CHAPTER - 21

JAVA SERVER PAGES (JSP)

KEY OBJECTIVES:_____

After completing this chapter, the readers will 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 different components of JSP pages
- get an overview about methods used to track sessions
- learn how to create and use JavaBean components
- learn how to use custom tags
- learn how to use JDBC technology

21.1 Introduction and Marketplace

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

JSP is nothing but high-level abstraction of Java servlet technology. It allows us to directly embed pure Java code in a 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 the 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, faster and maintenance also becomes very easy.

The 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 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 dynamic content of our web pages from static HTML content. We can write regular HTML files in usual way. We can then insert Java code for dynamic parts using special tags which usually starts with <% and ends with %>.

The JSP files have normally extension .jsp. They are also installed in a place where we could place our normal web pages. Many web servers let us define aliases of JSP page or servlet. 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 support 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 servlet has.

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

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, 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 JSP files and how to access them from web browser using URL. We shall use 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 out 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 root directory of you JDK is D:\Java\jdk1.6.0_10, the JAVA_HOME environment variable should point 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 the figure 21.1

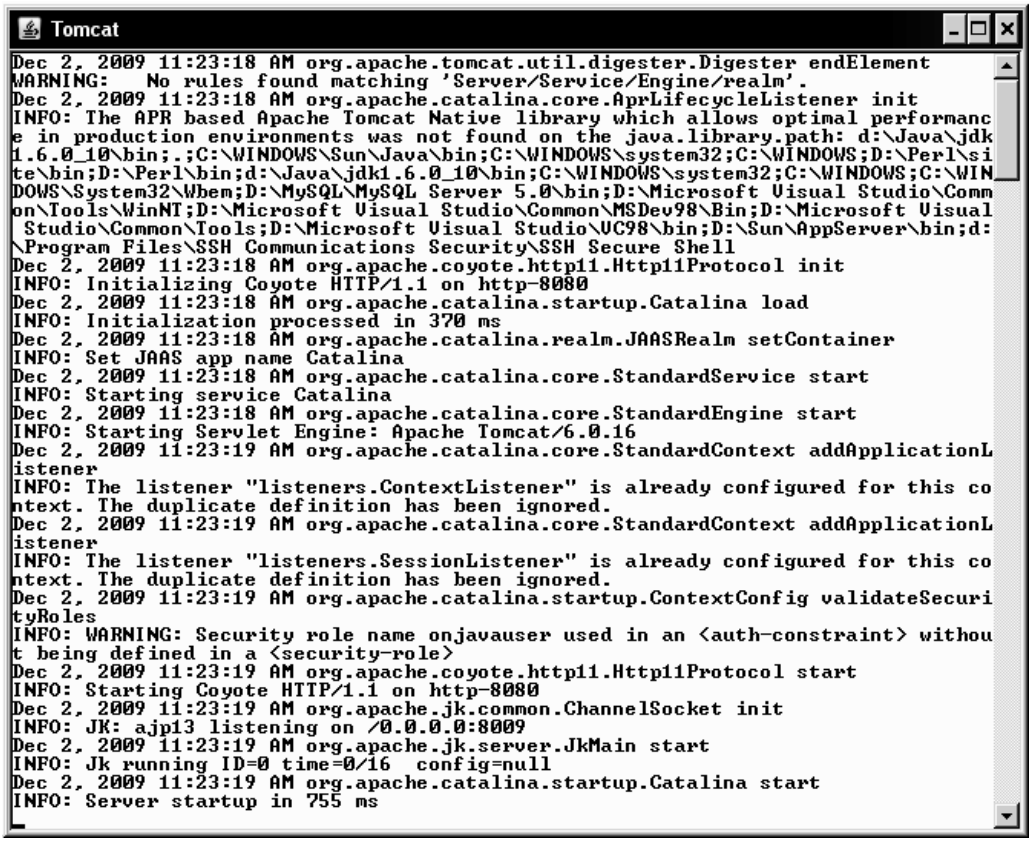


Figure 21.1: Starting tomcat server

- 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 as shown in the figure 21.2 appears.

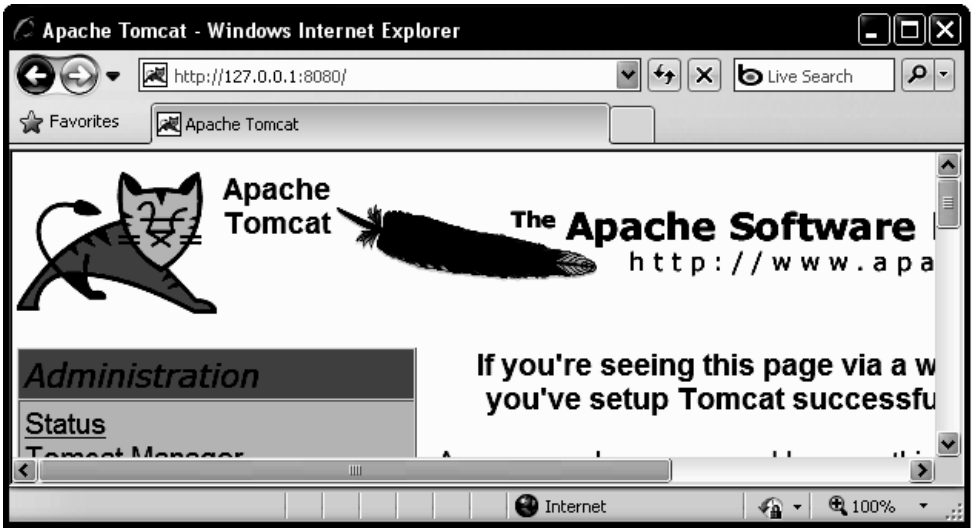


Figure 21.2: Tomcat home page

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 wide range of technologies such as JSP and Java Servlet technologies, PHP and CGI.

21.3.3 WebLogic

The 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 .NET framework for interoperability and allows integration of following native components:

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

The 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. And 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 class file.
- loads the class file and creates an instance.
- Initializes the servlet instance by calling the `jspInit()` method.
- Invokes the `_jspService()` method, passing a request and response object.

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

- If an instance is already running, it simple 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 memory.

21.5 JSP and Servlet

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

In most of the servlets, a small piece of code is written to handle application logic and large code is written using too many `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.

But, JSP separates the static presentation templates from 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 **J**ava **R**untime **E**nvironment (JRE).

Another problem of servlet is that, each time the servlet code is modified, it needs to be recompiled and the web server also needs to be restarted. JSP engine takes care of all these issue automatically. Whenever a JSP code is modified, JSP engine identifies it and translates to 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 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 more role. JSP pages can be moved easily across web servers and platforms, without any changes and without any significant efforts.

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.

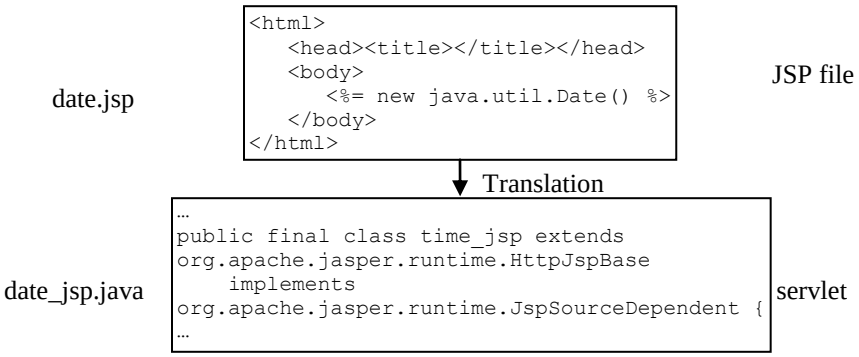


Figure 21.3: JSP to servlet translation

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

The JSP engine returns a `ParseException` if an error occurs during the translation phase and such case 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, 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. But, don't worry, we shall about those things through 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 a simple JSP page that prints temperature conversion chart from Celsius to Fahrenheit. If you open this page in the Internet Explorer browser, it looks like as shown in the figure 21.4

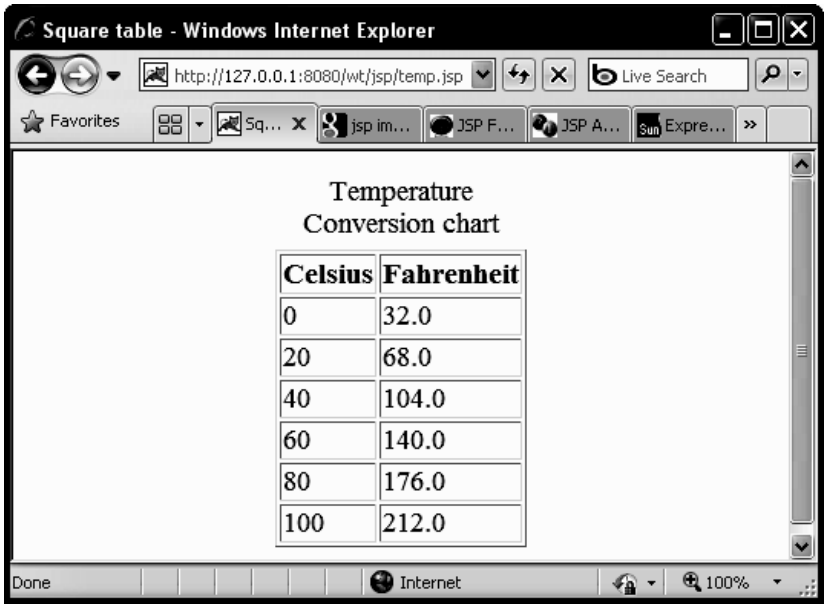


Figure 21.4: Temperature Conversion Chart

When you request this page for the first time, JSP engine translates the above JSP page into 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()).g
etExpressionFactory();
    _jsp_annotationprocessor = (org.apache.AnnotationProcessor)
getServletConfig().getServletContext().getAttribute(org.apache.AnnotationProces
sor.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()` corresponds to servlet life cycle methods `init()`, `service()` and `destroy()` methods respectively which were discussed in the previous chapter. The `_jspService()` method is responsible to serve client's requests. JSP specification prohibits overriding of `_jspService()`

method. However, the developers are allowed to override `jspInit()` and `jspDestroy()` methods within their JSP pages to initialize and shutdown servlets respectively.

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

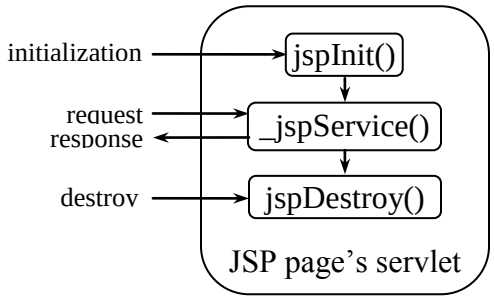


Figure 21.5: Life cycle of JSP's servlet

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

21.6 The Anatomy of a JSP Page

A JSP page consists of basically 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 five 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. The *actions* allow us use existing components, and otherwise control the behavior of the JSP engine.

There are three kinds of scripting elements; *scriptlets*, *declarations* and *expressions.* Scriptlets allows 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 use 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 tag.
- 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 the special character such as '%', add a '\' character before it. To use '\' character, add another '\' character before it.

21.8 JSP Components

A JSP page consisting of following components:

- Directives
- Declarations
- Expressions
- Scriptlets
- Actions

JSP tag	Meaning
<%@...%>	Used for JSP directives such as page, include etc.
<%=...%>	Used for JSP expressions
<%...%>	Used for JSP scriptlets which can contain arbitrary Java statements
<%!...%>	Used for variable, method and inner class declaration

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

21.8.1.1 page directive

Following is the syntax of `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 `import` attribute, no attribute should appear more than once. The bold face value indicate the default value. 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 comma (,) to be imported by JSP file. To import all the classes of a package, use ".*" at the end of the package name. 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 above line of code 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 the default value), 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 and such case output is written directly to the output stream of the response object of the servlet.

autoFlush

Syntax:

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

The `autoFlush` attributes specifies whether the buffer should be flushed automatically (`true`) when the buffer is full. If set to `false`, an 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 JSP container is allowed to send multiple concurrent client request 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, JSP container sends clients requests one by one using a single thread. This attribute basically provides information to the underlying servlet whether it should implement `SingleThreadModel` interface or not. The tomcat generates the following servlet class declaration if `false` is specified.

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

SingleThreadModel

In this case, JSP container loads multiple instances of the servlet in the memory. The JSP container then distributes concurrent client requests evenly among these servlet instances for processing in a round-robin fashion

If the `true` is specified following code is generated:

```
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 retrieved using `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 anything uncaught exception occurs in a JSP page, JSP container sends a information explaining the exception which is undesirable in commercial sites. The `errorPage` attribute specifies a 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 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 a HTML or text file, its content is directly included at the place of include directive. 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 included in the JSP page.

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

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

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

21.8.2 Comments

Comments are used to document JSP pages and can be inserted anywhere in the JSP page. 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 following HTML comment. This 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 small piece of data in a JSP page without using `out.print()` or `out.write()` statements. It is a faster,

easier and clearer way to display values of variables/parameters/expressions in a JSP page. The general syntax of 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 at the place of 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 above 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 by anything as long as it can be converted to a string. For example 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 arbitrary piece of Java code using 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 or class declarations, or expressions or other language statements. Scriptlets have the following form:

```
<% scriptlets %>
```

For example, 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);
…
```

which results 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. Following example illustrates this:

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

The above code gets converted to 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 that 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 client requests. So, variables declared in this way become instance variables of the underlying servlet.

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

This is translated in the servlet like this:

```
…
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 when the servlet is instantiated. This sometimes useful but sometimes not desirable. Following declarations creates as 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 following piece of code displays the number of time the JSP page is referred after the page was 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 translation unit. It is valid in the JSP file as well as all the files included statically. Declaration is not valid in a 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 or 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 object in the next section. The JSP objects created using `<jsp:useBean>` tag have also page scope.

request

A request can be served by more than one page. Objects having *request* scope can be accessed from any pages 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 pages that belong to the same application. 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 21.1

Table 21.1: JSP Implicit Objects

Variable	Class	Description
Out	javax.servlet.jsp.JspWriter	The output stream of the JSP page's servlet.
Request	Subtype of javax.servlet.ServletRequest	It refers to the current client request being handled by the JSp page.
response	Subtype of javax.servlet.ServletResponse	The 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	The session object for the client.
application	javax.servlet.ServletContext	The context of the JSP page's servlet and other web components contained in the same application.
exception	java.lang.Throwable	Represents error. Accessible only from an error page.
Page	java.lang.Object	It refers to JSP page's servlet processing the current request.
pageContext	javax.servlet.jsp.PageContext	The context of the JSP page which 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 `getParameter()` method) HTTP request type (GET, POST, HEAD etc), 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
```

JSP page can retrieve parameters sent through an HTML from. 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 `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 the figure 21.1

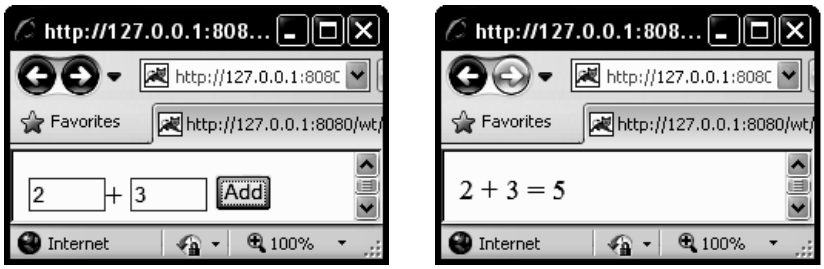


Figure 21.6: Retrieving form data from JSP page

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, cookies etc. 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, underlying servlet configuration etc. 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 this object cases an error . It is used to access session related information such as creation time, the id associated to this session etc. 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 only used 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 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 JSP page's servlet. This implicit object has page scope.

exception

This represents an uncaught exception which cases an error page to be called. This object is available in the JSP page for which `isErrorPage` attribute of `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 variable 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 using 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, following scriptlet is valid for the above declaration.

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

which displays

```
5
```

JSP allows us to declare variables and classes in the scriptlet section but not methods. Variables declared in the scriptlet section are local to the `_jspService()` method. They are created each time client makes a request and destroyed after the request is over. Similarly, classes declared in the scriptlet section becomes inner classes of `_jspService()` method. Consider the follow 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 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` 5 times and displays the value. The code segment works fine if only one request is dispatched to this JSP page at a time. But, if two or more requests are sent to this JSP page, result is not displayed correctly. To demonstrate this and to allow collision, an artificial delay is given. A sample output is shown in the figure 21.7 if two requests are sent simultaneously.

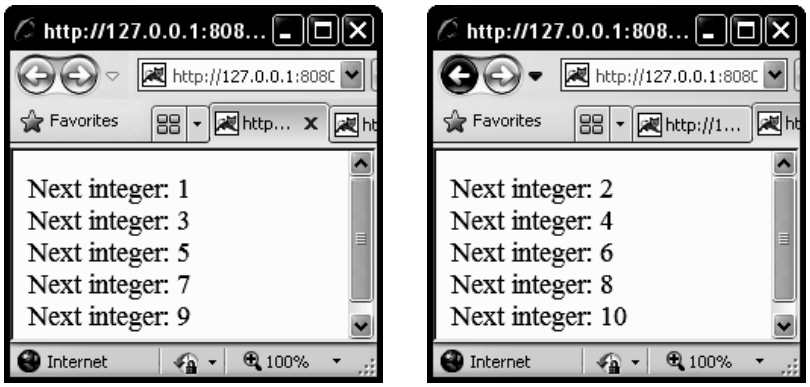


Figure 21.7: Accessing JSP page without synchronization

One of the solution of above 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 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 ease 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. But, JSP action tags are convenient way to use those functions. They also promote component framework as well as application maintainability. Following section describes commonly used JSP action tags.

include

This action tag provides an alternative way to include a file in a JSP page. General syntax of include action tag is:

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

For example, following code includes a 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 at 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 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 `<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 `page` attribute may be a dynamically generated file name. 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. And 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

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

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

The `name` and `value` attributes of `<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 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 `value` attribute may be a dynamic value but the value of `name` attribute must be static. 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. 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. General syntax of JSP forward action tag is:

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

The consider the `forward.jsp` file which forwards the current request to the `date.jsp` file using `<jsp:forward>` action as follows:

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

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

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

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

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

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

The <jsp:plugin> action tag has similar set of attributes as <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 Java plug-in fails to start due to any unavoidable reason.

Here is the source code for 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 the applet window.

As mentioned earlier, the web server generate either an <embed> or <object> tag depending upon the browser that requested the JSP page. For example, 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 `<jsp:plugin>` action tag, different vendor may implement it differently. For detail information, read the documentation of the web server.

useBean

It is used to instantiate a JavaBean object for later use. We shall discuss these actions later in this chapter and next chapter in detail.

setProperty

It is used to set a specified property of a JavaBean object.

getProperty

It is used to get a specified property from a JavaBean object.

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 an JSP page exactly like other HTML/XML tags using Java code. JSP engine interprets them and invokes their functionality. So, Java programmers can provide application functionality in term of custom tags while web designers can use them as building blocks without any knowledge about the Java programming. This way, JSP tag extensions provide a higher-level application specific approach to simple HTML authors.

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

21.8.10.1 Tag Type

JSP specification defines the following tags that 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 has attributes but no body. Example:

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

Tags with body

Tags that can contain other tags, scripting elements, text between the start and end tag. 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 the above 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 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 they are 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 else (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 `javax.servlet.jsp.tagext.Tag` interface which provides the framework of JSP tag extension. In this section, we shall develop a simple tag named `<ukr:hello>` that has a single optional attribute `name`. The `<ukr:hello>` tag prints "Hello <name>" or "Hello World!" depending upon the attribute `name` is specified or not.

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

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

which prints "Hello Monali" or

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

which prints "Hello World!". Here is the complete source code for 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 `javax.servlet.jsp.tagext.Tag` interface, it is convenient to use one of the default implementation `TagSupport` or `SimpleTagSuport` class. Our tag is one without any body and hence we have used `SimpleTagSupport` class. Each tag handler must define some predefined methods [Table 21.2] that are called by the JSP engine. For example, JSP engine calls `doStartTag()` and `doEndTag()` methods when start tag and end tag are encountered respectively. The class `SimpleTagSupport` implements most of the work of a tag handler. The implementation of our class is simple as it extends `SimpleTagSupport` class. We have only implemented `doTag()` method, which is called when JSP engine encounters start tag.

Table 21.2: 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. JSP engine sets the value of an attribute `xxx` using the method `setXxxx()` 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 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 a string "Hello" appended with the value of the `name` attribute to the servlet output stream. A reference to this output stream is obtained using `getOut()` method on `javax.servlet.jsp.JspContext` which is returned by `getJspContext()` method.

Deploy the tag

Save the above code in file `HelloTag.java` and put it in the application's `/WEB-INF/classes/tag` directory. To compile this file, we need necessary tag classes that are provided as jar file `jsp-api.jar`. This jar file can be found in `lib` directory of the 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 `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 is 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. 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 application's root directory is `wt`, put the `tags.tld` file in `/wt/WEB-INF/taglib` directory. Make th 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 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 a 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 others if they contain tags with 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
```

And for the following code

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

it displays

```
Hello World!
```

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

which 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 what 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 could have been using 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 flow control to tag handlers. The basic idea is to write a tag called iteration tags that iterates is body.

The iteration tag retrieves two parameters start and end. It calculates the factorial of each number between these two number (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 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 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 follow 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 count and fact variables are retrieved using expression language. Note that the JSP page does not contain a single piece of script. It displays as shown below:

```
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 `page` directive is set to `false`. An HTTP session is represented by the implicit object `session`, This `session` object has session scope and thus shared among all the pages within the session.

This session object can be used as 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 which 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 `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 int 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 a file `Factorial.java` and store it in the application's is `/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 add its subdirectories in the Java Runtime Environment (JRE). Those class files can now 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" />
```

where `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 `<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 following action is used:

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

This equivalent to 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 `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` attribute respectively. This is equivalent to calling `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 following:

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

The equivalent scriptlet is as follows:

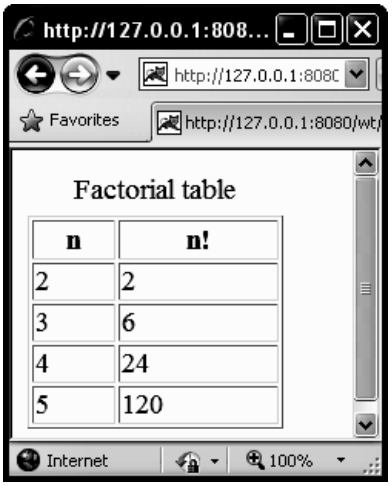
```
<%= fact.getValue() %>
```

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"
   />
   <%
   }
   %>
</table>
```

It generates the following code.



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 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()%>
```

which displays

```
720
```

21.10 Session Tracking

Since HTTP is a stateless protocol, web server can't remember previous requests. Consequently, web server can't relate the current request with the previous one. This makes a problem for some applications that require a sequence of related request-response cycle. Examples include online examination system, email checking, system, banking applications. How does the server know how may 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 user clicks on the *next* button, the server sends the integer next to the one displayed. Similarly, when *prev* button is clicked, 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 `<input>` tag with `type` attribute `hidden`. For example, following creates a hidden field.

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

The interesting part of 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 don't require great deal of information.

Let us now implement the application mentioned above using hidden field.

```
<%@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, 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, 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 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 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 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, JSP page should not pass sensitive information such as password in the hidden field.

21.10.2 URL Rewriting

This is another simple but elegant method to track sessions and is widely used. It also does not require any special support from 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 JSP page generates and sets the URL to be called for the hyperlinks “prev” and “next”. For example, 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 “prev” hyperlink, the integer that should be sent by the JSP page for this request is appended to the URL. JSP page simply extracts this information the sends it generating new URLs for those two hyperlinks. Similar things happens 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*. JSP page sends this cookie to the web browser. The browser must be configured properly to accept it. Upon receiving a cookie, 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 server can identify a session.

By default the most of the browsers support cookie. But, due to security reason, very often cookie support is disabled and that case session management will not work correctly. Following is the JSP page for the above application.

```
<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 `getValue()` method. Depending on the button clicked, the integer to be sent next is calculated and stored in a variable `current`. This integer together with a new cookie with name “last” having the current value set is sent to the web browser. But, it the JSP page does not find any cookie with the name “last”, which happens for the first time, it sends a new cookie having 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 `javax.servlet.http.HttpSession` interface. The `HttpSession` objects are just automatically associated with the client by cookie mechanism. An `HttpSession` object is one that persists during a session until it times out or it is shutdown by the JSP page participating this session.

In an JSP page, the an HTTP session is created by default if it is not suppressed by setting `session` attribute of `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 a `HttpSession` object explicitly using the following method in `HttpServletRequest`.

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

The first version takes 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 exists, otherwise the existing session object is returned. The parameter is false, it returns the existing session object (if exists), null otherwise. The second version simple 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 `getAttribute()` method is `Object`. So you must typecast it to 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 store with this key otherwise, the old value is overwritten with the specified value.

The JSP engine uses 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 client makes a request for the first time using HTTP response header `Set-Cookie` as follows:

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

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

```
Cookie: JSESSIONID=g52d15acu325dlw532h234
```

The server can then identify that this request has come from the same client. But, if the client does not accept cookies, this mechanism fails. In that case, programmers may use URL rewriting mechanism where each URL must have the session ID appended. 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 identifies whether client is configured to accept cookies. They append session ID only if 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 `sendRedirect()` method and `encodeURL()` method 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 necessary data passed to it. In this section, we shall discuss how to pass control to another page as pass data across pages.

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 email application, the `login.jsp` page, on user verification may want to forward the request to `home.jsp` page. This is done using `<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 `<jsp:forward>` action, it stops processing of current page and starts processing of the page (called target page) specified by the `page` attribute. The rest part of the original page is never processed further.

The `HttpServletRequest` and `HttpServletResponse` objects are passed to the target page. So, target page can access all information passed to the original page. You can pass additional parameters to the target page using `<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 using the same ways as original parameters using `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 as `page` scope in a page, are not available in another page where the control is transferred using `<jsp:forward>` action. If you make an object visible across multiple pages across same request, create the object as `request` scope. Following example illustrates this.

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

The above 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 it 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 cerate 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. But some times objects should be shared among multiple requests.

Think about on online examination system. JSP pages requested from same browser must share same user name that was provided during login procedure. This is required because different user has 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 the `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 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 starting time of the user 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` can retrieve the start time for this user can check whether the user has exhausted its time limit and take 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 window but using the same database. For efficiency purpose 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 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.

And here is the code fragment used in `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 application object to get `Statement` object and fire SQL query.

21.13 Database Connectivity

Most of the web applications need access to databases in the backend. **J**ava **D**ata**B**ase **C**onnectivity (JDBC) allows us to access databases through Java programs. It provides Java classes and interfaces to fire SQL, PL/SQL statements and process result (if any) and to perform other operations common to databases. Because Java Server Pages can contain Java code embedded in it, it is also possible to access database 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 rather 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 request to the 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 *Type 1* driver. This type of drivers can not talk to the database directly. It needs an intermediate ODBC (**O**pen **D**ata**B**ase **C**onnectivity) driver to 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 hence platform-dependent. To function this mechanism correctly, ODBC driver must be available at client machine and must also be configured correctly which is generally a long a tedious process. For this reason, Type 1 driver is used for experimental purpose or when no other JDBC driver is available. Sun provides a Type 1 JDBC driver with JDK 1.1 and later.

21.14.2 Native-API, Partly Java (Type 2)

This is very much similar to Type 1 driver. But, it does not forward the JDBC call to 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 native APIs. But, it has similar type of limitations as the previous one. This means client must have vendor specific native APIs installed and configured in it.

21.14.3 Middleware, Pure Java (Type 3)

In this case, JDBC driver forwards the JDBC calls to some middleware server using database independent network protocol. 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 specific database. Results are then sent back to intermediate server, which in turn sends the result back to client. This approach hides the details of connections to database servers and makes it possible to change database servers without affecting to 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. Because it talks with the database server directly, no other subsidiary driver is needed. In this book, We shall use only Type 4 driver.

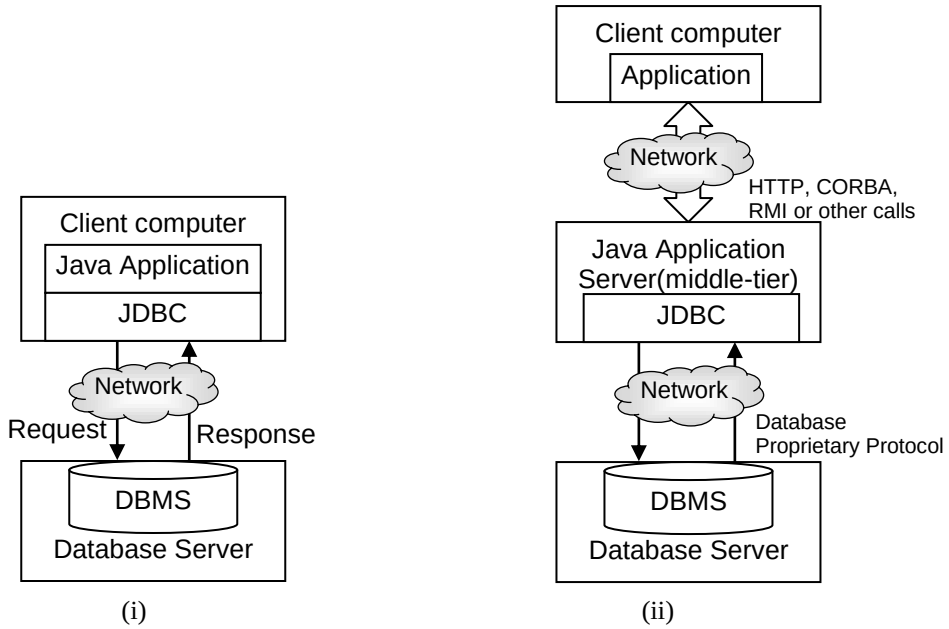


Figure 21.8: JDBC architecture (i) two tier (ii) three-tier

Table 21.3: 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
- Execute SQL statement

21.16 Loading a Driver

You have to first download appropriate driver depending upon the database you want to connect. Sun provides a Type 1 driver bundled with the JDK 1.1 and 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 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 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 and later is `ojdbc14.jar`.

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

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

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

One way to register a driver with the driver manager is to use the static method `forName` of Java class `Class` with a driver class name as an argument. For example 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 name `Driver.class` one in the `com.mysql.jdbc` package and another in `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 `DriverManager` class. The method `registerDriver()` in turn registers the driver with the driver manager. Some JDBC vendors such as Oracle recommend the later mechanism.

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

The similar procedure can be followed for other drivers as well. The implementation of 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 {}
}
```

Because, 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 JDBC call.

21.17 Making a connection

Once a driver is instantiated and registered with the driver manager, connection to the database can be established using methods provided by `DriverManager` class. For each connection created, `DriverManager` makes use of the appropriate driver registered to it. Following method are available on `DriverManager` class to establish a connection. All methods return a `Connection` object on successful creation of 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 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. The `DriverManager` uses this URL to find a suitable JDBC driver installed earlier that recognizes the URL and uses this driver to connect to the corresponding database.

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

```
jdbc:mysql://[host]:[port]/[database]
```

Here `host` is the name (or IP address) of machine running database at port number `port`. The `database` is a name of a database. Suppose a MySQL database `test` is running in a machine `thinkpad` at port `3306`, then 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, 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 21.4: 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
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 `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 (name, value) pair separated by "&" using same syntax as 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 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 `getConnection()` method. Following is an example using Properties.

```
String url = "jdbc:mysql://thinkpad:3306/test";
java.util.Properties p = new java.util.Properties();
p.setProperty("user", "root");
p.setProperty("password", "nbuser");
Connection con = DriverManager.getConnection(url, p);
```

21.18 Execute 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, committing or roll back a transaction etc. Following section describes different types of statement objects and their functionality.

21.19 SQL Statements

The `Connection` interface defines 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 `createStatement()` method on Connection object as follows:

```
Statement stmt = con.createStatement();
```

This `Statement` object defines following methods to fire different types of SQL commands on the database.

`executeUpdate()`

This method is used to execute DDL (CREATE, ALTER, 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. 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 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 statement such as SELECT. Remember, DQL statements only reads data from database tables; it can not change database tables. So, the return value of this method is a set of rows which is represented as a `ResultSet` object.

The result of `executeQuery` method is stored in an object of type `ResultSet`. This result set object looks very much similar to that of a table and hence has a number of rows. A particular row is selected by setting a cursor associated to the 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 methods provided by `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 cursor only in forward direction using method `next()`. JDBC 2.0 allows us to move the cursor both forward and backward direction as well as to move to a specified row relative to the current row. These types of result set are called scrollable result sets which will be discussed later in this chapter.

Because 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()` is used, where `X` is the data type of the column. Following is an example that retrieves information from `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 can not decide which method they should use. Such cases, `execute()` method is used. It can be used to execute any SQL commands. Because, it allows us to execute any SQL commands, the result can either be a `ResultSet` object or an integer. But, how does a user know it? Fortunately, this method returns a Boolean value which indicates return type. 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 obtain by calling `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 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>
```

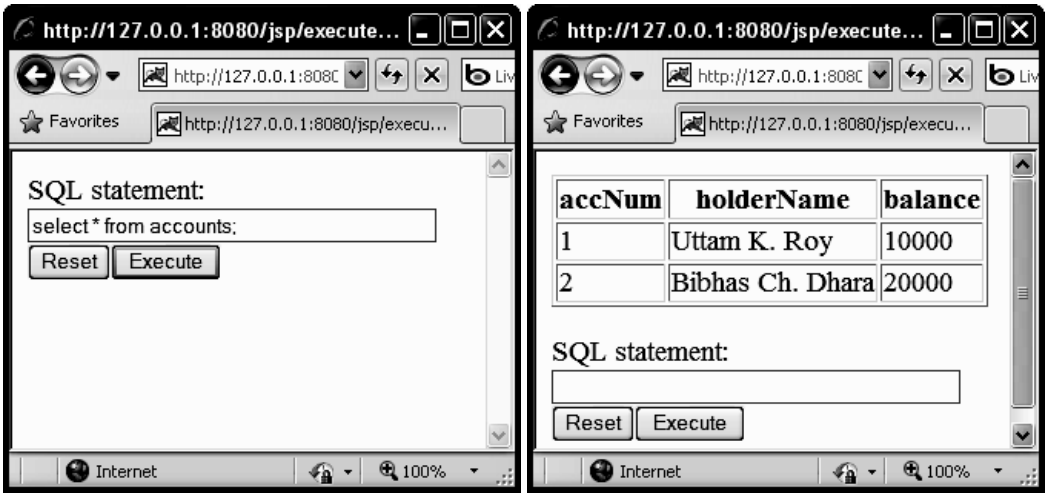


Figure 21.9: Executing SQL statements

Note that the above JSP page allows you to fire any valid SQL query including DML. So, JSP page must perform user verification before providing this interface.

21.19.2 Atomic transaction

The database transaction made by `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)
```

Bank manager wants to write a java program to transfer some amount of money `amount` from source account `src` to destination account `dst`. The basic task of this program will be to subtract `amount` from source account balance and add `amount` to destination account balance. A sample JSP page look 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. But, if anything goes wrong after updating the source account and before updating destination account in the `doTransfer()` method, destination account will hold incorrect balance.

This problem can be solved using `autoCommit()` method available on `Connection` object. First the `autoCommit` flag of `Connection` object is set to false. At the end of execution of all related statements transaction is committed. If anything goes wrong during the execution of those statements it can be caught and rolled back the transaction 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 `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 `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;
  }
}
```

Because 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 `Statement` object following steps are followed:

- 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 *same* SQL statement repeatedly with different data items. This incurs significant overhead that can be avoided using `PreparedStatement` object.

A `PreparedStatement` object is created using `prepareStatement()` method of `Connection` object. An SQL statement with place holders (?) is supplied to the method

`Connection.prepareStatement()` when a `PreparedStatement` object is created. This SQL statement together with the place holders is sent to the DBMS. DMBS in turn compiles the statement and if everything is correct, a `PreparedStatement` object is created. This means, a `PreparedStatement` object contains an SQL statement whose syntax has already been checked and hence called pre-complied statement. This SQL statement is then fired repeatedly with place holders substituted by different data items. Note that `PreparedStatement` is only useful if same SQL statement is executed repeatedly with different parameters. Otherwise it behaves exactly like `Statement` and no benefit can be obtained.

Following example creates a `PreparedStatement` object:

```
PreparedStatement ps = con.prepareStatement("INSERT INTO user values(?,?)");
```

The SQL statement has two place holders whose values will be supplied whenever this statement will be sent for execution. Placeholders are substituted using methods of the form `setX()` where `X` data type of the value used to substitute. These methods take two parameters; first indicates index of place holder to be substituted and second indicates the value to be used for substitution. Following example substitutes the first place holder by "user1".

```
ps.setString(1, "user1");
```

Consider a file `question.txt` that contains questions in the form `question_no:question_string` as follows:

```
1:What is the full form of JDBC?
2:How is a PreparedStatement created?
…
```

Following example inserts questions and their numbers stored in the above 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();
}
```

The `PreparedStatement` has another important role in executing parameterized SQL statements. Consider the above solution using `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();
}
```

The above 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 query will be

```
INSERT INTO questions values(3,'What's JDBC?')
```

which is an invalid query due the "'" character in the word "What's". If you try, you 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
```

The `PreparedStatement` can handle this situation very easily as it treats the entire parameter as input. So, the example given above using `PreparedStatement` will work correctly in this case.

21.19.4 SQL statements to call stored procedures

JDBC also allows calling stored procedures that are stored in the database server. This is done using `CallableStatement` object. A `CallableStatement` object is created using `prepareCall()` method on a `Connection` object.

```
CallableStatement prepareCall(String)
```

The method `prepareCall()` takes a primary string parameter that represents the procedure to be called and returns a `CallableStatement` object which is used to invoke stored procedures if 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 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, following syntax is used:

```
{call procedure-name}
```

Following syntax is used to call a procedure that takes single parameter:

```
{call procedure-name(?)}
```

If a procedure returns a value, 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, on can use `supportsStoredProcedures()` method on `DatabaseMetaData` object to verify if the underlying database supports the stored procedure. It is supports, the description of the stored procedures can be obtained using `getProcedures()` on. `DatabaseMetaData` object. Consider the following procedure has been 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;
```

The above 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, its old password and a new password.

If you are using MySQL command prompt to create procedure, you may face a problem. Note that stored procedure 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 following command;

```
DELIMITER //
```

At the end, change the delimiter to ; if you want.

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.

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();
```

The above 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 represents in JDBC by the `ResultSet` interface. The `ResultSet` objects are usually generated by executing SQL statements that queries the database. A pointer points to a particular row of ResultSet object at a time. This pointer is called *cursor*. The cursor is positioned before the first row when the ResultSet object if generated. To retrieve data from a row of the `ResultSet`, the cursor must be positioned at the row. The ResultSet Interface provides methods to move this cursor.

next()

- This method on `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

Because `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 by the cursor. These have the form `getXxx()`, where `Xxx` is the name of data type of the column. For example, the data types is 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. Column index start from 1. 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 to the capabilities of DataBase Management System (DBMS) before dealing with it. This because different DBMSs often provide different features,

implement them differently, and also use different data types. Moreover, driver may also implement additional features on the top of 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 `getMetaData()` method on `Connection` object as follows:

```
DatabaseMetaData md = con.getMetaData();
```

We can then use various methods on this `DatabaseMetaData` object to collect required information about the DBMS. Following a list of commonly used methods:

`getDatabaseMetaData()`

Returns `DatabaseMetaData` object that contains detail information about the underlying database. Some important methods of `DatabaseMetaData` are

`String getSQLKeywords()`

Returns keywords available

`getDatabaseProductName()`

Returns name of the manufacturer

`getDatabaseProductVersion()`

Returns current version

`getDriverName()`

Returns driver used

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 the above page is shown in the table 21.8.

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 cursor can be moved backward and forward. Additionally, a result set can be *updatable* such that any change to the result set reflects the database immediately. A result set can be scrollable as well as updatable. *Note that scrollable and updatable results set incur significant overhead. So, such result sets should be created if underlying application really performs scrolling.*

In addition to the *scrollability* and *updatability*, another important concept is defined called *sensitivity*. Sensitivity broadly answers to the following question:

```
Can a result set see the changes that are made to the underlying database?
```

If a result can't see any changes, it is said to be insensitive. Otherwise, sensitivity of a result set is defined with respect to the database operation as well as operating party. For example a result set is said to be sensitive to update if it can see any update operation made on underlying database. The sensitivity rules are shown in the table 21.5.

The `createStatement()` and `prepareStatement()` methods take extra parameters that specify the type of result returned by subsequent execution of SQL statements. The prototype of `createStatement()` method to generate 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 following static integer constants defined in `ResultSet`. Their meaning is described below:

- `TYPE_FORWARD_ONLY`
  If this constant is used, 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

Table 21.5: Visibility of Internal and External Changes to Scrollable Result Set for 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 following static integer constants defined in `ResultSet`. They are briefly describes below:

- `CONCUR_READ_ONLY`
  The result set won't be updatable

- `CONCUR_UPDATABLE`
  Rows can be added and deleted and columns can be updated and is visible by others

21.22.3 Examples

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 only be navigated in one direction (forward) and starting at only one position (the first row). In JDBC 2.0 row pointer can be manipulated as if it were an array index. Some of the important methods available on scrollable `ResultSet` object are shown in the table 21.6. Following examples demonstrate how to navigate a scrollable result set:

- Forward cursor one row
```
rs.next();
//or
rs.relative(1);
```

- Backward cursor one row
```
rs.previous();
//or
rs.relative(-1);
```

- Set cursor before the first row
```
rs.beforeFirst();
```

- Set cursor after the last row
```
rs.afterLast();
```

- Set cursor at the first row (row 1)
```
rs.first();
//or
rs.absolute(1);
```

- Set cursor at the last row
```
rs.last();
//or
rs.absolute(-1);
```

- Set cursor at the second row
```
rs.absolute(2);
```

- Set cursor at the second last row
```
rs.absolute(-2);
```

- Forward cursor 6 rows from the current position.
```
rs.relative(6);
//Sets the cursor after the last row, if it goes beyond the last row
```

- Backward cursor 4 rows from the current position.
```
rs.relative(-4);
//Sets the cursor before the first row, if it goes before the first row
```

Table 21.6: 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 to just before the first row
afterLast()	Sets cursor to 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().

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 following steps are used:

- Obtain an updatable result set
- Move cursor to the row to be updated using positioning methods available on `ResultSet` object
- Update the value of one or more columns that row using `updateXxx()` method on `ResultSet` object where xxx is the data type of the column.
- Finally, update the database table using `updateRow()` method.

Following example demonstrate how to update, insert or delete a row from a database table using 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()
```

Following JSP page changes the password of a specified user using 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); }
%>
```

Following example populates the table questions by inserting questions into 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); }
%>
```

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 to the result set. Before retrieving data from a row, you should invoke `refreshRow()` method of `ResultSet` object so that it contains the updated row. The following JSP page shows how to use 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 simple 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 of 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 `getMetaData()` method on `ResultSet` object as follows:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM questions");
ResultSetMetaData rsmd = rs.getMetaData();
```

It can used to get some useful information such as number of rows, number of columns, column name, their type etc. Following some commonly used methods on `ResultSetMetaData` object are:

`int getColumnCount()`
: Returns number of columns in the result

`String getColumnName(int)`
: Returns name of a column in a ResultSet. Requires an integer argument indicating the position of the column within the result set

`int getColumnType(int)`
: Returns type of the specified column in a form of `java.sql.Types`

`String getColumnTypeName(int)`
: Returns type of the specified column as string

`String getColumnClassName(int)`
: Returns the fully qualified Java type name of the specified column

`int getPrecision(int)`
: Returns number of decimal positions

`int getScale(int)`
: Returns 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 the table 21.7.

Table 21.7: 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 21.8: 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
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
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
storesUpperCaseIdentifiers	false
storesUpperCaseQuotedIdentifiers	true
supportsANSI92EntryLevelSQL	true
supportsANSI92FullSQL	false
supportsANSI92IntermediateSQL	false
supportsAlterTableWithAddColumn	true
supportsAlterTableWithDropColumn	true
supportsBatchUpdates	true
supportsCatalogsInDataManipulation	true
supportsCatalogsInIndexDefinitions	true
supportsCatalogsInPrivilegeDefinitions	true
supportsCatalogsInProcedureCalls	True
supportsCatalogsInTableDefinitions	True
supportsColumnAliasing	True

supportsConvert	False
supportsCoreSQLGrammar	True
supportsCorrelatedSubqueries	True
supportsDataDefinitionAndDataManipulationTransactions	False
supportsDataManipulationTransactionsOnly	False
supportsDifferentTableCorrelationNames	True
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
supportsStoredProcedures	true
supportsSubqueriesInComparisons	true
supportsSubqueriesInExists	true
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

21.24 Key Words

JSP Engine—A specialized servlet that supervises and controls the execution of JSP pages

Web container— JSP pages run under the a web server called web container.

Translation—The procedure of converting JSP page to its servlet source code

Directives—Instructions used in the JSP page to inform JSP engine how this page is interpreted and executed.

Scriptlets— Sections in a JSP page where we can insert arbitrary piece of Java code>

Expression—It is a faster, easier and clearer way to display values of variables/parameters/expressions in a JSP page

Actions— JSP actions are XML tags that can be used in a JSP page to use functions provided by the JSP engine.

Template text—The static HTML/XML components in a JSP page

Declarations— JSP declarations are used to declare one or more variables, methods or inner classes that can be used later in the JSP page

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.

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.

Plug-in— The <jsp:plugin> action is used to generate HTML file that can download Java plug-in on demand and execute applets or JavaBeans.

Bean— JavaBeans are reusable Java components which allow us to separate business logic from presentation logic.

Tag extension— It allows us to define and use custom tags in an JSP page exactly like other HTML/XML tags using Java code.

Session tracking—A procedure using which JSP pages can relate a sequence of requests

Cookie—A name/value pair

JDBC— A Java framework that allows us to access databases through Java programs.

DDL Statement—SQL statements typically used to create tables

DML Statement—SQL statements typically used to manipulate tables such as insert, update etc.

DCL Statement—SQL statements typically used to control database tables

DQL Statement—SQL statements typically used to read values from tables

Atomic Transaction—A transaction that either does not occur or occurs completely without any interleaving

PreparedStatement—Precompiled statements used to fire parameterized queries

Stored Procedure—Subroutine written using SQL

CallableStatement—Statements that are used to call stored procedure in a database

ResultSet—The result of a DQL statement

Scrollable Result Set—A result set whose pointer (cursor) can be moved back and forth

Updatable Result Set—A result set which can be used directly to modify original database tables

Database Metadata—An object that represents the meta information about a database

Result Set Metadata—An object that represents the meta information about a result set.

21.25 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 which 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 the servlet technology. The JSP pages are finally converted to the servlets automatically.

A JSP page basically consists of static HTML/XML components called template text as well as JSP constructs. JSP constructs consists 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 piece of data in a JSP page without using out.print() or out.write() statements.

You can insert 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 or 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.

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 an 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 CallableStatement object. A table of data is represents in JDBC by the ResultSet interface. Result sets can be scrollable in the sense that cursor can be moved backward and forward. The DatabaseMetaData interface provides methods to collect comprehensive information about a DBMS.

21.26 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

21.27 Exercises

21.27.1 Objective Type Questions

1. What is the full form of JSP ?
Java Servlet Pages
Java Server Pages
Java Small Pages
Java Special Pages

2. Which of the following statements is true?
JSP and servlets are completely different technologies.
Servlets are built on JSP technology and all servlets are ultimately converted to JSP pages

Servlet is a client side technology whereas JSP is server-side technology
JSPs are built on servlet technology and all JSPs are ultimately translated to servlets

3. What is the advantage of using RequestDispatcher to forward a request to another resource than using a sendRedirect() ?

The RequestDispatcher does not use the reflection API.
sendRedirect() is no longer available in the current servlet API.
The RequestDispatcher does not require a round trip to the client, and thus is more efficient and allows the server to maintain request state.
sendRedirect() is not a cross-web serve mechanism.

4. Which one of the following cases the scriptlet is better suited?

Code that handles cookies.
Code that deals with logic that relates to database access.
Code that deals with session management.
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 JSP page.
A session manager.
A servlet.
A JavaBean.

6. Which of the following can be included using JSP include action?

Servlet
Another JSP
Plain text file
All of the above

7. Which of the following is used to read parameters from a JSP page?

<jsp:getParam/> action
<jsp:param> action
request.getParameter() method
<jsp:readParam/> action

8. Which of the following is not a method of JSP's servlet ?

_jspService()
jspDestroy()
jspService()
jspInit()

9. Which of the following JSP action is used to include a file in another file?

<jsp:import>
<jsp:include>
<jsp:read>
<jsp:get>

10. What is a JSP page translated into ?

CGI
Servlet

Applet
JavaBean

11. Which of the following is not true regarding cookie?
Cookie class has a two argument constructor
The response object is used to add a cookie
The request object is used to get the cookie
The request object is used for creating cookies

12. Which of the following scopes the implicit object exception has?
page
request
session
application

13. Which of the following tags is used to override JSP file's initialization method?
<%= %>
<%@ %>
<% %>
<%! %>

14. Which of the following scopes is not valid with respect to JavaBean in JSP ?
request
response
session
application

15. Which of the following statement is true regarding HttpServletResponse.sendRedirect()?
Server itself redirects the current request
sendRedirect() executes on the client
Server sends a redirection instruction to the client
Server drops the current request

16. Which of the following tags for scriptlets?
<%= %>
<%@ %>
<% %>
<%! %>

17. Which of the following attributes of page directive is used to indicate that the current page is an error page?
errorPage
isErrorPage
anErrorPage
pageError

18. Which of the following tags for expressions?
<%= %>
<%@ %>
<% %>

<%! %>

19. Which packages contain the JDBC classes?
java.db.sql and javax.db.sql
java.jdbc and javax.jdbc
java.db and javax.db
java.sql and javax.sql

20. Which type of the following drivers converts JDBC calls into the network protocol to communicate with the database management system directly?
Type 1 driver
Type 2 driver
Type 3 driver
Type 4 driver

21. Which of the following type of objects is used to execute parameterized queries?
ParameterizedStatement
Statement
PreparedStatement
All of the above

22. Which of the following method on Statement object is used to execute DML statements?
executeInsert()
execute()
executeQuery()
executeDML()

23. Which type of driver types makes a JDBC-ODBC bridge?
Type 1 driver
Type 2 driver
Type 3 driver
Type 4 driver

24. What is the meaning of ResultSet.TYPE_SCROLL_INSENSITIVE
This means that the ResultSet is insensitive to scrolling
This means that the ResultSet is sensitive to scrolling, but insensitive to changes made by others
This means that the ResultSet is insensitive to scrolling and insensitive to changes made by others
This means that the ResultSet is sensitive to scrolling, but insensitive to updates, i.e. not updateable

25. Which of the following SQL keyword is used to read data from a database table?
SELECT
CHOOSE
READ
EXTRACT

26. Which of the following statement objects is used to call a stored procedure in JDBC?
Statement
PreparedStatement

CallableStatement
ProcedureStatement

27. Which of the following objects is used to obtain DatabaseMetaData object?
Driver
DriverManager
Connection
ResultSet

28. Which of the following methods is used to call a stored procedure in the database?
execute()
executeProcedure()
call()
run()

29. What will be the effect if we call deleteRow() method on a ResultSet object?
The row pointed by cursor is deleted from the ResultSet, but not from the database.
The row pointed by cursor is deleted from the ResultSet and from the database
The row pointed by cursor is deleted from the database, but not from the ResultSet.
None of the above

30. If you want to work with a ResultSet, which of these methods will not work on PreparedStatement?
execute()
executeQuery()
executeUpdate()
All of the above

31. Which of the following characters is used as a place holder in CallableStatement?
$
@
?
#

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"?
rs.getString(0)
rs.getString("login")
rs.getString(1)
All of the above

33. Which of the following interfaces is used to control transactions?
Statement
Connection
ResultSet
DatabaseMetaData

34. Which one of the following represents a correct order for
INSERT, INTO, SELECT, FROM, WHERE
SELECT, FROM, WHERE, INSERT, INTO

INTO, INSERT, VALUES, FROM, WHERE
INSERT, INTO, WHERE, AND, VALUES

35. Which of the following is NOT a benefit of using JDBC?
JDBC programs are tightly integrated with the server operating system.
Systems built with JDBC are relatively easy to move to different platforms.
JDBC programs can be written to connect with a wide variety of databases.
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?
database layer.
client program layer.
both client program and database layers.
JDBC layer.

37. Which database application model would an enterprise-wide solution most likely adopt?
The monolithic model
The two-tier model
The three-tier model
The n-tier model

38. Which code segment could execute the stored procedure "calculate()" located in a database server?

```
Statement stmt = connection.createStatement();
        stmt.execute("calculate()");
CallableStatement cs = con.prepareCall("{call calculate}");
        cs.executeQuery();
PrepareStatement pstmt = connection.prepareStatement("calculate()");
        pstmt.execute();
Statement stmt = connection.createStatement();
        stmt.executeStoredProcedure("calculate()");
```

21.27.2 Subjective Type Questions

1. What is the key difference between HttpServletResponse.sendRedirect() and <jsp:forward>?
2. What is a benefit of using JavaBeans to separate business logic from presentation markup within the JSP environment ?
3. Write the differences between Type-2 and Type-3 driver
4. How do you to get the ResultSet of stored procedure ?
5. What is the purpose of setAutoCommit( ) method on Connection object?
6. How to move the cursor in scrollable resultsets?
7. Describe the procedure we use to retrieve data from the ResultSet ?
8. What are the three statements in JDBC & differences between them ?
9. How do you update a ResultSet programmatically?
10. Why do we use PreparedStatement instead of Statement?
11. What is stored procedure. How do you create stored procedure ?
12. How do you insert and delete a row programmatically?
13. What are batch updates? What are their usefulness?
14. What are four types of JDBC driver?

15. How can you use PreparedStatement ?