = "UPDATE accounts SET balance=" + dstBal + " WHERE accNum=" + dst;
      stmt.executeUpdate(query);
      con.commit();
      return true;
   } catch (SQLException e) {
      try {
         con.rollback();
      } catch (SQLException e1) {
      }
      return false;
   }
}
```

`executeBatch()`

This method allows us to execute a set of related commands as a whole. Commands to be fired on the database are added to the `Statement` object one by one using the method `addBatch()`. It is always safe to clear the `Statement` object using the method `clearBatch()` before adding any command to it. Once all commands are added, `executeBatch()` is called to send them as a unit to the database. The DBMS executes the commands in the order in which they were added. Finally, if all commands are successful, it returns an array of update counts. To allow correct error handling, we should always set auto-commit mode to `false` before beginning a batch command.

Following is the method `doTransfer`, rewritten using this mechanism.

```
public boolean doBatchTransfer(int src, int dst, int srcBal, int dstBal) {
   try {
      String query;
      con.setAutoCommit(false);
      stmt.clearBatch();
      query = "UPDATE accounts SET balance=" + srcBal + " WHERE accNum=" + src;
      stmt.addBatch(query);
      query = "UPDATE accounts SET balance=" + dstBal + " WHERE accNum=" + dst;
      stmt.addBatch(query);
      stmt.executeBatch();
      con.commit();
      return true;
   } catch (SQLException e) {
      try {
         con.rollback();
      } catch (SQLException e1) {
      }
      return false;
   }
}
```

Since all the commands are sent as a unit to the database for execution, it improves the performance significantly.

21.19.3 Pre-compiled Statement

When an SQL statement is fired to the database for execution using the `Statement` object the following steps get executed:

- DBMS checks the syntax of the statement being submitted.
- If the syntax is correct, it executes the statement.

DBMS compiles every statement unnecessarily, even if users want to execute the *same* SQL statement repeatedly with different data items. This creates significant overhead, which can be avoided using the `PreparedStatement` object.

A `PreparedStatement` object is created using the `prepareStatement()` method of the `Connection` object. An SQL statement with placeholders (?) is supplied to the method `Connection.prepareStatement()` when a `PreparedStatement` object is created. This SQL statement, together with the placeholders is sent to the DBMS. DMBS, in turn, compiles the statement and if everything is correct, a `PreparedStatement` object is created. This means that a `PreparedStatement` object contains an SQL statement whose syntax has already been checked and hence is called pre-compiled statement. This SQL statement is then fired repeatedly, with placeholders substituted by different data items. Note that `PreparedStatement` is useful only if the same SQL statement is executed repeatedly with different parameters. Otherwise, it behaves exactly like `Statement` and no benefit can be obtained.

The following example creates a `PreparedStatement` object:

```
PreparedStatement ps = con.prepareStatement("INSERT INTO user values(?,?)");
```

The SQL statement has two placeholders, whose values will be supplied whenever this statement is sent for execution. Placeholders are substituted using methods of the form `setX()`, where `X` is the data type of the value used to substitute. These methods take two parameters. The first parameter indicates the index of the placeholder to be substituted and the second one indicates the value to be used for substitution. The following example substitutes the first placeholder with "user1".

```
ps.setString(1, "user1");
```

Consider a file, `question.txt`, which contains questions of the form `question_no:question_string` as follows:

```
1:What is the full form of JDBC?
2:How is a PreparedStatement created?
…
```

The following example inserts questions and their numbers stored in this file in the table `questions`.

```
PreparedStatement ps = con.prepareStatement("INSERT INTO questions
values(?,?)");
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream("question.txt")));
String line = br.readLine();
while (line != null) {
   StringTokenizer st = new StringTokenizer(line, ":");
   String qno = st.nextToken();
   String question = st.nextToken();
   ps.setString(1, qno);
   ps.setString(2, question);
   ps.executeUpdate();
   line = br.readLine();
}
```

`PreparedStatement` has another important role in executing parameterized SQL statements. Consider this solution using the `Statement` object.

```
Statement stmt = con.createStatement();
BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(application.getRealPath("/")+"question.txt")));
String line = br.readLine();
while (line != null) {
   StringTokenizer st = new StringTokenizer(line, ":");
   String qno = st.nextToken();
   String question = st.nextToken();
   String query = "INSERT INTO questions values("+qno+",'"+question+"')";
   stmt.executeUpdate(query);
   line = br.readLine();
}
```

This code segment will work fine, provided the question does not contain characters such as "'". For example, if the file `question.txt` contains a line "3:What's JDBC?", the value of the query will be

```
INSERT INTO questions values(3,'What's JDBC?')
```

This is an invalid query due to the "'" character in the word "What's". If you try, you will get an error message like this:

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error
in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near 's JDBC?')' at line 1
```

`PreparedStatement` can handle this situation very easily, as it treats the entire parameter as input. So, the example given using the `PreparedStatement` will work correctly in this case.

21.19.4 SQL Statements to Call Stored Procedures

JDBC also allows the calling of stored procedures that are stored in the database server. This is done using the `CallableStatement` object. A `CallableStatement` object is created using the `prepareCall()` method on a `Connection` object.

```
CallableStatement prepareCall(String)
```

The method `prepareCall()` takes a primary string parameter, which represents the procedure to be called, and returns a `CallableStatement` object, which is used to invoke stored procedures if the underlying database supports them. Here is an example:

```
String proCall = "{call changePassword(?, ?, ?)}";
CallableStatement cstmt = con.prepareCall(proCall);
```

The variable `cstmt` can now be used to call the stored procedure `changePassword`, which has three input parameters and no result parameter. Whether the ? placeholders are `IN`, `OUT`, or `INOUT` parameters depends on the definition of the stored procedure `changePassword`.

JDBC API allows the following syntax to call stored procedures:

```
{call procedure-name [(?, ?, ...)]}
```

For example, to call a stored procedure with no parameter and no return type, the following syntax is used:

```
{call procedure-name}
```

The following syntax is used to call a procedure that takes a single parameter:

```
{call procedure-name(?)}
```

If a procedure returns a value, the following syntax is used:

```
{? = call procedure-name (?, ?)}
```

Since MySQL procedures are not allowed to return values, the last format is not allowed in MySQL. The web developer must know what stored procedures are available in the underlying database. Before using any stored procedure, one can use the `supportsStoredProcedures()` method on the `DatabaseMetaData` object to verify if the underlying database supports the stored procedure. If it supports, the description of the stored procedures can be obtained using `getProcedures()` on the `DatabaseMetaData` object. Consider the following procedure created in MySQL.

```
CREATE PROCEDURE changePassword(IN loginName varchar(10), IN oldPassword
varchar(10), IN newPassword varchar(10))
BEGIN
  DECLARE old varchar(10);
  SELECT password INTO @old FROM users WHERE login=loginName;
  IF @old = oldPassword THEN
    UPDATE users SET password=newPassword WHERE login=loginName;
  END IF;
END;
```

This procedure changes the password of a specified user in the table `users`. It takes three parameters: the login id of the user whose password has to be changed, the old password, and a new password.

If you are using MySQL command prompt to create a procedure, you may face a problem. Note that the stored procedures use ";" as the delimiter. The default MySQL statement delimiter is also ";". This would make the SQL in the stored procedure syntactically invalid. The solution is to temporarily change the command-line utility delimiter using the following command;

```
DELIMITER //
```

In the end, change the delimiter to ";" if necessary.

The `IN` parameters are passed to a `CallableStatement` object using methods of the form `setXxx()`. For example, `setFloat()` and `setBoolean()` methods are used to pass `float` and `boolean` values, respectively.

The following code segment illustrates how to call this procedure.

```
<%@page import="java.sql.*"%>
<%
   try {
      new com.mysql.jdbc.Driver();
      String url = "jdbc:mysql://thinkpad:3306/test";
      Connection con = DriverManager.getConnection(url, "root", "nbuser");
      String proCall = "{call changePassword(?, ?, ?)}";
      CallableStatement cstmt = con.prepareCall(proCall);
      String login = request.getParameter("login");
      String oldPassword = request.getParameter("oldPassword");
      String newPassword = request.getParameter("newPassword");
      cstmt.setString(1, login);
      cstmt.setString(2, oldPassword);
      cstmt.setString(3, newPassword);
      if (cstmt.executeUpdate() > 0) {
         out.println("Password changed successfully");
      } else {
         out.println("Couldn't change password");
      }
      cstmt.close();
      conn.close();
   } catch (Exception e) {
      out.println(e);
   }
%>
```

The `CallableStatement` object also allows batch update exactly like `PreparedStatement`. Following is an example:

```
String proCall = "{call changePassword(?, ?, ?)}";
CallableStatement cstmt = con.prepareCall(proCall);

cstmt.setString(1, "user1");
cstmt.setString(2, "user1");
cstmt.setString(3, "pass1");
cstmt.addBatch();

cstmt.setString(1, "user2");
cstmt.setString(2, "user2");
cstmt.setString(3, "pass2");
cstmt.addBatch();

int [] updateCounts = cstmt.executeBatch();
```

This example illustrates how to use batch update facility to associate two sets of parameters with a `CallableStatement` object.

21.20 RETRIEVING RESULT

A table of data is represented in the JDBC by the `ResultSet` interface. The `ResultSet` objects are usually generated by executing the SQL statements that query the database. A pointer points to a particular row of the `ResultSet` object at a time. This pointer is called *cursor*. The cursor is positioned before the first row when the `ResultSet` object is generated. To retrieve data from a row of `ResultSet`, the cursor must be positioned at the row. The `ResultSet` interface provides methods to move this cursor.

next()

- This method on the `ResultSet` object moves the cursor to the next row of the result set.
- It returns true/false depending upon whether there are more rows in the result set.

Since the `next()` method returns false when there are no more rows in the `ResultSet` object, it can be used in a while loop to iterate through the result set as follows:

```
String query = "SELECT * from users";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
   //process it
}
```

The `ResultSet` interface provides reader methods for retrieving column values from the row pointed to by the cursor. These have the form `getXxx()`, where `Xxx` is the name of the data type of the column. For example, if data types are string and int, the name of the reader methods are `getString()` and `getInt()`, respectively.

Values can be retrieved using either the column index or the name of the column. Using the column index, in general, is more efficient. The column index starts from 1. The following example illustrates how to retrieve data from a `ResultSet` object.

```
String query = "SELECT * from users";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
   String login = rs.getString("login");
   String password = rs.getString("password");
   System.out.println(login+"\t"+password);
}
```

21.21 GETTING DATABASE INFORMATION

Sometimes, it is necessary to know the capabilities of DataBase Management System (DBMS) before dealing with it. This is because different DBMSs often provide different features, implement them differently, and also use different data types. Moreover, the driver may also implement additional features on top of the DBMS. The `DatabaseMetaData` interface provides methods to collect comprehensive information about a DBMS. We can discover features a DBMS supports and develop our application accordingly. For example, before creating a table, one may want to

know what data types are supported by this DBMS. User may also want to know whether the underlying DBMS supports batch update.

A `DatabaseMetaData` object is obtained using the `getMetaData()` method on the `Connection` object as follows:

```
DatabaseMetaData md = con.getMetaData();
```

We can then use various methods on this `DatabaseMetaData` object to collect the required information about the DBMS. Following is a list of commonly used methods:

`getDatabaseMetaData()`

Returns the `DatabaseMetaData` object, which contains detailed information about the underlying database. Some important methods of `DatabaseMetaData` are

`String getSQLKeywords()`

Returns keywords available

`getDatabaseProductName()`

Returns the name of the manufacturer

`getDatabaseProductVersion()`

Returns the current version

`getDriverName()`

Returns driver used

The following JSP page retrieves most of the MySQL database information.

```
<%@page import="java.sql.*, java.lang.reflect.*"%>
<%
   new com.mysql.jdbc.Driver();
   String url = "jdbc:mysql://thinkpad:3306/test";
   Connection con = DriverManager.getConnection(url, "root", "nbuser");
   DatabaseMetaData md = con.getMetaData();
   Method[] methods = md.getClass().getMethods();
   Object[] param = new Object[0];
   out.println("<table border=\"1\">");
   for (int i = 0; i < methods.length; i++) {
      if (methods[i].getParameterTypes().length == 0) {
         if (methods[i].getReturnType() == Boolean.TYPE ||
methods[i].getReturnType() == String.class) {
            out.println("<tr>");
            out.println("<td>"+methods[i].getName() + "</td>");
            out.println("<td>" + methods[i].invoke(md, param)+"</td>");
            out.println("</tr>");
         }
      }
   }
   out.println("</table>");
%>
```

The result of this page is shown in Table 21.9.

21.22 SCROLLABLE AND UPDATABLE RESULTSET

The result set returned so far by a query can be navigated in one direction (forward). Moreover the data the result sets contain are read-only. Any change to the result set does not affect the actual database.

Result sets can be *scrollable* in the sense that the cursor can be moved backward and forward. Additionally, a result set can be *updatable*, such that any change to the result set reflects in the database immediately. A result set can be scrollable as well as updatable. *Note that scrollable and updatable result sets incur significant overhead. So, such result sets should be created if the underlying application performs scrolling.*

In addition to *scrollability* and *updatability*, another important concept, called *sensitivity*, is defined. Sensitivity broadly answers the following question:

```
Can a result set see the changes that are made to the underlying database?
```

If a result cannot see any changes, it is said to be insensitive. Otherwise, the sensitivity of a result set is defined with respect to the database operation as well as the operating party. For example, a result set is said to be sensitive to update if it can see any update operation made on the underlying database. The sensitivity rules are shown in Table 21.6.

The `createStatement()` and `prepareStatement()` methods take extra parameters that specify the type of result returned by subsequent execution of SQL statements. The prototype of the `createStatement()` method to generate a scrollable and updatable result set is as follows:

```
Statement createStatement(int resultSetType, int resultSetConcurrency)
```

Here, scrollability and updatability are controlled by the parameters `resultSetType` and `resultSetConcurrency`, respectively.

21.22.1 Scrollability Type

The parameter `resultSetType` can assume the following static integer constants defined in `ResultSet`. Their meaning is as follows:

- `TYPE_FORWARD_ONLY`
  If this constant is used, the cursor starts at the first row and can only move forward.

- `TYPE_SCROLL_INSENSITIVE`
  All cursor positioning methods are enabled; the result set does not reflect changes made by others in the underlying table.

- `TYPE_SCROLL_SENSITIVE`
  All cursor positioning methods are enabled; the result set reflects changes made by others in the underlying table.

The visibility of internal and external changes to scrollable `ResultSet` in Oracle JDBC is shown in Table 21.6.

Table 21.6 Visibility of internal and external changes to scrollable result set in oracle JDBC

Scroll Type		TYPE_FORWARD_ONLY	TYPE_SCROLL_SENSITIVE	TYPE_SCROLL_INSENSITIVE
Internal	DELETE	No	Yes	Yes
	UPDATE	Yes	Yes	Yes
	INSERT	No	No	No
External	DELETE	No	No	No
	UPDATE	No	Yes	No
	INSERT	No	No	No

21.22.2 Concurrency Type

The parameter `resultSetConcurrency` can assume the following static integer constants defined in `ResultSet`. The following is a brief description:

- `CONCUR_READ_ONLY`

  The result set is not updatable.

- `CONCUR_UPDATABLE`

  Rows can be added and deleted; columns can be updated and are visible to others.

21.22.3 Examples

The following example creates a `Statement` object, whose methods will return scrollable, update insensitive, and read-only result sets.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
```

The `ResultSet` object `rs` is now scrollable, but update insensitive.

Originally, result sets could be navigated only in one direction (forward) and starting at only one position (the first row). In JDBC 2.0, the row pointer can be manipulated as if it were an array index. Some of the important methods available on the scrollable `ResultSet` object are shown in Table 21.7. The following examples demonstrate how to navigate a scrollable result set:

- Move the cursor forward by one row.

```
rs.next();
//or
rs.relative(1);
```

- Move the cursor backward by one row.

```
rs.previous();
//or
rs.relative(-1);
```

- Set the cursor before the first row.

```
rs.beforeFirst();
```

- Set the cursor after the last row.

```
rs.afterLast();
```

- Set the cursor at the first row (row 1).

```
rs.first();
//or
rs.absolute(1);
```

- Set the cursor at the last row.

```
rs.last();
//or
rs.absolute(-1);
```

- Set the cursor at the second row.

```
rs.absolute(2);
```

- Set the cursor at the second last row.

```
rs.absolute(-2);
```

- Move the cursor forward six rows from the current position.

```
rs.relative(6);
//Sets the cursor after the last row, if it goes beyond the last row
```

- Move the cursor backward four rows from the current position.

```
rs.relative(-4);
//Sets the cursor before the first row, if it goes before the first row
```

Table 21.7 Scrollable ResultSet methods

Method	Description
next()	Advances cursor to the next row
previous()	Moves cursor back one row
first()	Sets cursor to the first row
last()	Sets cursor to the last row
beforeFirst()	Sets cursor just before the first row
afterLast()	Sets cursor just after the last row
absolute(int rowNumber)	Sets the cursor to the specified row number. +ve and –ve numbers indicate positions relative to the position before first row and after last row, respectively. For example, 1 and –1 represent first and last row, respectively.
relative(int rows)	Forwards or reverses cursor the specified number of rows relative to the current position. +ve number indicates forwarding and –ve number indicates reversing. For example, relative(1) forwards the cursor one position, which is equivalent to next(). Similarly, relative(–1) moves the cursor one position back and is equivalent to previous(). It throws an SQLException if cursor points before the first row or after the last row.
moveToInsertRow()	Sets the cursor to a special row called "insert row" and remembers the current position before moving the cursor.
moveToCurrentRow()	Sets the cursor to the row from where the cursor was moved to "insert row" using moveToInsertRow().

The following example creates a `Statement` object, whose methods will return scrollable as well as external-update-insensitive and updatable result sets.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
```

To update a row in a database table, the following steps are used:

- Obtain an updatable result set.
- Move the cursor to the row to be updated using positioning methods available on the `ResultSet` object.
- Update the value of one or more columns in that row using the `updateXxx()` method on the `ResultSet` object, where `Xxx` is the data type of the column.
- Finally, update the database table using the `updateRow()` method.

The following example demonstrates how to update, insert, or delete a row from a database table using an updatable result set.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

//Updating existing row
rs.absolute(4);
rs.updateString("password","newPassword");
rs.updateRow();

//inseting new row
rs.moveToInsertRow();
rs.updateString(1, "anik");
rs.updateString(2, "anik123");
rs.insertRow();

//Deleting a row
rs.deleteRow()
```

The following JSP page changes the password of a specified user using the updatable result set.

```
<%@ page import="java.sql.*" %>
<%
    try {
        String login = request.getParameter("login");
        String oldPassword = request.getParameter("oldPassword");
        String newPassword = request.getParameter("newPassword");

        new com.mysql.jdbc.Driver();
        String url = "jdbc:mysql://thinkpad:3306/test";
        Connection con = DriverManager.getConnection(url, "root", "nbuser");
        Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

```
            String query = "SELECT * FROM users WHERE login='" + login + "'";
            ResultSet rs = stmt.executeQuery(query);    //rs contains one row
            rs.next();                                  //set cursor at first row
            String password = rs.getString("password");
            if (password.equals(oldPassword)) {
               System.out.println(password);
               rs.updateString("password", newPassword); //update the password column
               rs.updateRow();                           //update database table
            }
         } catch (Exception e) {out.println(e); }
      %>
```

The following example populates the table questions by inserting questions into the updatable result set.

```
<%@ page import="java.sql.*, java.io.*, java.util.*" %>
<%
   try {
      new com.mysql.jdbc.Driver();
      String url = "jdbc:mysql://thinkpad:3306/test";
      Connection con = DriverManager.getConnection(url, "root", "nbuser");
      Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
      ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
      BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(application.getRealPath("/")+"question.txt")));
      String line = br.readLine();
      while (line != null) {
         StringTokenizer st = new StringTokenizer(line, ":");
         String qno = st.nextToken();
         String question = st.nextToken();

         rs.moveToInsertRow();
         rs.updateString(1, qno);
         rs.updateString(2, question);
         rs.insertRow();
         line = br.readLine();
      }
      br.close();
   } catch (Exception e) {out.println(e); }
%>
```

The following example creates a `Statement` object, whose methods will return scrollable as well as external-update-sensitive and updatable result sets.

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
```

In this case, if the database table is updated, it is reflected in the result set. Before retrieving data from a row, you should invoke the `refreshRow()` method of the `ResultSet` object so that it contains the updated row. The following JSP page shows how to use an updatable result set.

```
<%@page import="java.sql.*"%>
<%!
```

```
    Connection con;
    Statement stmt;
    ResultSet rs;
    String query;
    public void jspInit() {
       try {
          new com.mysql.jdbc.Driver();
          String url = "jdbc:mysql://thinkpad:3306/test";
          con = DriverManager.getConnection(url, "root", "nbuser");
          Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
          query = "SELECT * FROM users";
          rs = stmt.executeQuery(query);
          System.out.println("loaded");

       }catch(Exception e) {}
    }
%>
<table border="1">
<tr><th>Login name</th><th>Password</th></tr>
<%
    try {
       response.setHeader("Pragma", "no-cache");
       response.setHeader("Cache-Control", "no-cache");
       response.setDateHeader("Expires", -1);

       rs.beforeFirst();
       while(rs.next()) {
          rs.refreshRow();
          out.println("<tr><td>" + rs.getString("login") + "</td>");
          out.println("<td>" + rs.getString("password")+"</td></tr>");
       }
    }catch(Exception e) {out.println(e);}
%>
</table>
```

This JSP page creates the `ResultSet` when this page is requested for the first time. The `ResultSet` is created in the `jspInit()` method. Consequently, it becomes an instance variable. For subsequent requests, it simply uses the `ResultSet` variable.

21.23 RESULT SET METADATA

The `ResultSetMetaData` object is used to retrieve information about the types and properties of the columns and other meta information about a `ResultSet` object. This is sometime very useful, if you do not know much about the underlying database table. A `ResultSetMetaData` object is obtained using the `getMetaData()` method on the `ResultSet` object as follows:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
ResultSetMetaData rsmd = rs.getMetaData();
```

It can be used to get some useful information such as number of rows, number of columns, column names, and their type. Following are some commonly used methods on the `ResultSetMetaData` object:

`int getColumnCount()`
Returns the number of columns in the result

`String getColumnName(int)`
Returns the name of a column in a result set. Requires an integer argument indicating the position of the column within the result set

`int getColumnType(int)`
Returns the type of the specified column in the form of `java.sql.Types`

`String getColumnTypeName(int)`
Returns the type of the specified column as a string

`String getColumnClassName(int)`
Returns the fully qualified Java type name of the specified column

`int getPrecision(int)`
Returns the number of decimal positions

`int getScale(int)`
Returns the number of digits after the decimal position

`String getTableName(int)`
Returns the name of the column's underlying table

`int isNullable(int)`
Returns a constant indicating whether the specified column can have a NULL value

The following JSP page shows how to retrieve meta information from a `ResultSet` object.

```
<%@page import="java.sql.*, java.lang.reflect.*"%>
<%
   try {
      Class.forName("org.gjt.mm.mysql.Driver");
      String url = "jdbc:mysql://localhost:3306/test";
      Connection conn = DriverManager.getConnection(url, "root", "nbuser");
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
      ResultSetMetaData rsmd = rs.getMetaData();
      Object obj[] = new Object[1];
      Method[] methods = rsmd.getClass().getDeclaredMethods();
      out.println("<table border=\"0\"><tr><td>Method Name</td>");
      for(int j = 0; j < rsmd.getColumnCount(); j++)
         out.println("<td>" + rsmd.getColumnName(j+1) + "</td>");
      out.println("</tr>");
      for (int i = 0; i < methods.length; i++) {
      if(Modifier.isPublic(methods[i].getModifiers()))
            if (methods[i].getParameterTypes().length == 1) {
               if(!methods[i].getName().equals("isWrapperFor"))
                  if(!methods[i].getName().equals("unwrap")) {
                     out.print("<tr><td>" + methods[i].getName() + "</td>");
                     for(int j=0;j<rsmd.getColumnCount();j++) {
                        obj[0] = new Integer(j+1);
                        out.print("<td>" + methods[i].invoke(rsmd,obj) +
"</td>");
```

```
                    }
                    out.println("</tr>");
                }
            }
        }
        out.println("<table>");
    }catch(Exception e) {e.printStackTrace();}
%>
```

A sample result for MySQL database is shown in Table 21.8.

Table 21.8 Result set metadata

Method Name	Qno	Question
isWritable	True	True
isCaseSensitive	False	False
getPrecision	11	200
getColumnDisplaySize	11	200
getTableName	Questions	Questions
getColumnLabel	Qno	Question
isAutoIncrement	False	False
getCatalogName	test	Test
getColumnClassName	java.lang.Integer	java.lang.String
getColumnType	4	12
getColumnTypeName	INT	VARCHAR
getScale	0	0
getSchemaName		
isCurrency	false	False
isDefinitelyWritable	true	True
isNullable	0	1
isSearchable	true	True
isSigned	true	False
getColumnCharacterEncoding	null	Null
getColumnCharacterSet	US-ASCII	Cp1252
isReadOnly	false	False
getColumnName	qno	Question

Table 21.9 Database metadata

Method Name	Return value
autoCommitFailureClosesAllResultSets	False
getDriverName	MySQL-AB JDBC Driver
supportsTransactions	True
getDriverVersion	mysql-connector-java-5.1.7 (Revision: ${svn.Revision})
getIdentifierQuoteString	`
allProceduresAreCallable	False

(Contd.)

allTablesAreSelectable	False
dataDefinitionCausesTransactionCommit	True
dataDefinitionIgnoredInTransactions	false
doesMaxRowSizeIncludeBlobs	true
getCatalogSeparator	.
getCatalogTerm	database
getDatabaseProductName	MySQL
getDatabaseProductVersion	5.0.51b-community-nt
getExtraNameCharacters	#@
getNumericFunctions	ABS,ACOS,ASIN,ATAN,ATAN2,BIT_COUNT,CEILING,COS,COT,DEGREES,EXP,FLOOR,LOG,LOG10,MAX,MIN,MOD,PI,POW,POWER,RADIANS,RAND,ROUND,SIN,SQRT,TAN,TRUNCATE
getProcedureTerm	PROCEDURE
getSQLKeywords	ACCESSIBLE,ANALYZE,ASENSITIVE,BEFORE,BIGINT,BINARY,BLOB,CALL,CHANGE,CONDITION,DATABASE,DATABASES,DAY_HOUR,DAY_MICROSECOND,DAY_MINUTE,DAY_SECOND,DELAYED,DETERMINISTIC,DISTINCTROW,DIV,DUAL,EACH,ELSEIF,ENCLOSED,ESCAPED,EXIT,EXPLAIN,FLOAT4,FLOAT8,FORCE,FULLTEXT,HIGH_PRIORITY,HOUR_MICROSECOND,HOUR_MINUTE,HOUR_SECOND,IF,IGNORE,INFILE,INOUT,INT1,INT2,INT3,INT4,INT8,ITERATE,KEYS,KILL,LEAVE,LIMIT,LINEAR,LINES,LOAD,LOCALTIME,LOCALTIMESTAMP,LOCK,LONG,LONGBLOB,LONGTEXT,LOOP,LOW_PRIORITY,MEDIUMBLOB,MEDIUMINT,MEDIUMTEXT,MIDDLEINT,MINUTE_MICROSECOND,MINUTE_SECOND,MOD,MODIFIES,NO_WRITE_TO_BINLOG,OPTIMIZE,OPTIONALLY,OUT,OUTFILE,PURGE,RANGE,READS,READ_ONLY,READ_WRITE,REGEXP,RELEASE,RENAME,REPEAT,REPLACE,REQUIRE,RETURN,RLIKE,SCHEMAS,SECOND_MICROSECOND,SENSITIVE,SEPARATOR,SHOW,SPATIAL,SPECIFIC,SQLEXCEPTION,SQL_BIG_RESULT,SQL_CALC_FOUND_ROWS,SQL_SMALL_RESULT,SSL,STARTING,STRAIGHT_JOIN,TERMINATED,TINYBLOB,TINYINT,TINYTEXT,TRIGGER,UNDO,UNLOCK,UNSIGNED,USE,UTC_DATE,UTC_TIME,UTC_TIMESTAMP,VARBINARY,VARCHARACTER,WHILE,X509,XOR,YEAR_MONTH,ZEROFILL
getSchemaTerm	
getSearchStringEscape	\
getStringFunctions	ASCII,BIN,BIT_LENGTH,CHAR,CHARACTER_LENGTH,CHAR_LENGTH,CONCAT,CONCAT_WS,CONV,ELT,EXPORT_SET,FIELD,FIND_IN_SET,HEX,INSERT,INSTR,LCASE,LEFT,LENGTH,LOAD_FILE,LOCATE,LOCATE,LOWER,LPAD,LTRIM,MAKE_SET,MATCH,MID,OCT,OCTET_LENGTH,ORD,POSITION,QUOTE,REPEAT,REPLACE,REVERSE,RIGHT,RPAD,RTRIM,SOUNDEX,SPACE,STRCMP,SUBSTRING,SUBSTRING,SUBSTRING,SUBSTRING,SUBSTRING_INDEX,TRIM,UCASE,UPPER

(Contd.)

(Contd.)

getSystemFunctions	DATABASE,USER,SYSTEM_USER,SESSION_USER,PASSWORD,ENCRYPT,LAST_INSERT_ID,VERSION
getTimeDateFunctions	DAYOFWEEK,WEEKDAY,DAYOFMONTH,DAYOFYEAR,MONTH,DAYNAME,MONTHNAME,QUARTER,WEEK,YEAR,HOUR,MINUTE,SECOND,PERIOD_ADD,PERIOD_DIFF,TO_DAYS,FROM_DAYS,DATE_FORMAT,TIME_FORMAT,CURDATE,CURRENT_DATE,CURTIME,CURRENT_TIME,NOW,SYSDATE,CURRENT_TIMESTAMP,UNIX_TIMESTAMP,FROM_UNIXTIME,SEC_TO_TIME,TIME_TO_SEC
isCatalogAtStart	true
locatorsUpdateCopy	true
nullPlusNonNullIsNull	true
nullsAreSortedAtEnd	false
nullsAreSortedAtStart	false
nullsAreSortedHigh	false
nullsAreSortedLow	true
storesLowerCaseIdentifiers	true
storesLowerCaseQuotedIdentifiers	true
storesMixedCaseIdentifiers	false
storesMixedCaseQuotedIdentifiers	false

supportsExpressionsInOrderBy	True
supportsExtendedSQLGrammar	False
supportsFullOuterJoins	False
supportsGetGeneratedKeys	True
supportsGroupBy	True
supportsGroupByBeyondSelect	True
supportsGroupByUnrelated	True
supportsIntegrityEnhancementFacility	False
supportsLikeEscapeClause	true
supportsLimitedOuterJoins	true
supportsMinimumSQLGrammar	true
supportsMixedCaseIdentifiers	false
supportsMixedCaseQuotedIdentifiers	false
supportsMultipleOpenResults	true
supportsMultipleResultSets	false
supportsMultipleTransactions	true
supportsNamedParameters	false
supportsNonNullableColumns	true
supportsOpenCursorsAcrossCommit	false
supportsOpenCursorsAcrossRollback	false
supportsOpenStatementsAcrossCommit	false
supportsOpenStatementsAcrossRollback	false
supportsOrderByUnrelated	false
supportsOuterJoins	true
supportsPositionedDelete	false
supportsPositionedUpdate	false
supportsSavepoints	true
supportsSchemasInDataManipulation	false
supportsSchemasInIndexDefinitions	false
supportsSchemasInPrivilegeDefinitions	false
supportsSchemasInProcedureCalls	false
supportsSchemasInTableDefinitions	false
supportsSelectForUpdate	true
supportsStatementPooling	false
supportsStoredFunctionsUsingCallSyntax	true

(Contd.)

supportsSubqueriesInIns	true
supportsSubqueriesInQuantifieds	true
supportsTableCorrelationNames	true
supportsUnion	true
supportsUnionAll	true
usesLocalFilePerTable	false
usesLocalFiles	false
providesQueryObjectGenerator	false
getURL	jdbc:mysql://thinkpad:3306/test
isReadOnly	false
getUserName	root@localhost
toString	com.mysql.jdbc.JDBC4DatabaseMetaData@5bf624

KEYWORDS

Actions: JSP actions are XML tags that can be used in a JSP page to use functions provided by the JSP engine.

Atomic Transaction: A transaction that either does not occur or occurs completely without any interleaving.

Bean: JavaBeans are reusable Java compon... us to separate b...

Implicit Objects: A web container allows us to directly access many useful objects defined in the `_jspService()` method of the JSP page's underlying servlet. These objects ... implicit objects.

Template text: The static HTML/XML components in a JSP page.

Translation: The procedure for converting a JSP page to its servlet source code.

Updatable Result Set: A result set that can be used directly to modify original database tables.

Web container: JSP pages run under the web server called web container.

SUMMARY

JSP allows us to directly embed Java code in the HTML pages. To process JSP pages, a JSP engine is needed. It is interesting to note that what we know as the "JSP engine" is nothing but a specialized servlet that runs under the supervision of the servlet engine. Commonly used JSP engines include Apache's Tomcat, Java Web Server, IBM's WebSphere, etc.

JSP technology was developed on top of servlet technology. The JSP pages are finally converted to servlets automatically.

A JSP page basically consists of static HTML/XML components called template text as well as JSP constructs. JSP constructs consist of directives, declarations, expressions, scriptlets, and actions.

A JSP page may contain instructions to be used by the JSP container to indicate how this page is interpreted and executed. Those instructions are called directives.

Expressions are used to insert usually small pieces of data in a JSP page without using `out.print()` or `out.write()` statements.

You can insert an arbitrary piece of Java code using JSP scriptlets construct in a JSP page.

JSP declarations are used to declare one or more variables, methods, or inner classes that can be used later in the JSP page.

JSP actions are XML tags that can be used in a JSP page to use functions provided by the JSP engine.

In a JSP page, objects may be created using directives, actions, or scriptlets. Every object created in a JSP page has a scope. The scope of a JSP object is defined as the availability of that object for use from a particular place of the web application. There are four object scopes: page, request, session, and application.

A web container allows us to directly access many useful objects defined in the `_jspService()` method of the JSP page's underlying servlet. These objects are called implicit objects as they are instantiated automatically. The implicit objects contain information about request, response, session, configuration, etc.

Variables, methods, and classes can be declared in a JSP page. If they are declared in the declaration section, they become part of the class.

JSP tag extension allows us to define and use custom tags in a JSP page exactly like other HTML/XML tags, using Java code.

JavaBeans are reusable Java components called beans which allow us to separate business logic from presentation logic. There are three action elements that are used to work with beans. A JSP action element `<jsp:useBean>` instantiates a JavaBean object into the JSP page. The `<jsp:setProperty>` action tag assigns a new value to the specified property of the specified bean object. The `<jsp:getProperty>` action element retrieves the value of the specified property of the specified bean object.

There are many ways to track sessions in JSP. Hidden fields may be used to send information back and forth between server and client to track sessions, without affecting the display. This is another simple but elegant method to track sessions and is widely used. Cookies are also used to track sessions. JSP technology (and its underlying servlet technology) provides a higher level approach for session tracking.

Most of the web applications need access to databases in the backend. Java DataBase Connectivity (JDBC) allows us to access databases through Java programs. A Java class that provides interfaces to a specific database is called JDBC driver. Each database has its own set of JDBC drivers. JDBC drivers are classified into four categories depending upon the way they work. The following basic steps are followed to work with JDBC: Loading a Driver, Making a connection, Execute SQL statement.

The `Statement` interface is used to execute static SQL statements. JDBC also allows calling stored procedures that are stored in the database server. This is done using the `CallableStatement` object. A table of data is represented in JDBC by the `ResultSet` interface. Result sets can be scrollable in the sense that the cursor can be moved backward and forward. The `DatabaseMetaData` interface provides methods to collect comprehensive information about a DBMS.

WEB RESOURCES

`http://java.sun.com/products/jsp/`
JavaServer Pages Technology

`http://java.sun.com/products/jsp/syntax/2.0/syntaxref20.html`
Java Server Pages (JSP) v2.0 Syntax Reference

`http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html`
Tag Libraries Tutorial (v. 1.0)

`http://en.wikipedia.org/wiki/Java_Server_Pages`
JavaServer Pages

`http://www.visualbuilder.com/jsp/tutorial/`
JSP Tutorial Home

`http://java.sun.com/products/jsp/archive.html`
JavaServer Pages Specification 1.0

`http://tomcat.apache.org/tomcat-6.0-doc/index.html`
Apache Tomcat 6.0, Documentation Index

`http://tomcat.apache.org/tomcat-5.5-doc/jasper-howto.html`
Apache Tomcat 6.0, Jasper 2 JSP Engine How To

`http://tomcat.apache.org/tomcat-6.0-doc/config/index.html`
Apache Tomcat Configuration Reference

EXERCISES

Multiple Choice Questions

1. What is the full form of JSP?
   (a) Java Servlet Pages (b) Java Server Pages
   (c) Java Small Pages (d) Java Special Pages

2. Which of the following statements is true?
   (a) JSP and servlets are completely different technologies.
   (b) Servlets are built on JSP technology and all servlets are ultimately converted to JSP pages.
   (c) Servlet is a client-side technology, whereas JSP is server-side technology.
   (d) JSPs are built on servlet technology and all JSPs are ultimately translated to servlets.

3. What is the advantage of using RequestDispatcher over sendRedirect() to forward a request to another resource?
   (a) The RequestDispatcher does not use the reflection API.
   (b) sendRedirect() is no longer available in the current servlet API.
   (c) The RequestDispatcher does not require a round trip to the client, and thus is more efficient and allows the server to maintain request state.
   (d) sendRedirect() is not a cross-web server mechanism.

4. In which one of the following cases is the scriptlet better suited?
   (a) Code that handles cookies
   (b) Code that deals with logic that relates to database access
   (c) Code that deals with session management
   (d) Code that deals with logic that is common across requests

5. Which of the following handles a request first in a page-centric approach?
   (a) A JSP page (b) A session manager
   (c) A servlet (d) A JavaBean

6. Which of the following can be included using the JSP include action?
   (a) Servlet (b) Another JSP
   (c) Plain text file (d) All of the above

7. Which of the following is used to read parameters from a JSP page?
   (a) <jsp:getParam/> action
   (b) <jsp:param> action
   (c) request.getParameter() method
   (d) <jsp:readParam/> action

8. Which of the following is not a method of JSP's servlet?
   (a) _jspService() (b) jspDestroy()
   (c) jspService() (d) jspInit()

9. Which of the following JSP actions is used to include a file in another file?
   (a) <jsp:import> (b) <jsp:include>
   (c) <jsp:read> (d) <jsp:get>

10. What is a JSP page translated into?
   (a) CGI (b) Servlet
   (c) Applet (d) JavaBean

11. Which of the following is not true regarding cookies?
   (a) Cookie class has a two argument constructor
   (b) The response object is used to add a cookie
   (c) The request object is used to get the cookie
   (d) The request object is used for creating cookies

12. Which of the following scopes does the implicit object exception have?
   (a) page (b) request
   (c) session (d) application

13. Which of the following tags is used to override the JSP file's initialization method?
   (a) <%= %> (b) <%@ %>
   (c) <% %> (d) <%! %>

14. Which of the following scopes is not valid with respect to JavaBean in JSP?
   (a) request (b) response
   (c) session (d) application

15. Which of the following statements is true regarding HttpServletResponse.sendRedirect()?
   (a) Server itself redirects the current request
   (b) sendRedirect() executes on the client
   (c) Server sends a redirection instruction to the client
   (d) Server drops the current request

16. Which of the following tags is used for scriptlets?
   (a) <%= %> (b) <%@ %>
   (c) <% %> (d) <%! %>

17. Which of the following attributes of the page directive is used to indicate that the current page is an error page?
   (a) errorPage (b) isErrorPage
   (c) anErrorPage (d) pageError

18. Which of the following tags is used for expressions?
   (a) <%= %> (b) <%@ %>
   (c) <% %> (d) <%! %>

19. Which packages contain the JDBC classes?
   (a) java.db.sql and javax.db.sql
   (b) java.jdbc and javax.jdbc
   (c) java.db and javax.db
   (d) java.sql and javax.sql

20. Which of the following drivers converts JDBC calls into network protocol, to communicate with the database management system directly?
   (a) Type 1 driver (b) Type 2 driver
   (c) Type 3 driver (d) Type 4 driver

21. Which of the following types of objects is used to execute parameterized queries?
   (a) ParameterizedStatement
   (b) Statement
   (c) PreparedStatement
   (d) All of the above

22. Which of the following methods on the Statement object is used to execute DML statements?
   (a) executeInsert() (b) execute()
   (c) executeQuery() (d) executeDML()

23. Which type of driver makes a JDBC–ODBC bridge?
   (a) Type 1 driver (b) Type 2 driver
   (c) Type 3 driver (d) Type 4 driver

24. What is the meaning of ResultSet. TYPE_SCROLL_INSENSITIVE
   (a) This means that the ResultSet is insensitive to scrolling.
   (b) This means that the ResultSet is sensitive to scrolling, but insensitive to changes made by others.
   (c) This means that the ResultSet is insensitive to scrolling and insensitive to changes made by others.
   (d) This means that the ResultSet is sensitive to scrolling, but insensitive to updates, i.e., not updateable.

25. Which of the following SQL keywords is used to read data from a database table?
    (a) SELECT (b) CHOOSE
    (c) READ (d) EXTRACT

26. Which of the following statement objects is used to call a stored procedure in JDBC?
    (a) Statement (b) PreparedStatement
    (c) CallableStatement (d) ProcedureStatement

27. Which of the following objects is used to obtain the DatabaseMetaData object?
    (a) Driver (b) DriverManager
    (c) Connection (d) ResultSet

28. Which of the following methods is used to call a stored procedure in the database?
    (a) execute() (b) executeProcedure()
    (c) call() (d) run()

29. What will be the effect if we call the deleteRow() method on a ResultSet object?
    (a) The row pointed to by the cursor is deleted from the ResultSet, but not from the database.
    (b) The row pointed to by the cursor is deleted from the ResultSet and from the database
    (c) The row pointed to by the cursor is deleted from the database, but not from the ResultSet.
    (d) None of the above

30. If you want to work with a ResultSet, which of these methods will not work on PreparedStatement?
    (a) execute() (b) executeQuery()
    (c) executeUpdate() (d) All of the above

31. Which of the following characters is used as a placeholder in CallableStatement?
    (a) $ (b) @ (c) ? (d) #

32. Which one of the following will not get the data from the first column of ResultSet rs, returned from executing the SQL statement: "SELECT login, password FROM USERS"?
    (a) rs.getString(0) (b) rs.getString("login")
    (c) rs.getString(1) (d) All of the above

33. Which of the following interfaces is used to control transactions?
    (a) Statement (b) Connection
    (c) ResultSet (d) DatabaseMetaData

34. Which one of the following represents the correct order?
    (a) INSERT, INTO, SELECT, FROM, WHERE
    (b) SELECT, FROM, WHERE, INSERT, INTO
    (c) INTO, INSERT, VALUES, FROM, WHERE
    (d) INSERT, INTO, WHERE, AND, VALUES

35. Which of the following is *not* a benefit of using JDBC?
    (a) JDBC programs are tightly integrated with the server operating system.
    (b) Systems built with JDBC are relatively easy to move to different platforms.
    (c) JDBC programs can be written to connect with a wide variety of databases.
    (d) JDBC programs are largely independent of the database to which they are connected.

36. In which of the following layers of the JDBC architecture does the JDBC–ODBC bridge reside?
    (a) database layer
    (b) client program layer
    (c) both client program and database layers
    (d) JDBC layer

37. Which database application model would an enterprise-wide solution most likely adopt?
    (a) The monolithic model
    (b) The two-tier model
    (c) The three-tier model
    (d) The n-tier model

38. Which code segment could execute the stored procedure "calculate()" located in a database server?
    (a) Statement stmt = connection.createStatement();
    stmt.execute("calculate()");
    (b) CallableStatement cs = con.prepareCall("{call calculate}");
    cs.executeQuery();
    (c) PrepareStatement pstmt = connection.prepare Statement("calculate()");
    pstmt.execute();
    (d) Statement stmt = connection.createStatement();
    stmt.executeStoredProcedure("calculate()");

22 INTRODUCTION TO J2EE

KEY OBJECTIVES

After completing this chapter readers will be able to

- have an idea about the technologies used in J2EE
- learn JavaBean technology and its advantages
- understand key JavaBean features such as Introspection, Customization, Persistence, etc
- get an idea about EJB component architecture
- learn Model–View–Controller architecture
- get an idea about struts framework

22.1 OVERVIEW OF J2EE

Today, in the fast-moving and highly demanding world of e-commerce and Information Technology (IT), there is a tremendous need for low-cost, distributed applications (especially transactional applications) for enterprises. These distributed enterprise applications must be designed, implemented, and deployed in less time, with few resources and greater speed.

Fortunately, the Java 2 Enterprise Edition (J2EE) provides a component-based technology for the design, development, assembly, and deployment of low-cost and fast-track enterprise applications. The purpose of J2EE is to simplify the design and implementation of distributed enterprise applications. J2EE offers the following functionalities:

- Multi-tiered distributed application model
- Reusable components
- Flexible transaction control
- Web services support through integrated data interchange on Extensible Markup Language (XML)-based open standards and protocols
- Unified security model

Since J2EE solutions are basically Java-based, they are platform-independent and are not tied to a specific vendor or customer. Customers and vendors are free to select from a wide variety of products and components.

J2EE is not a single technology; it consists of a large set of technologies, some of which are mentioned as follows:

- Enterprise Java Beans (EJB)
- Java Servlets
- Java Server Pages (JSP)
- Java Message Service (JMS)
- Java Naming and Directory Interface (JNDI)
- Java-XML
- J2EE Connector Architecture
- Java Mail
- Java DataBase Connectivity (JDBC)
- Remote Method Invocation (RMI)
- CORBA
- RMI-IIOP

We have already covered some of these technologies in previous chapters. In this chapter, we shall discuss some other J2EE technologies, especially, JavaBean and EJB. In Chapter 21, an introduction to JavaBean technology was given. Since JavaBean is the fundamental component technology and is a basic building block of EJB, we must discuss this technology in detail.

22.2 INTRODUCTION TO JAVABEANS

JavaBean technology is a Java-based technology to design and develop reusable and platform-independent software components. These components can then be integrated virtually in all Java-compatible applications. JavaBean components are knows as *beans*. The appearance and features of a bean can be changed or customized using builder tools such as BeanBox and NetBeans.

A bean class is nothing but a usual Java class with the following requirements:

- It must have a public no-argument constructor.
- It has public accessor methods to read and write properties.
- It should implement `Serializable` or `Externalizable` interface.

So, virtually all Java classes are already bean classes or they can be converted to beans with a little effort. For example, following is an example of a bean class.

```
//Factorial.java
import java.io.*;
public class Factorial implements Serializable {
   protected int n;
   public int getN() {
      return n;
```

```
    }
    public void setN(int n) {
        this.n = n;
        long prod = 1;
        for(int i = 2; i <= n; i++)
            prod *= i;
        fact = prod;
    }
    protected long fact;
    public long getFact() {
        return fact;
    }
}
```

Here is another example of a bean class:

```
//State.java
import java.io.Serializable;
public class State implements Serializable {
    protected String state = "off";
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}
```

Most often, JavaBeans have a GUI representation of themselves. For example, button, calculator, and calendar beans are expected to have a visual representation, so as to be useful. However, some beans do not have any visual representation, and are called *invisible* beans. Invisible beans include a spell checker, random number generator, and temperature monitor.

22.2.1 Properties

A bean typically has one or more *properties* that control appearance and behavior. Properties are discrete, named attributes of a bean. They constitute the data part of a bean's structure and represent the internal state of a bean. Properties allow us to isolate component state information into discrete pieces that can be easily modified.

In our `Factorial` bean, there are two properties: `n` and `fact`. The property `n` holds an integer value, whose factorial is stored in the property `fact`. Here, `n` is said to be independent property and `fact` is dependent on `n` as the value of `fact` can be calculated from `n`. There need not be such a relationship in all the cases. For example, in the `State` bean, there is only one property `state`. In general, a bean can have any number of properties.

22.2.2 Accessor Methods

The primary way to access bean properties is through accessor methods. An accessor method is a public method of the bean that directly reads or writes the value of a particular bean. For example, the `getN()` method of the `Factorial` bean returns the value of the property `n`. Similarly,

`setN()` sets the value of `n`. For the `fact` property, only one method is provided, which returns the value of the property `fact`. In general, a property can have two methods associated with it. The methods that are responsible for reading property values are called *reader* methods. Similarly, methods that are responsible for changing property values are called *writer* methods.

22.3 BEAN BUILDER

Readymade applications are available that provide an environment in which we can build and test beans. Those tools are known as builder tools. Some of the popular bean builder tools are Sun's BeanBox and NetBeans, JBuilder, Borland's Delphi, etc.

A builder tool is able to display several beans simultaneously. It is also able to connect different beans together in a visual manner. So, we can manipulate a bean without writing any piece of code. Most builder tools have a palette of components from which developers can drag and drop components onto a graphical canvas.

The facilities supported by a bean builder vary widely, but there are some common features as follows:

- *Introspection*—ability to analyze a bean to discover properties, methods, and events.
- *Properties*—used for customization of the state of a bean
- *Events*—a way used by beans to tell the outside that something is happening, or to react to something that happened outside the bean
- *Customization*—a developer of a bean can customize the behavior and appearance of the bean within a builder tool.
- *Persistence*—a builder tool allows us to customize a bean, and have its state saved away and reloaded later

22.4 ADVANTAGES OF JAVABEANS

Before we discuss the features and facilities supported by JavaBean technology, let us understand what JavaSoft wanted to accomplish by developing this Java-based component technology. In this regard, we can refer to JavaSoft's own JavaBeans mission statement: "Write once, run anywhere, use everywhere". Let us examine each part of this statement to realize what JavaSoft had in mind.

Write Once

A good software component technology should encourage component developers to write code only once. The component should not require rewriting of code to add and/or improve features. Keeping this point in mind, JavaSoft developed JavaBean technology, which allows us to add or improve functionality in the existing code, without re-engineering the original code.

The concept of writing components once also adds sense in terms of version control. This means that developers can make changes to components incrementally instead of rewriting significant portions from scratch. This results in a steady improvement of functionality, which, in turn, dictates a more consistent software component development through increasing versions.

Run Anywhere

A software technology must be platform independent to have a realistic meaning in today's rapidly varying software environment. This refers to the capability of software components developed using a technology to be executed (run) in any environment. Fortunately, JavaBeans components are ultimately Java components. Consequently, cross-platform support comes automatically.

Good software components should also run across distributed network environments. This can be simply achieved using Java's powerful technology, Remote Method Invocation (RMI), Socket, and RMI-IIOP.

Use Everywhere

This is perhaps the most important part of the JavaSoft's JavaBean mission statement. Well-designed JavaBeans components are capable of being used in a wide range of situations, which include applications, other components, documents, websites, and application builder tools, and so on.

In addition to these key features, JavaBean technology has the following built-in features.

JavaBeans are compact components and can be transferred across a low-bandwidth Internet connection that facilitates a reasonable transfer time.

JavaBeans components are so simple that they are not only easy to use, but also easy to develop. Therefore, we can devote more time to embellishing components with interesting features than debugging them.

22.5 BDK INTROSPECTION

A bean builder usually uses Java core reflection API to discover methods of a bean. It then applies the design pattern to discover other bean features (such a properties and events). This procedure is known as *introspection.*

22.5.1 Design Patterns

The JavaBeans framework introduces a lot of rules for names to be used for classes and methods. They are collectively called *design patterns*.

Note that the conventions and design patterns are all optional. However, by following those conventions, we can create really useful beans that can be used within a builder tool as well as in other JavaBean-enabled environments. Builder tools such as BeanBox, NetBeans, and JBuilder use these conventions and design patterns to introspect a bean's properties, methods, and events. Consequently, they can provide an environment where we can customize bean features.

Suppose, a bean has a property called "xxx". JavaBean technology encourages us to use "getXxx" as the name of the reader method and "setXxx" as the name of the writer method. If the property happens to be a Boolean, the reader method could be named "isXxx". For example, our State bean has the property `state`. So, the reader method is "getState" and the writer method is "setState".

A `BeanInfo` class is a class that is used to provide information about a bean explicitly. To do this, a class is created implementing the `BeanInfo` interface. The name of this class should also follow a naming rule. The name of the `BeanInfo` class must be the name of the target bean,

followed by the string "BeanInfo". For example, the name of the `BeanInfo` class for the bean `Person` must be "PersonBeanInfo".

Similarly, the name of the `Customizer` class (see Section 22.9) for the bean `MyBean` must be `MyBeanCustomizer`.

JavaBean API provides interfaces and classes that can be used to discover properties and methods of a bean dynamically. Following is brief description of those interfaces and classes:

`BeanInfo`

The `java.beans.BeanInfo` interface defines a set of methods that allow bean developers to provide information about their beans explicitly. By specifying `BeanInfo` for a bean component, a developer can hide methods, specify an icon for the toolbox, provide descriptive names for properties, define which properties are bound, etc.

`Introspector`

The `java.beans.Introspector` class provides descriptor classes (see Section 22.7) with information about properties, methods, and events of a bean.

The `getBeanInfo()` method of the Introspector class can be used by builder tools and other automated environments to provide detailed information about a bean. The `getBeanInfo()` method relies on the naming conventions for the bean's properties, events, and methods. A call to `getBeanInfo()` results in the introspection process analyzing the bean's classes and super classes. The following example finds the properties and methods of our `Factorial` bean.

```
//IntrospectionDemo.java
import  java.beans.*;
public  class  IntrospectionDemo  {
   public static void main( String[] args ) throws IntrospectionException  {
         BeanInfo  info  =  Introspector.getBeanInfo(  Factorial.class,
Object.class );
      System.out.println("properties:  ");
      for  (  PropertyDescriptor  pd  :  info.getPropertyDescriptors()  )
         System.out.println("  "  +  pd.getName());
      System.out.println("methods:  ");
      for  (  MethodDescriptor  pd  :  info.getMethodDescriptors()  )
         System.out.println("  "  +  pd.getName());
   }
}
```

It generates the output shown in Figure 22.1:

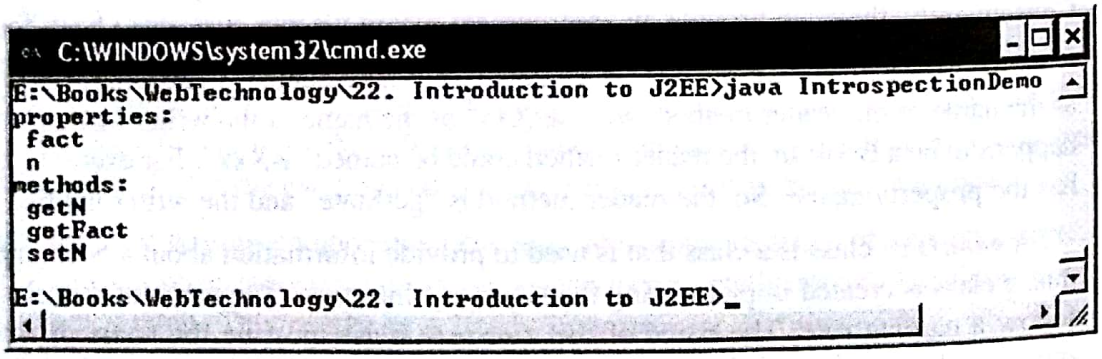


Figure 22.1 Bean introspection

22.6 PROPERTIES

The properties of a bean represent and control its behavior and appearance. JavaBean API allows us to create the following primary property types.

22.6.1 Simple Properties

Simple properties of a bean are those that do not depend on other beans or control properties of another bean. Properties are typically declared as `private`. To access them, `get` and `set` methods are used. The names of the `get` and `set` methods follow specific rules, known as *design patterns*. JavaBean-enabled builder/tester tools (such as NetBeans and BeanBox) use these design patterns to do the following:

- Discover the properties of a bean
- Determine the types of the properties
- Display the properties in the property window
- Determine the read/write attribute of the properties
- Find the appropriate property editor for each property type
- Allow us to change the properties

Suppose a bean builder/tester tool encounters the following methods on a bean:

```
public int getValue() { ... }
public void setValue(int v) { ... }
```

The tool infers the following:

- There exists a property name `value`.
- Its type is `int`.
- It is readable and writable.
- It displays the value of this property in the property editor.
- Moreover, it finds the property editor that allows us to change the value of the property.

Creating a simple property

The following code is used in the bean to create the property `propertyName`:

```
private PropertyType propertyName = initialValue;
```

Providing a reader method

The following code is used to provide a reader method:

```
PropertyType getPropertyName() {
  return propertyName;
}
```

Providing a writer method

The following code is used to provide a writer method:

```
void getPropertyName(PropertyType value) {
```

```
        propertyName = value;
    }
```

The following example shows how to create a simple property:

```
//MyBean.java
public class MyBean {
   private int value = 0;

    /**
     * Get the value of value
     *
     * @return the value of value
     */
    public int getValue() {
       return value;
    }

    /**
     * Set the value of value
     *
     * @param value new value of value
     */
    public void setValue(int value) {
       this.value = value;
    }
}
```

22.6.2 Bound Properties

Sometimes, when a property of a bean changes, you might want to notify another object about this change. This object typically reacts to the change by changing one of its properties. For example, consider two beans `Parent` and `Child`, each having two properties, `firstName` and `familyName`. The `familyName` property of `Child` and `Parent` must have the same value. So, whenever the `familyName` property of `Parent` changes, the `familyName` property of `Child` must also be changed to make them synchronized.

This synchronization can be implemented using the *bound* property. The accessor or modifier methods for a bound property are defined in the same way, except that whenever a bound property changes, a notification is sent to the interested listeners. Whenever a property of a bean changes, a "PropertyChange" event gets fired. The bean generating the event is called *source* bean. We can register one or more "Listener" objects with a source, so that these objects get notified when a bound property of the source bean is updated.

Bean Development Kit (BDK) provides special classes to accomplish coordination between the notifier and listener.

`PropertyChangeListener`

If an object is interested in being notified about the property changes of a source bean, its class must implement the `PropertyChangeListener` interface. This interface defines the following method:

```
void propertyChange(PropertyChangeEvent pce)
```

The listener's class must implement this method. If the listener is registered for a property change event, this method gets called when a bound property of the source bean changes. In this method, the listener object reacts to the property change by modifying one or more of its properties.

PropertyChangeEvent

A `PropertyChangeEvent` object is generated whenever a bound property of a bean changes. This object is then sent from the source bean to all registered `PropertyChangeListener` objects, as an argument of their respective `PropertyChange()` method. This class encapsulates the property change information. It provides several useful methods as follows:

```
String getPropertyName()
```

Returns the name of the property that was changed and caused this event firing

```
Object getOldValue()
```

Returns the value of the property as an object before it was changed. The bean writer should typecast it to the desired type.

```
Object getNewValue()
```

Returns the value of the property as an object after it was changed. The bean writer must typecast it to the desired type.

PropertyChangeSupport

This class is used by the beans having at least one bound property to keep track of registered listeners. It is also used to deliver `PropertyChangeEvent` objects to those registered listeners when a bound property changes. A source bean can instantiate this class as a member field or inherit the functionality of this class by extending it. Figure 22.2 gives the framework for bound property support.

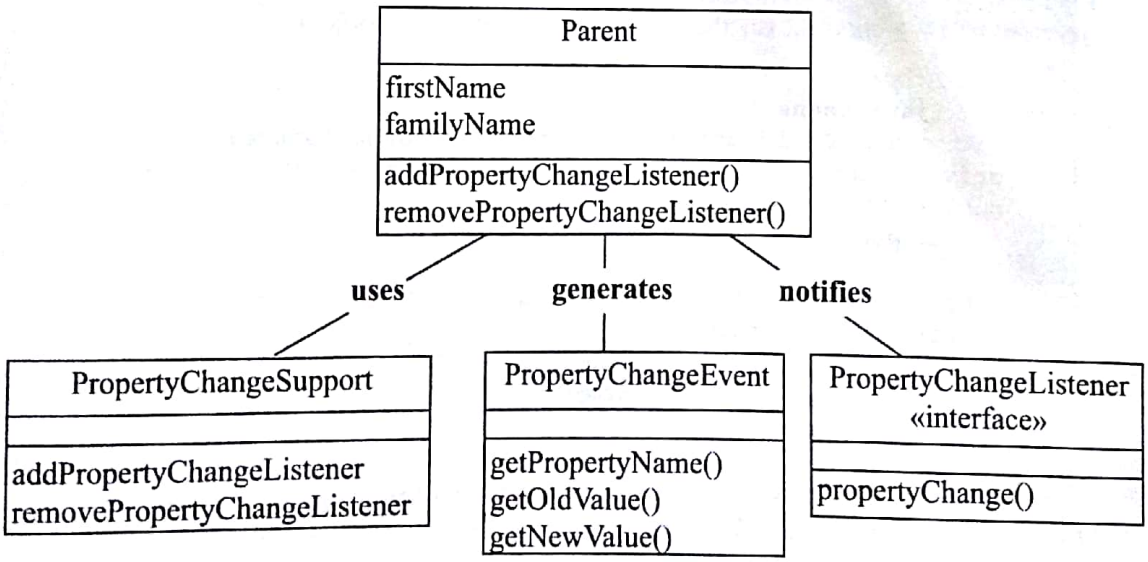


Figure 22.2 Bound property support framework

Let us now discuss how to write a bound property. The following class acts as a source bean.

```
//Parent.java
import java.beans.*;
public class Parent {
   private String firstName = "Uttam", familyName = "Roy";
   private PropertyChangeSupport pcs;
   public Parent() {
      pcs = new PropertyChangeSupport(this);
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String fname) {
      firstName = fname;
   }
   public String getFamilyName() {
      return familyName;
   }
   public void setFamilyName(String newFamilyName) {
      String oldFamilyName = familyName;
      familyName = newFamilyName;
      pcs.firePropertyChange("familyName", oldFamilyName, newFamilyName);
   }
   public void addPropertyChangeListener(PropertyChangeListener pcl) {
      pcs.addPropertyChangeListener(pcl);
   }
   public void removePropertyChangeListener(PropertyChangeListener pcl) {
      pcs.removePropertyChangeListener(pcl);
   }
}
```

The following bean is a listener that wants to be notified when the `familyName` property of the `Parent` bean changes. So, it implements the `PropertyChangeListener` interface and defines the method `propertyChange()`. In the `propertyChange()` method, it changes its own `familyName` property to the new value of the `familyName` property of the `Parent` bean by using the `getNewValue()` method on the `PropertyChangeEvent` object.

```
//Child.java
import java.beans.*;
public class Child implements PropertyChangeListener {
   private String firstName = "Rimisha", familyName = "Roy";
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String fname) {
      firstName = fname;
   }
   public String getFamilyName() {
      return familyName;
   }
   public void setFamilyName(String fname) {
      familyName = fname;
   }
```

```
    public void propertyChange(PropertyChangeEvent pce) {
    if(pce.getPropertyName().equals("familyName"))
          setFamilyName((String)pce.getNewValue());

    }
}
```

For demonstration purposes, we have created the following Java application program. Typically, a bean builder (such as BeanBox provided by Sun) may be used.

```
//BoundDemo.java
public class BoundDemo {
   public static void main(String args[]) {
      Parent p = new Parent();
      Child c = new Child();
      p.addPropertyChangeListener(c);
      System.out.println("Before changing family name");
      System.out.println("Parent: " + p.getFirstName() + " " + p.getFamilyName());
      System.out.println("Child: " + c.getFirstName() + " " + c.getFamilyName());

      p.setFamilyName("Biswas");
      System.out.println("After changing family name of parent to 'Biswas'");
      System.out.println("Parent: " + p.getFirstName() + " " + p.getFamilyName());
      System.out.println("Child: " + c.getFirstName() + " " + c.getFamilyName());
   }
}
```

This program creates a `Parent` bean and a `Child` bean. The `Child` bean is then registered with the `Parent` bean. So, whenever the `familyName` property of the `Parent` bean changes, the `Child` bean gets notified and it can change its own `familyName` property. Finally, we have changed the `familyName` property of the `Parent` bean using the `setFamilyName()` method. We also have displayed the details of parent and child before and after changing the property. The output is as shown in Figure 22.3.

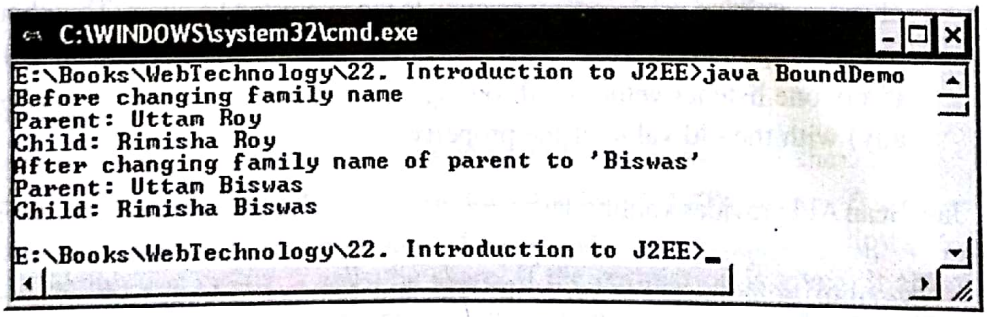


Figure 22.3 Bean bound property demo

22.6.3 Constrained Properties

A *constrained* property of a bean is a special type of property, which can be changed subject to prior permission taken from external object(s). External objects act as *vetoers* and exercise such authority in this case. For example, think about `Broker` and `ShareHolder` beans. The `Broker` can sell the shares at a rate provided that `ShareHolder` allows it to do so.

JavaBean API provides a mechanism very similar to the one used for the bound property, which allows other objects to veto the change of the source bean's property. The basic idea for implementing a constrained property is as follows:

- Store the old value of source bean's property if it is to be vetoed.
- Ask listeners (vetoers) of the new proposed value.
- Vetoers are authoritative to process this new value. They disallow by throwing an exception. Note that no exception is thrown if a vetoer gives the permission.
- If no listener vetoes the change, i.e., no exception is thrown, the property is set to the new value; optionally notify "PropertyChange" listeners, if any.

Three objects are involved in this process.

- The *source* bean containing one or more constrained property
- A listener object that accepts or rejects changes proposed by the source bean
- An `PropertyChangeEvent` object encapsulating the property change information

22.6.3.1 Implementing constrained property support

In the source bean, the accessor method for the constrained property is defined in the same way, except that it throws a `PropertyVetoException` as follows:

```
public void setPropertyName(PropertyType pt) throws PropertyVetoException {
   //method body
}
```

A source bean containing one or more constrained properties implements the following functionalities:

- Provides methods to add and remove `VetoableChangeListener` objects to register and unregister them so that they receive notification for property change proposal.
- When a property change is proposed, the source bean sends the `PropertyChangeEvent` object containing proposed information to the interested listeners. This should be done before the actual property change takes place. It allows listeners to accept or refuse a proposal.
- If any one listener vetoes by throwing an exception, continue to notify other listeners (if any) with the old value of the property.

JavaBean API provides a utility class, `VetoableChangeSupport`, similar to `PropertyChangeSupport`. The `VetoableChangeSupport` class provides the methods `addVetoableChangeListener()` and `removeVetoableChangeListener()` to add and remove `VetoableChangeListener` objects, respectively, and keeps track of such listeners. It also provides a method, `fireVetoableChange()`, to send the `PropertyChangeEvent` object to each registered listener when a property change is proposed.

The source bean can instantiate this class as a member field or inherit the functionality of this class by extending it. A `VetoableChangeSupport` object is instantiated as follows:

```
VetoableChangeSupport vcs = new VetoableChangeSupport(this);
```

22.6.3.2 Implementing constrained property listener

If an object's class wants to act as a vetoer, it must implement the `VetoableChangeListener` interface. This interface defines a single method as follows:

```
void vetoableChange(PropertyChangeEvent pce) throws PropertyVetoException;
```

A `VetoableChangeListener` must implement this method. This is the method where the vetoer exercises its power and agrees or disagrees with the proposal. In this method, the listener processes proposed property change and disagrees (vetoes) by throwing a `PropertyVetoException` exception. A typical code looks like this:

```
void vetoableChange(PropertyChangeEvent pce) throws PropertyVetoException {
   if(the_condition_is_not_fulfilled)
      throw new PropertyVetoException("NO", pce);
}
```

Figure 22.4 shows the constrained property support framework.

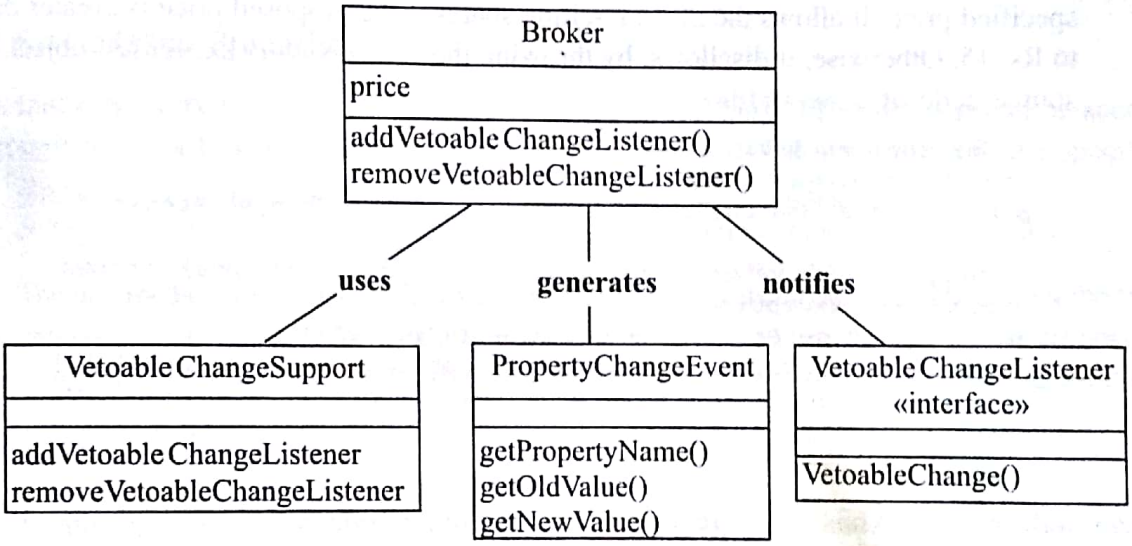


Figure 22.4 Constrained property support framework

22.6.3.3 Example

Let us now demonstrate the constrained property with our `Broker` and `ShareHolder` beans. The `Broker` is the source bean that has a single constrained property, `price`. A `Broker` can sell a share owned by a `ShareHolder`, provided that the `ShareHolder` object allows it. So, `Broker` should take permission before it sells the share. If the permission is given, it actually sells the shares. Otherwise, it does not do anything. Following is the source code of the `Broker` bean.

```
//Broker.java
import java.beans.*;
public class Broker {
   private float price = 10;
   private VetoableChangeSupport vcs;
   public Broker() {
      vcs = new VetoableChangeSupport(this);
   }
```

```
    public float getPrice() {
        return price;
    }
    public void setPrice(float newPrice) throws PropertyVetoException {
        float oldPrice = price;
        vcs.fireVetoableChange("price", oldPrice, newPrice);
        System.out.println("Setting new price limit");
        price = newPrice;
    }
    public void addVetoableChangeListener(VetoableChangeListener pcl) {
        vcs.addVetoableChangeListener(pcl);
    }
    public void removeVetoableChangeListener(VetoableChangeListener pcl) {
        vcs.removeVetoableChangeListener(pcl);
    }
}
```

The `ShareHolder` is the listener bean. It decides whether the `Broker` can sell shares at the specified price. It allows the `Broker` selling shares if the proposed price is greater than or equal to Rs. 15. Otherwise, it disallows, by throwing the `PropertyVetoException` object. Here is the source code of `ShareHolder`.

```
//ShareHolder.java
import java.beans.*;
public class ShareHolder implements VetoableChangeListener {

  public void vetoableChange(PropertyChangeEvent pce) throws
PropertyVetoException {
    if(pce.getPropertyName().equals("price")) {
      float price = ((Float)pce.getNewValue()).floatValue();
      if(!price >= 15 ) throw new PropertyVetoException("NO", pce);
    }

  }
}
```

For demonstration purposes, we have created a Java application that creates a `Broker` and a `ShareHolder` bean. It then registers the `ShareHolder` with the `Broker` so that it receives notification for the property change proposal. Finally, it tries to change the price limit of the share passed as an argument.

```
public class ConstrainedDemo {
  public static void main(String args[]) {
    Broker b = new Broker();
    ShareHolder s = new ShareHolder();
    b.addVetoableChangeListener(s);
    float price = Float.parseFloat(args[0]);
    try {
      System.out.println("Old price limit: " + b.getPrice());

      b.setPrice(price);
      System.out.println("New price limit: " + b.getPrice());
    }catch(Exception e) {
```

```
            System.out.println("Ah! failed to set price at " + price);
        }
    }
}
```

Figure 22.5 shows a sample output.

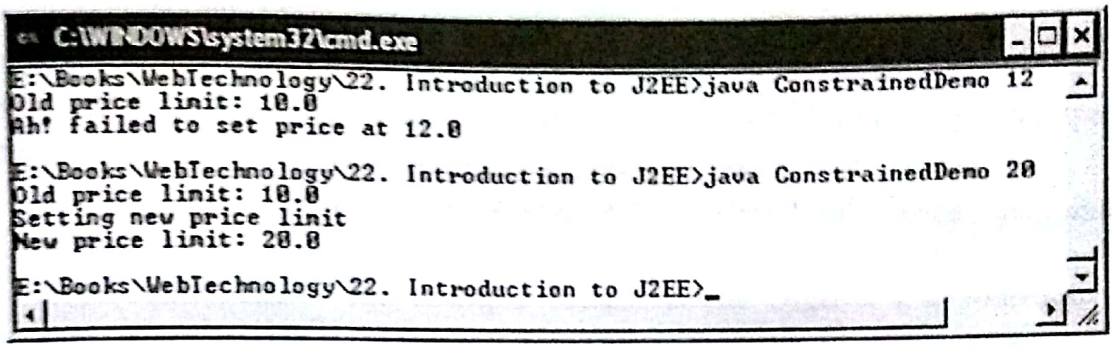


Figure 22.5 Bean constrained property demo

22.6.4 Indexed Properties

An index property is an array of properties. It can hold a range of values that can be accessed through accessor functions. To access an individual element, following methods are specified:

```
public PropertyElementType getPropertyName(int index)
public void setPropertyName(int index, PropertyElementType element)
```

The `get` method takes an array index and returns the element at that index. The `set` method takes two arguments: an index of the property array and its value, and sets the element at the specified index to the specified value. Methods to access the entire array are also specified as follows:

```
public PropertyElementType[] getPropertyName()
public void setPropertyName(PropertyElementType element[])
```

In this case, the `get` method returns the entire property array and the `set` method takes an entire array to be used to set the property array.

The following bean class stores the temperatures of the last seven days.

```
//TemperatureBean.java
public class TemperatureBean {
   private float[] temperatures = new float[7];
   /**
    * Get the value of temperatures
    *
    * @return the value of temperatures
    */
   public float[] getTemperatures() {
      return temperatures;
   }

   /**
    * Set the value of temperatures
    *
```

```
     * @param temperatures new value of temperatures
     */
    public void setTemperatures(float[] temperatures) {
        this.temperatures = temperatures;
    }

    /**
     * Get the value of temperatures at specified index
     *
     * @param index
     * @return the value of temperatures at specified index
     */
    public float getTemperatures(int index) {
        return this.temperatures[index];
    }

    /**
     * Set the value of temperatures at specified index.
     *
     * @param index
     * @param newTemperatures new value of temperatures at specified index
     */
    public void setTemperatures(int index, float newTemperatures) {
        this.temperatures[index] = newTemperatures;
    }
}
```

22.7 BEANINFO INTERFACE

A bean builder typically uses the *introspection* process to discover features (such as properties, methods, and events). In this case, the bean builder exposes all the features to the outside world.

We can also expose the bean's features by using an associated class explicitly. By doing this we obtain the following benefits:

- Hide features we do not want to disclose
- Provide more information about bean features
- Associate an icon with the target bean
- Group bean features into different categories such as normal and advanced groups
- Specify a customizer class
- Expose some bean features explicitly and others using Java reflection API

The features of a bean are exposed using a separate special class. This class must implement the `BeanInfo` interface. The `BeanInfo` interface defines several methods that can be used to inspect properties, methods, and events of a bean. Following is the declaration of the `BeanInfo` interface:

```
public interface java.beans.BeanInfo{
    public static final int ICON_COLOR_16x16;
    public static final int ICON_COLOR_32x32;
    public static final int ICON_MONO_16x16;
    public static final int ICON_MONO_32x32;
```

```
    public abstract java.beans.BeanDescriptor getBeanDescriptor();
    public abstract java.beans.EventSetDescriptor[] getEventSetDescriptors();
    public abstract int getDefaultEventIndex();
    public abstract java.beans.PropertyDescriptor[] getPropertyDescriptors();
    public abstract int getDefaultPropertyIndex();
    public abstract java.beans.MethodDescriptor[] getMethodDescriptors();
    public abstract java.beans.BeanInfo[] getAdditionalBeanInfo();
    public abstract java.awt.Image getIcon(int);
}
```

The BeanInfo interface uses several *descriptor* classes, each of which describes specific features. Following is a list of descriptors:

- `BeanDescriptor`—This descriptor class describes the bean's name, type, and its customizer class, if any.
- `PropertyDescriptor`—This descriptor encapsulates the target bean's properties.
- `MethodDescriptor`—This descriptor encapsulates the target bean's methods.
- `EventSetDescriptor`—This descriptor encapsulates the events that the target bean can fire.

Let us now write a `BeanInfo` class that describes the features of a bean. For this purpose, we shall first write a bean and show how the BeanBox extracts information of this bean. Consider the following bean class:

```
//Person.java
public class Person {
   private String name = "B. S. Roy";
   private String address = "Narayanpur, Kol-136";
   private String PAN = "AGCPR8830P";
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getAddress() {
      return address;
   }
   public String getPAN() {
      return PAN;
   }
   public void setPAN(String PAN) {
      this.PAN = PAN;
   }
   public void setAddress(String address) {
      this.address = address;
   }
}
```

The bean `Person` has three properties: `name`, `address`, and `PAN`. Each of the three properties has a `get` and a `set` method. Compile this bean using the following command:

```
java Person.java
```

Create a class file, `Person.class`. Create a manifest file, `manifest_person.mf`, as follows:

```
Manifest-Version: 1.0

Name: Person.class
Java-Bean: True
```

Place this file in the same directory as `Person.class`. Now, create a jar file, `person.jar`, using the following command:

```
jar cvfm person.jar manifest_person.mf Person.class
```

It typically generates the following output:

```
added manifest
adding: Person.class(in = 754) (out= 410)(deflated 45%)
```

The bean is now ready to test and use. Load the `person.jar` file in the Bean Box. Create an instance of the `Person` bean and view its properties. It looks as shown in Figure 22.6.

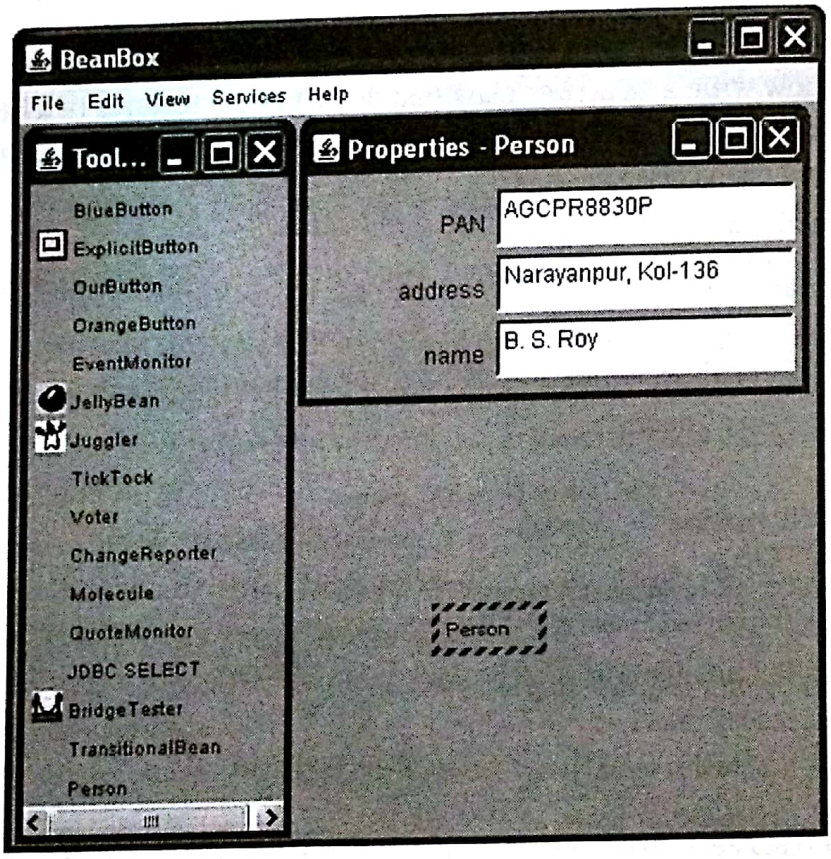


Figure 22.6 Bean introspection

As you can see, BeanBox shows all the properties of the `Person` bean. Suppose we do not disclose the `PAN` property of the `Person` bean, as it is sensitive information. For that purpose, we can write the `BeanInfo` class. Following is the source code of the `BeanInfo` class.

```
//PersonBeanInfo.java
import java.beans.*;
public class PersonBeanInfo extends SimpleBeanInfo {
```

```
    private Class personClass = Person.class;
    public PropertyDescriptor[] getPropertyDescriptors() {
        PropertyDescriptor name = null, address = null, PAN = null;
        try {
                name = new PropertyDescriptor("name", personClass);
            name.setDisplayName("Name:");
            name.setPreferred(true);

            address = new PropertyDescriptor("address", personClass);
            address.setDisplayName("Address:");
            address.setPreferred(true);
        } catch (IntrospectionException e) {}
        PropertyDescriptor[] result = {name, address};
        return result;
    }
}
```

To write a `BeanInfo` class, we need to give a name. The name of each `BeanInfo` class follows a naming rule. The name of the `BeanInfo` class of the target bean X must have the name `XBeanInfo`. For example, the name of the `BeanInfo` class of our `Person` bean must be `PersonBeanInfo`. The string "BeanInfo" is appended to the target bean class name.

Since the `BeanInfo` interface implements several methods, we have to implement all of them. Alternatively, a `BeanInfo` class can be created by sub-classing the `SimpleBeanInfo` class. The `SimpleBeanInfo` class is a convenient base class for the `BeanInfo` classes. It implements all the methods, but all are empty. So, the `SimpleBeanInfo` class does not disclose any property of the method at all. We can override a specific method, where we want to return specific information. Since we want to hide the PAN property, we have overridden only the `getPropertyDescriptors()` method. In this method, we have returned an array of only three properties.

Now, modify the manifest file, `manifest_person.mf`, as follows:

```
Manifest-Version: 1.0

Name: Person.class
Java-Bean: True

Name: PersonBeanInfo.class
Java-Bean: False
```

Create the `person.jar` file as shown in Figure 22.7.

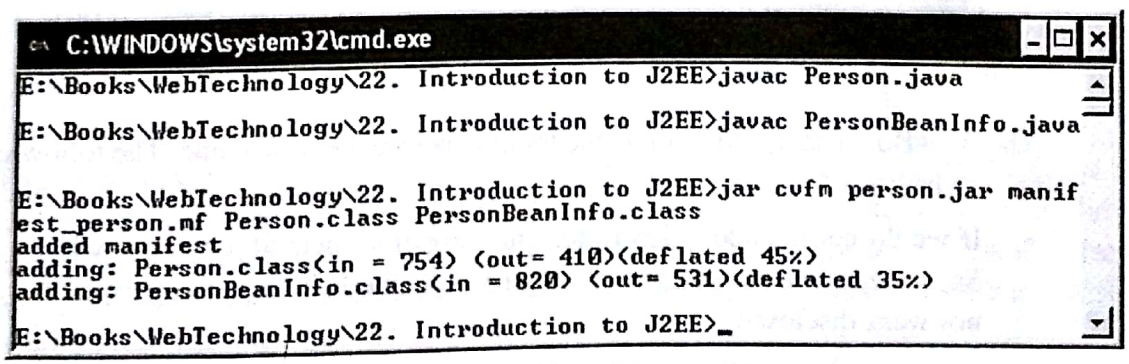


Figure 22.7 Bean introspection

Load the jar file in BeanBox and create a `Person` bean. You can see that only three properties are shown. The result is shown in Figure 22.8.

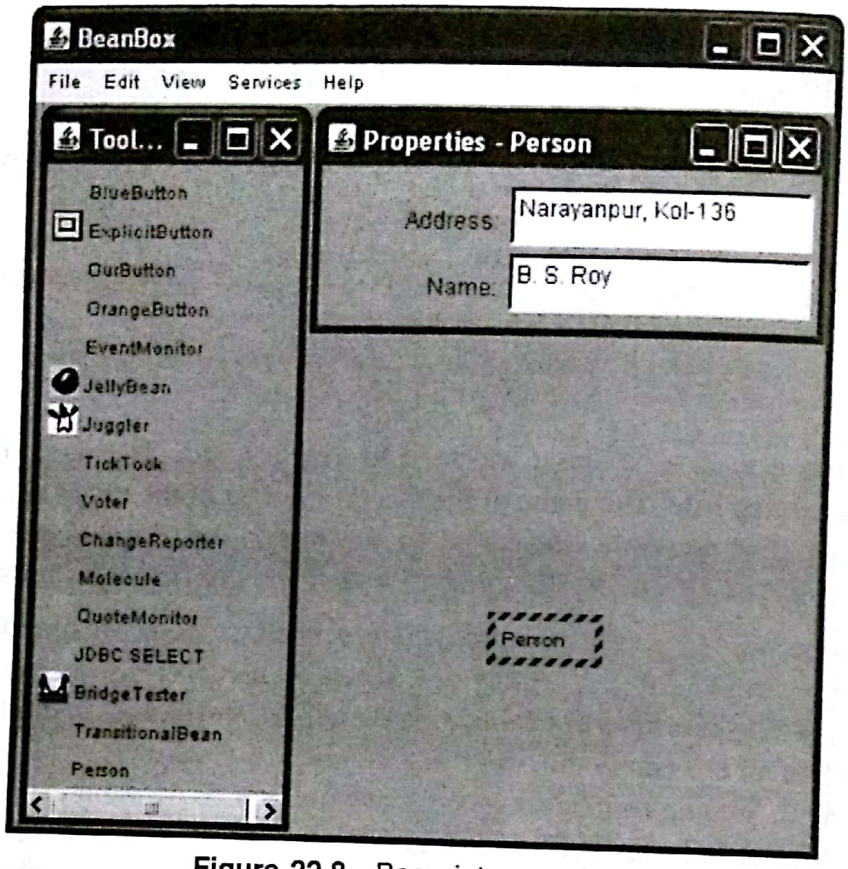


Figure 22.8 Bean introspection

We can also associate an icon with the bean. Override the `getIcon()` method as follows:

```
public java.awt.Image getIcon(int iconKind) {
   if (iconKind == BeanInfo.ICON_MONO_16x16 ||
         iconKind == BeanInfo.ICON_COLOR_16x16 ) {
      java.awt.Image img = loadImage("person16.gif");
      return img;
   }
   if (iconKind == BeanInfo.ICON_MONO_32x32 ||
         iconKind == BeanInfo.ICON_COLOR_32x32 ) {
      java.awt.Image img = loadImage("person32.gif");
      return img;
   }
   return null;
}
```

The BeanBox displays the icon in the toolbox before the bean name. The following important points can be noted:

- If we do not include a descriptor, that property, method, or event will *not* be exposed. So, we can selectively expose properties, methods, or events by excluding those we do not want disclosed.

- If a feature's `get` (for example, `getMethodDescriptor()`) method returns null, low-level Java reflection API is then used to extract that feature. This means that developer-defined as well as low-level reflection both can be used to discover the methods. If default methods of the `SimpleBeanInfo` class, which return null, are not overridden, low-level reflection is used for that feature.

22.8 PERSISTENCE

Sometimes, it is necessary to store the beans for later use. *Persistence* is a procedure to save a bean in non-volatile storage such as a file. The bean can be reconstructed and used later. The important point is that persistence allows bean developers to save the current state of the bean and retrieve the same at some later point of time.

To understand the importance of persistence, let us understand the life cycle of a bean. A bean is first created (possibly using a builder tool). Then its properties are accessed and/or manipulated by calling its public methods. After a degree of use, the bean is no longer needed, and is destroyed and removed from the memory. This is a typical life cycle of a bean. However, think about a situation where you have decided to complete your application temporarily but use of the bean is not yet over. You want the same bean with the state of the bean unchanged whenever the application is restarted later. Persistence is a procedure that does exactly what we have described here.

Let us consider a simple example. Consider the following `Factorial` bean.

```
//Factorial.java
import java.io.*;
public class Factorial implements Serializable {
   protected int n;
   public int getN() {
      return n;
   }
   public void setN(int n) {
      this.n = n;
      long prod = 1;
      for(int i = 2; i <= n; i++)
         prod *= i;
      fact = prod;
   }
   protected long fact;
   public long getFact() {
      return fact;
   }
}
```

This bean has two properties, `n` and `fact`. The property `n` has a `get` method and a `set` method and `fact` has only a `get` method. The `getFact()` method returns the factorial of `n`, which can be specified using the `setN()` method. So, this bean can be used to create a factorial table. Suppose we want to create a table containing factorial of numbers from 2 to 10. So, we can write a code like this:

```
Factorial f = new Factorial();
for(int i = 2; i <= 10; i++) {
   f.setN(i);
   System.out.println(i + "! = " + f.getFact()+ " ");
}
```

However, during the calculation, assume that an external interrupt may occur and in such a case, the application must be terminated. Before terminating, we must save the state of the bean so that we can retrieve the value of the integer up to which we calculated the factorial, so as to find the factorial of the rest of the integers. Java's Serialization procedure may be used to save and retrieve the state of a bean. An object is said to be serializable if its class implements the `Serializable` interface and all members are serializable. The `Serializable` interface does not define any method; it merely indicates that the class implementing it can be serialized. The following code shows how to save the state of a bean.

```
//SaveBean.java
import java.io.*;
public class SaveBean {
   public static void main(String args[]) {
      try {
         Factorial f = new Factorial();
         for(int i = 2; i < 10; i++) {
            f.setN(i);
            System.out.println(i + "! = " + f.getFact()+ " ");
            if(i > Math.random()*10 ) break;
         }
         //save the state of f now
         FileOutputStream fos = new FileOutputStream("out.dat");
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(f);
         oos.close();
         System.out.println("Factorial bean saved in file out.dat");
      }catch(Exception e) {e.printStackTrace();}
   }
}
```

The following code demonstrates how to retrieve the bean with its original state.

```
//RetrieveBean.java
import java.io.*;
public class RetriveBean {
   public static void main(String args[]) {
      try {
         FileInputStream fis = new FileInputStream("out.dat");
         ObjectInputStream ois = new ObjectInputStream(fis);
         Factorial f1 = (Factorial)ois.readObject();
         ois.close();
         System.out.println("Factorial bean retrieved from file out.dat");
         for(int i = f1.getN()+1; i <= 10; i++) {
            f1.setN(i);
            System.out.println(i + "! = " + f1.getFact()+ " ");
         }
      }catch(Exception e) {e.printStackTrace();}
   }
}
```

Figure 22.9 shows a sample output that is obtained if you run these two applications.

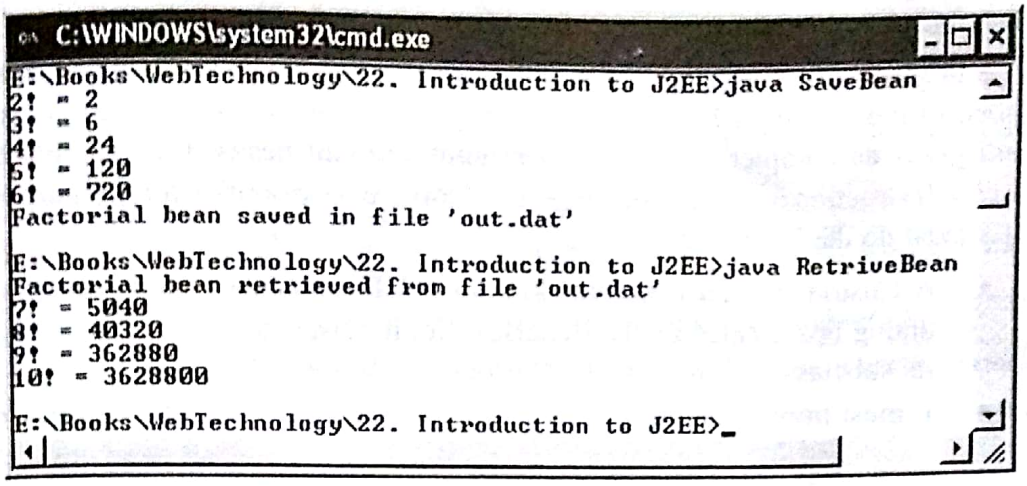


Figure 22.9 Saving and retrieving bean

The JavaBean framework provides a utility class, `XMLEncoder`, which allows us to save a bean in XML format. The following example shows how to save a bean as an XML file:

```
XMLEncoder encoder = new XMLEncoder(
   new BufferedOutputStream(new FileOutputStream("fact.xml")));
encoder.writeObject( f );
encoder.close();
```

It creates the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_10-beta" class="java.beans.XMLDecoder">
 <object class="Factorial">
  <void property="n">
   <int>6</int>
  </void>
 </object>
</java>
```

The bean can later be retrieved from this XML document using the following code:

```
XMLDecoder decoder = new XMLDecoder(
   new BufferedInputStream(new FileInputStream("fact.xml")));
f1 = (Factorial)decoder.readObject();
decoder.close();
```

The object `f1` refers to the original bean now.

22.9 CUSTOMIZER

Bean *customization* allows us to customize the appearance and behavior of a bean at design time, within a bean-compliant builder tool. There are two ways to customize a bean:

- Using a property editor
- Using Customizers

Property editors are usually supplied by the builder tool, although you can write your own property editor. In this section, we shall discuss how to customize a bean's appearance and behavior using customizers.

A property editor usually displays all properties that can be manipulated. Sometimes, it is not a good idea to display a large number of properties, most of which are irrelevant. A Customizer class gives us complete control to configure and edit beans. Customizers are used when a specific instruction is needed and property editors are too primitive for this purpose. A Customizer class must do the following:

- A Customizer must be some kind of AWT component that is suitable for display in a dialog box created by the BeanBox. So, it must extend `java.awt.Component` or one of its subclasses. Typically, the Customizer class extends `Panel`.
- It must implement the `Customizer` interface. This interface looks like this:

```
public interface java.beans.Customizer{
    public abstract void setObject(java.lang.Object);
    public abstract void addPropertyChangeListener(java.beans.PropertyCha
ngeListener);
    public abstract void removePropertyChangeListener(java.beans.Property
ChangeListener);
}
```

So, the Customizer class must implement those properties. It must also fire `PropertyChangeEvent` to all registered listeners when a change to the target Bean has occurred.

- Implement a default constructor
- Associate the customizer with its target class via `BeanInfo.getBeanDescriptor`

If a Bean that has an associated Customizer is dropped into the BeanBox, a "Customize..." item is added by the BeanBox in the Edit menu.

22.10 JAVABEANS API

JavaBean API provides a set of related interfaces and classes necessary to design and develop beans in a separate package, `java.beans`. Not all the classes and interfaces are used all the time to develop a bean. For example, the event classes are used by beans that fire property and vetoable change events. However, most of the classes in this package are used by a bean builder/editor. In particular, these classes and interfaces help the bean builder/editor to create user interfaces that the user can use to customize their beans. Tables 22.1, 22.2, and 22.3 respectively show the list of interfaces, classes, and exceptions, which are used to develop beans.

Table 22.1 Java bean interfaces

Interface Name	Description
AppletInitializer	It is designed to work in collusion with java.beans.Beans.instantiate.
BeanInfo	A bean implementor who wants to provide information about their bean explicitly may provide a class that implements this BeanInfo interface.

(Contd.)

(Contd.)

Customizer	It provides a complete custom GUI for customizing a target Java Bean.
DesignMode	It is intended to be implemented by, or delegated from, instances of BeanContext, in order to propagate to its nested hierarchy of BeanContextChild instances, the current 'designTime' property.
ExceptionListener	It is notified of internal exceptions.
PropertyChangeListener	This event gets fired whenever a bean changes a "bound" property.
PropertyEditor	It provides support for GUIs that want to allow users to edit a property value of a given type.
VetoableChangeListener	This event gets fired whenever a bean changes a "constrained" property.
Visibility	A bean may be run on servers where a GUI is not available under some circumstances.

Table 22.2 Java bean classes

Class Name	Description
BeanDescriptor	It provides global information about a bean, including Java class and displayName.
Beans	It provides some general purpose methods to control beans.
DefaultPersistenceDelegate	It is an implementation of the abstract PersistenceDelegate class and is the delegate used by default for classes about which no information is available.
Encoder	A class that is used to create streams or files which encode the state of a JavaBean collection in terms of its public APIs.
EventHandler	It provides support for dynamically generating event listeners, whose methods execute a statement involving an incoming event object and a target object.
EventSetDescriptor	It describes a group of events that a specified Java bean fires.
Expression	It represents a primitive expression where a single method is applied to a target and a set of arguments to return a result.
FeatureDescriptor	It is the common baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc.
IndexedPropertyChangeEvent	This event gets delivered whenever a component that conforms to the JavaBeans specification (a "bean") changes a bound indexed property.
IndexedPropertyDescriptor	It describes a property that acts as an array and has an indexed read and/or indexed write method to access specific elements of the array.
Introspector	This class provides a standard way for buider tools to learn about the properties, methods, and events that a target Java Bean supports.
MethodDescriptor	It describes a particular method that a Java Bean supports for external access.
ParameterDescriptor	It allows bean implementors to provide additional information for each parameter.
PersistenceDelegate	It takes the responsibility for expressing the state of an instance of a given class in terms of the methods in the class's public API.
PropertyChangeEvent	This event is delivered when a "bound" or "constrained" property is changed.
PropertyChangeListenerProxy	It extends the EventListenerProxy to add a named PropertyChangeListener.
PropertyChangeSupport	It is a utility class that can be used by beans that support bound properties.

(Contd.)

(Contd.)

PropertyDescriptor	It describes a property that a Java Bean exports through accessor methods.
PropertyEditorManager	It is used to locate a property editor for any given type name.
PropertyEditorSupport	It helps to build property editors.
SimpleBeanInfo	This is a support class to make it easier for people to provide BeanInfo classes.
Statement	A Statement object represents a primitive statement in which a single method is applied to a target and a set of arguments.
VetoableChangeListenerProxy	It extends the EventListenerProxy specifically for associating a VetoableChangeListener with a "constrained" property.
VetoableChangeSupport	It is a utility class that is used by beans that support constrained properties.
XMLDecoder	The XMLDecoder class is used to read XML documents created using the XMLEncoder and is used just like ObjectInputStream.
XMLEncoder	The XMLEncoder class is a complementary alternative to ObjectOutputStream and can be used to generate a textual representation of a JavaBean in the same way that the ObjectOutputStream can be used to create binary representation of Serializable objects.

Table 22.3 Java bean exceptions

Exception Name	Description
IntrospectionException	It is thrown when an exception happens during Introspection.
PropertyVetoException	It is thrown when a proposed change to a property represents an unacceptable value.

22.11 EJB

Enterprise JavaBeans (EJB) is a Java-based comprehensive component architecture for design and development of world-class distributed modular enterprise applications. EJBs are not only platform-independent, but also they can run in any application server that implements EJB specifications. The EJB component architecture integrates several enterprise-level requirements such as distribution, transactions, security, messaging, persistence, and Enterprise Resource Planning (ERP) systems.

22.11.1 Benefits of EJB

Unlike other distributed component technologies such as CORBA and Java RMI, the EJB architecture hides most of the underlying system-level semantics such as instance management, object pooling, connection pooling, and thread management. The EJB container performs these tasks on behalf of us.

In the EJB architecture, beans contain the business logic, not the client. Therefore, the client developer can devote more time to the presentation of the client. Since clients do not have to implement business logic, they are thinner. This is very much required if devices running client applications are small and resource-constrainted.

EJBs are portable and can run on any EJB-compliant server.

It also provides us different types of components for business logic, session, persistence, and enterprise messages.

22.11.2 Usage Scenario

The EJB architecture should be used for those applications that have any of the following requirements:

- The applications must be scalable. The application's components need to be distributed across multiple machines to accommodate a large number of users. EJBs can run on different machines. Moreover, their location will remain transparent to the clients.
- Transactions are required to ensure data integrity. Enterprise beans allow us to perform transactions in a safe way. They also provide mechanisms for accessing shared objects concurrently.
- The application will have a wide variety of clients. Remote clients can easily locate enterprise beans using just a few lines of code. These clients can be thin, various, and numerous.

22.11.3 EJB Architecture

Before discussing the EJB architecture, let us understand the fundamental requirements of the distributed component architecture.

- It must provide mechanisms to instantiate the server-side and client-side proxies. A client-side proxy is created at the client side that represents the actual remote server object and acts as a proxy. A server-server side proxy, on the other hand, is created at the server-side. It must provide basic mechanism to accept client requests and delegate these requests to the object implementation.
- It must provide a mechanism that allows us to have a reference to the client-side proxy. Using this reference, clients invoke methods. The client-side proxy is responsible for communicating with the server-side proxy.
- It must support a mechanism that can be used to inform the system that a specific component is no longer needed.

To satisfy these requirements, the EJB architecture specifies two types of interfaces: `javax.ejb.EJBHome` and `javax.ejb.EJBObject`. The `javax.ejb.EJBHome` defines the methods that allow a remote client to create, find, and remove EJB objects, as well as home business methods that are not specific to a bean instance. On the other hand, `javax.ejb.EJBObject` defines methods that collectively provide the remote client view of an EJB object.

There are three kinds of beans in the EJB architecture: *Session Beans*, *Entity Beans*, and *Message Driven Beans*.

22.11.4 Session Beans

Session beans perform specific tasks for a client. They are plain remote objects meant for abstracting business logic.

A client invokes methods of the session bean to access an application, which is deployed on the EJB server. The session bean performs business tasks inside the server on behalf of the client, hiding all the complexities from the client.

A session bean, as its name suggests, can be considered as an interactive session. A session bean is not shared. It can handle a single client in the same way as an interactive session that has just one client. A session bean, like an interactive session, is also not persistent. This means that its state is not saved to a database. When the client finishes, its session bean is also terminated. There are two types of session beans: *stateless* and *stateful*.

22.11.4.1 Stateless session beans

A stateless session bean does not keep the state between client requests. When a client sends a request for method invocation, the EJB container assigns an instance of a stateless bean. During the method invocation, bean's instance variables may have a state, but that is valid only during the method invocation. When the client makes another request, the same instance may not be assigned to the client possibly due to the reason that it is already assigned to another client.

Since a stateless bean can be assigned to any client, it supports multiple clients. Therefore, to support the same number of clients, a lesser number of stateless session beans is required than stateful session beans. Session beans are useful for applications that have a large number of clients.

22.11.4.2 Stateful session beans

The state of a bean consists of values of its instance variable. Unlike stateless session beans, stateful session beans maintain their states. It means that a client sees the same state of the bean when it interacts with the stateful session bean through multiple method invocation. This is accomplished by assigning the same session bean to the client across multiple requests.

The stateful session bean loses its state when a client terminates. When the client terminates, the state is no longer necessary.

22.11.5 Entity Beans

Entity beans model real world business objects such as customers, orders, and products. They contain business logic that can be saved in a persistent storage for later use. The state of an entity bean persists beyond the lifetime of the application or the EJB server process. In J2EE, the persistency is obtained using a relational database. Usually, each entity bean has an underlying table in a relational database. Each instance of the bean corresponds to a row in that table.

Persistency can be obtained in two ways: *bean-managed* and *container-managed*. In the bean-managed persistency mechanism, the entity bean itself contains the code to save its state in the underlying table. On the other hand, in the container-managed persistence mechanism, the EJB container generates the necessary database access calls automatically.

Container-managed entity beans are not tied to a specific database. Therefore, you can redeploy the same entity bean on different J2EE servers that use different databases without modifying or recompiling the bean's code.

Multiple clients may share entity beans. So, entity beans work within transactions. The EJB container usually supports transaction management. In this case, you only have to specify the transaction attributes in the bean's deployment descriptor. The EJB container will take care of the rest of the procedure.

22.11.6 Message Driven Beans

A Message Driven Bean acts as a listener for the Java Message Service API, processing messages asynchronously. It is similar to an event listener, except that it receives messages instead of events.

The messages may be sent by a wide range of entities such as J2EE components, application clients, other enterprise beans, web components, JMS applications, or even systems that do not use J2EE technology. Message Driven Beans currently process only JMS (Java Message Service) messages, but provisions are left so that they can process other kinds of messages in the future.

The message-driven beans differ from session and entity beans in the sense that clients do not access message-driven beans through interfaces. Message Driven Beans and stateless session beans are similar with respect to the following points:

- They do not maintain state for a client. However, sometimes they can contain state across the handling of client messages such as an open database connection, an object reference to an enterprise bean object, or a JMS API connection.
- All instances are similar. The EJB container can use any instance to process a message. So, it can process multiple messages concurrently.
- One instance of a Message Driven Bean can handle multiple requests.

Note that, session and entity beans allow us to send and receive JMS messages. However, it happens in a synchronous way. We should not use synchronous 'send–receive', which are costly and blocking in a server-side component. Instead, message-driven beans can be used to send and receive messages asynchronously.

22.12 INTRODUCTION TO STRUTS FRAMEWORK

Struts is an application framework for developing Java EE web applications. This is an open-source framework and was originally developed by Craig McClanahan and donated to Apache. Fundamentally, it uses and extends Java servlet API and supports Model–View–Controller (MVC) architecture. It allows us to create flexible, extensible, and maintainable large web applications based on standard technologies, such as Java servlets, JSP, JavaBeans, resource bundles, and XML.

22.12.1 Basic Idea

In a Java EE-based web application, a client usually submits data to the server using the HTML form. These data are then handed over to a servlet or a Java Server Page, which processes them, typically interacts with the database, and finally processes the HTML output that is sent to the client. Since servlet as well as JSP technologies mix application logic with the presentation, they are inadequate for large web projects.

The primary task of struts is to separate application logic that interacts with the database, called *model* from HTML pages that are sent to the client, called *view* and instance that passes information between model and view, called *controller*. For this purpose, struts provides a controller as a servlet, called `ActionServlet`, and allows the writing of templates for the presentation (view), typically in terms of JSP. This special servlet acts as a switchboard to route requests

from clients to the appropriate server page. This simple idea makes web applications much easier to design, develop, and maintain. Figure 22.10 describes the struts framework.

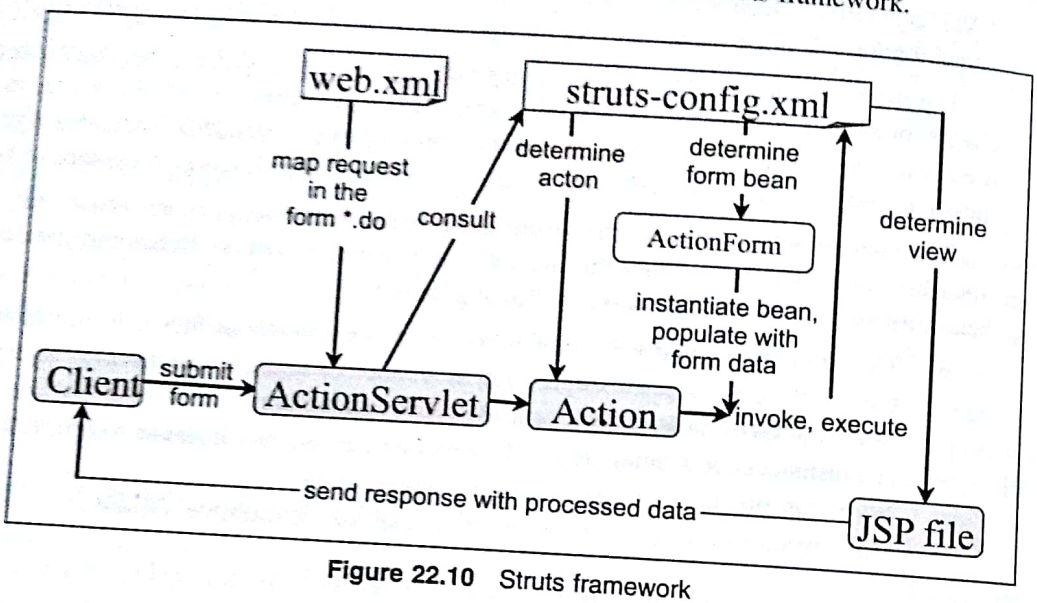


Figure 22.10 Struts framework

That was a high-level description of the struts framework. Let us now discuss the framework technically in more detail.

Every web application has an associated XML file (`WEB-INF/web.xml`) called deployment descriptor. This file must be configured by the web application developer. The deployment descriptor file specifies the configuration of the web application such as servlet mapping, parameter to those servlets, welcome pages, etc. This `web.xml` file is first configured to forward [Figure 22.9] all requests with a specified pattern (usually `*.do`) to the struts framework's `ActionServlet` using the following entry:

```
<servlet>
   <servlet-name>action</servlet-name>
      <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
      <init-param>
         <param-name>config</param-name>
         <param-value>/WEB-INF/struts-config.xml</param-value>
      </init-param>
      <init-param>
         <param-name>debug</param-name>
         <param-value>2</param-value>
      </init-param>
      <init-param>
         <param-name>detail</param-name>
         <param-value>2</param-value>
      </init-param>
      <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
```

```
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

If you use an IDE such as NetBeans, these entries are automatically inserted. The web.xml file also specifies one or more configuration XML file (struts-config.xml) that the ActionServlet will consult. This ActionServlet performs switching functions as mentioned earlier.

Suppose we are developing a web application using /myapp as a location relative to the server's document root. In the struts-config.xml file, we associate paths with the controller components called Action classes. Following is an example:

```
<action-mappings>
   <action name="LoginForm" path="/login" scope="request"
   type="com.myapp.struts.LoginAction" validate="false">
   </action>
</action-mappings>
```

It specifies that the ActionServlet should invoke the controller component com.myapp.struts.LoginAction for the URL http://mydost/myapp/login.do. We also give a name to this association ("LoginForm" in our case) to refer to it further. Note that the web.xml file specified that for the *.do pattern URL, ActionServlet should be invoked by the web container. So, the .do in this URL causes the web container to invoke ActionServlet of the struts framework. This ActionServlet consults the struts-config.xml file and sees the path "login" and invokes LoginAction. The form data are temporarily stored in an ActionForm type bean. The developer has to write this bean by extending the ActionForm class explicitly. Action and ActionForm are bound using the name property of the <form-bean> tag. Following is an example:

```
<form-beans>
   <form-bean name="LoginForm" type="com.myapp.struts.LoginForm"/>
</form-beans>
```

This "LoginForm" is the name of the action defined earlier and com.myapp.struts.LoginForm is the ActionForm bean.

For each Action, we can specify the names of the resulting page(s) that has(ve) to be sent to the client as a result of that action. There can be more than one view as the result of an action. Typically, there are at least two: one for "success" and one for "failure". This is accomplished by returning a string from the execute() method of Action to the ActionServlet.

The action form, in turn, can retrieve data sent by the client from ActionForm bean. The following example illustrates this:

```
public class LoginAction extends org.apache.struts.action.Action {

   /* forward name="success" path="" */
   private final static String SUCCESS = "success";
   /* forward name="failure" path="" */
   private final static String FAILURE = "failure";

   /**
    * This is the action called from the Struts framework.
```

```
     * @param mapping The ActionMapping used to select this instance,
     * @param form The optional ActionForm bean for this request.
     * @param request The HTTP Request we are processing.
     * @param response The HTTP Response we are processing.
     * @throws java.lang.Exception
     * @return
     */
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
      // extract user data
      LoginForm formBean = (LoginForm) form;
      String name = formBean.getName();
      String email = formBean.getEmail();

      // perform validation
      if ((name == null) || // name parameter does not exist
         email == null || // email parameter does not exist
         name.equals("") || // name parameter is empty
         email.indexOf("@") == -1) {    // email lacks '@'
         formBean.setError();
         return mapping.findForward(FAILURE);
      }
      return mapping.findForward(SUCCESS);
    }
}
```

The parameter `mapping` in the `execute()` method refers to the `ActionServlet` and `form` refers to the associated `ActionForm` bean. The `Action` (the controller component) reports back to the `ActionServlet` using words like "success", "failure", "ready", "OK", "UserError", etc. We specify corresponding view pages in the configuration file as follows:

```
<action name="LoginForm" path="/login" scope="request"
type="com.myapp.struts.LoginAction" validate="false">
   <forward name="success" path="/success.jsp"/>
   <forward name="failure" path="/login.jsp"/>
</action>
```

The `ActionServlet` knows from the configuration file where to forward and what to show as a result. This has the added advantage of reconfiguration of the view layer by simply editing the XML configuration file.

The `ActionForm` beans can be validated ensuring that the user provides valid data in the form. The validation can be carried out on a per-session basis. It allows forms to span multiple pages of the View and Actions in the Controller.

Since the framework works on the server side, the client's view has to be composed at the server before sending it to the client. Typically, some sort of server-side technology such as JSP, Velocity, and XSLT is used for the view layer. Simple HTML files may be used to compose client's view. However, they cannot provide full advantage of all of the dynamic features.

KEYWORDS

Accessor methods: Public methods of a bean used to read and write bean properties.

Bean builder: A tool that is used to build and/or test beans.

BeanInfo interface: The `BeanInfo` interface defines several methods that can be used to inspect properties, methods, and events of a bean.

Bound property: A bound property of a bean is one which when changed, notifies one or more beans about this change.

Constraint property: A constrained property of a bean is a special type of property which can be changed subject to prior permission from external object(s).

Controller: Controls the Model and View components.

Customization: Bean customization allows us to customize the appearance and behavior of a bean at design time, within a bean-compliant builder tool.

Design Patterns: The JavaBeans framework introduces several rules for names to be used for classes and methods. They are collectively called design patterns.

EJB: Enterprise JavaBeans (EJB) is Java-based comprehensive component architecture for design and development of world-class distributed modular enterprise applications.

Entity Beans: Entity beans model real world business objects such as customers, orders, and products. They contain business logic that can be saved in a persistent storage for later use.

Indexed Property: An index property is an array of properties. It can hold a range of values that can be accessed through accessor functions.

Introspection: A procedure to discover properties, methods, and events of a bean dynamically.

Message Driven Beans: A Message Driven Bean acts as a listener for the Java Message Service API, processing messages asynchronously.

Model: Specific code that handles the business logic.

Persistence: Persistence is a procedure to save a bean in non-volatile storage such as a file. The bean can be reconstructed and used later.

Session Beans: Session beans perform specific tasks for a client. They are plain remote objects meant for abstracting business logic.

Stateful session beans: Stateful session beans maintain their states across client requests.

Stateless session beans: A stateless session bean does not keep the state between client requests.

View: Takes care of the appearance of web pages.

SUMMARY

J2EE is not a single technology; it consists of a large set of technologies.

JavaBean technology is a Java-based technology to design and develop reusable and platform-independent software components. These components can then be integrated virtually in all Java-compatible applications. JavaBean components are knows as beans. The appearance and features of a bean can be changed or customized using builder tools such as BeanBox and NetBeans. A bean class is nothing but a usual Java class with the following requirements:

- It must have a public no-argument constructor.
- It has public accessor methods to read and write properties.
- It should implement `Serializable` or `Externalizable` interface.

Properties control the behavior and appearance of a bean. Usually, beans are developed and tested using tools called bean builder. A bean builder provides several facilities to build a bean.

A bean builder usually uses Java core reflection API to discover the methods of a bean. It then applies design patterns to discover other bean features (such as properties and events). This procedure is known as introspection. The JavaBeans framework introduces a lot of rules for names to be used for classes and methods. They are collectively called design patterns. The different types of bean properties, namely, simple, bound constraint, and indexed properties were explained with extensive examples. JavaBean API provides interfaces and classes to discover properties, methods, and events of a bean. The `BeanInfo` interface defines several methods that can be used to inspect properties, methods, and events of a bean. Sometimes, it is necessary to store the beans for later use. Persistence is a procedure to save a bean in non-volatile storage such as a file. The bean can be reconstructed and used later.

Bean customization allows us to customize the appearance and behavior of a bean at design time, within a bean-compliant builder tool.

Enterprise JavaBeans (EJB) is Java-based comprehensive component architecture for design and development of world-class distributed modular enterprise applications. EJBs are not only platform-independent, but also run in any application server that implements EJB specification. The EJB component architecture integrates several enterprise-level requirements such as distribution, transactions, security, messaging, persistence, and Enterprise Resource Planning (ERP) systems. There are three kinds of beans in EJB architecture: Session Beans, Entity Beans, and Message Driven Beans.

Session beans perform specific tasks for a client. They are plain remote objects meant for abstracting business logic. A stateless session bean does not keep the state between client requests. Unlike stateless session beans, stateful session beans maintain their states. Entity beans contain business logic that can be saved in a persistent storage for later use. A Message Driven Bean acts as a listener for the Java Message Service API, processing messages asynchronously.

Struts is an application framework for developing Java EE web applications. It uses and extends Java servlet API and supports Model–View–Controller (MVC) architecture. It allows us to create flexible, extensible, and maintainable large web applications based on standard technologies, such as Java servlets, JSP, JavaBeans, resource bundles, and XML.

WEB RESOURCES

`http://java.sun.com/docs/books/tutorial/javabeans/`
Trail: JavaBeans

`http://java.sun.com/j2ee/1.4/docs/tutorial/doc/`
The J2EE 1.4 Tutorial

`http://www.ibm.com/developerworks/edu/j-dw-java-gsejb-i.html`
Getting started with Enterprise JavaBeans technology

`http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts.html`
Enterprise Beans

`http://struts.apache.org/`
Struts

`http://netbeans.org/kb/docs/web/quickstart-webapps-struts.html`
Introduction to the Struts Web Framework

`http://www.vaannila.com/struts-2/struts-2-tutorial/struts-2-tutorial.html`
Struts 2 Tutorial

EXERCISES

Multiple Choice Questions

1. How to make an instance variable into a JavaBean property?
   (a) Drop the Bean into the BeanBox property sheet
   (b) Specify get and set methods for the variable
   (c) Declare the instance variable public and static
   (d) Declare the instance variable private and let the BeanBox define get and set methods for the variable

2. JavaBean methods are all
   (a) Properties
   (b) Event listeners
   (c) Identical to methods of other Java classes
   (d) Events

3. Which of the following aspects do properties control?
   (a) How a JavaBean is compiled and dropped into the BeanBox (or other tool)
   (b) The tools you can use to customize a JavaBean
   (c) The communication between JavaBeans
   (d) A JavaBean's appearance and behavior

4. Which of the following is done using BeanBox's property tool?
   (a) Change the value of a Bean's property
   (b) Add new properties to a Bean
   (c) Delete a Bean's property
   (d) None of the above