CSE 686 Internet Programming

Servlets & JSP

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

Servlets

Servlets

- Servlets were conceived as a Java-based alternative to CGI scripting.
  - **Common Gateway Interface** (**CGI**) is a standard method for web servers to delegate dynamic content generation programs/scripts, commonly known as **CGI scripts** or simply **CGIs.** They are usually written in a scripting language, such as Perl or PHP.
- A servlet was intended to be an instance of a (Java) class that could handle HTTP ‘GET’ and ‘POST’ requests.
  - A web server that supported servlets should be similar to Apache combined with mod-Perl or mod-PHP.

The Servlet Container

- The servlet **container** (system) is an object-oriented framework application.
  - Sun's programmers wrote the first servlet container, which has evolved into **Apache Tomcat.**
  - The 'main' program they wrote instantiates various Sun-defined classes to create the objects that form the 'servlet container'.
  - Afterwards, control is then passed to one of these objects, which reads the **deployment files** for all the entries in the associated **webapps** directory and builds **tables** identifying the servlets that it can run.
  - The container then waits to handle incoming HTTP **get** and **post** requests.

The Servlet Container, cont.

- When the first HTTP **get** or **post** request is received for a particular servlet, the container identifies the class needed for the request, creates an instance of that class, and initializes it.
  - Information about the class and initialization data are obtained from the **tables** built earlier.
  - The container then creates **input and output streams** so that the new servlet can read request data and write response data.
  - Then the **service** method of the servlet is invoked to handle the request.
  - When the servlet finishes handling its client's request, the input and output streams are closed by the container.
  - The servlet itself is normally kept around, waiting for the next request, but the container can destroy servlet objects.

The Servlet Container & Threads

- When your server starts to get busy, you will have many concurrent get and post requests.
  - The servlet container has no problems with this, as it is **multi-threaded -** each client is associated with a separate thread and separate input and output streams.
  - The container handles thread management issues; it may create and destroy threads, or it may use a pool of reusable threads.
  - But you only have one servlet object.
  - Servlets should be ‘thread-safe’ – which means that generally neither instance data members nor static class members should be present.

Where Do You Come In?

- Sun's programmers have written all the code for managing:
  - Servlets (the servlet framework)
  - Connections
  - I/O
  - Threads
- So where do you come in?
  - You have to define a concrete servlet class that implements behaviors that are left abstract in the servlet classes defined in Sun's framework.
- Sun's framework includes the definitions of a servlet class hierarchy: (See next slide)

The Servlet Interface

- **Servlet** (interface)
  - This defines essential functionality: init, destroy, get-config-info, **service**.
- **GenericServlet** (abstract)
  - This class adds logging, parameters, context, and other features.
- **HttpServlet** (abstract)
  - Still an abstract class, the HttpServlet class has an effective **service** method.
    - This method uses data defining the HTTP request, and dispatches the request to the appropriate ‘get’, ‘put’, ‘post’ or ‘delete’ method.

javax.servlet

Interface Servlet

All Known Subinterfaces:

HttpJspPage, JspPage

All Known Implementing Classes:

FacesServlet, GenericServlet, HttpServlet

`public interface` **`Servlet`**

Defines methods that all servlets must implement.

A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

To implement this interface, you can write a generic servlet that extends `javax.servlet.GenericServlet` or an HTTP servlet that extends `javax.servlet.http.HttpServlet`.

This interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:

1. The servlet is constructed, then initialized with the `init` method.
2. Any calls from clients to the `service` method are handled.
3. The servlet is taken out of service, then destroyed with the `destroy` method, then garbage collected and finalized.

In addition to the life-cycle methods, this interface provides the `getServletConfig` method, which the servlet can use to get any startup information, and the `getServletInfo` method, which allows the servlet to return basic information about itself, such as author, version, and copyright.

Author:

Various

See Also:

`GenericServlet`, `HttpServlet`

javax.servlet

Class GenericServlet

java.lang.Object
    javax.servlet.GenericServlet

All Implemented Interfaces:

Serializable, Servlet, ServletConfig

Direct Known Subclasses:

HttpServlet

```
public abstract class GenericServlet
extends Object
implements Servlet, ServletConfig, Serializable
```

Defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, extend `HttpServlet` instead.

`GenericServlet` implements the `Servlet` and `ServletConfig` interfaces. `GenericServlet` may be directly extended by a servlet, although it's more common to extend a protocol-specific subclass such as `HttpServlet`.

`GenericServlet` makes writing servlets easier. It provides simple versions of the lifecycle methods `init` and `destroy` and of the methods in the `ServletConfig` interface. `GenericServlet` also implements the `log` method, declared in the `ServletContext` interface.

To write a generic servlet, you need only override the abstract `service` method.

Author:

Various

See Also:

Serialized Form

javax.servlet.http

Class HttpServlet

java.lang.Object
  javax.servlet.GenericServlet
    javax.servlet.http.HttpServlet

All Implemented Interfaces:

Serializable, Servlet, ServletConfig

```
public abstract class HttpServlet
extends GenericServlet
```

Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. A subclass of `HttpServlet` must override at least one method, usually one of these:

- `doGet`, if the servlet supports HTTP GET requests
- `doPost`, for HTTP POST requests
- `doPut`, for HTTP PUT requests
- `doDelete`, for HTTP DELETE requests
- `init` and `destroy`, to manage resources that are held for the life of the servlet
- `getServletInfo`, which the servlet uses to provide information about itself

There's almost no reason to override the `service` method. `service` handles standard HTTP requests by dispatching them to the handler methods for each HTTP request type (the `do`*XXX* methods listed above).

Likewise, there's almost no reason to override the `doOptions` and `doTrace` methods.

Servlets typically run on multithreaded servers, so be aware that a servlet must handle concurrent requests and be careful to synchronize access to shared resources. Shared resources include in-memory data such as instance or class variables and external objects such as files, database connections, and network connections. See the Java Tutorial on Multithreaded Programming for more information on handling multiple threads in a Java program.

Author:

Various

See Also:

Servlets

- Your own servlet class should extend the **HttpServlet** abstract class and provide an effective implementation of at least one of **doGet**, **doPost** or the other action methods required by the **HttpServlet.service** method.
  - Only one of these methods is required.
  - Sun's framework code will generate an error response if your servlet is invoked using an HTTP request that you do not support.

doGet

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
              throws ServletException,
                     java.io.IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a GET request.

Overriding this method to support a GET request also automatically supports an HTTP HEAD request. A HEAD request is a GET request that returns no body in the response, only the request header fields.

When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

Where possible, set the Content-Length header (with the `ServletResponse.setContentLength(int)` method), to allow the servlet container to use a persistent connection to return its response to the client, improving performance. The content length is automatically set if the entire response fits inside the response buffer.

When using HTTP 1.1 chunked encoding (which means that the response has a Transfer-Encoding header), do not set the Content-Length header.

The GET method should be safe, that is, without any side effects for which users are held responsible. For example, most form queries have no side effects. If a client request is intended to change stored data, the request should use some other HTTP method.

doPost

```
protected void doPost(HttpServletRequest req,
                      HttpServletResponse resp)
               throws ServletException,
                      java.io.IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a POST request. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers.

When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

Where possible, set the Content-Length header (with the `ServletResponse.setContentLength(int)` method), to allow the servlet container to use a persistent connection to return its response to the client, improving performance. The content length is automatically set if the entire response fits inside the response buffer.

When using HTTP 1.1 chunked encoding (which means that the response has a Transfer-Encoding header), do not set the Content-Length header.

This method does not need to be either safe or idempotent. Operations requested through POST can have side effects for which the user can be held accountable, for example, updating stored data or buying items online.

If the HTTP POST request is incorrectly formatted, `doPost` returns an HTTP "Bad Request" message.

Servlet.service

service

```
void service(ServletRequest req,
             ServletResponse res)
             throws ServletException,
                    java.io.IOException
```

Called by the servlet container to allow the servlet to respond to a request.

This method is only called after the servlet's `init()` method has completed successfully.

The status code of the response always should be set for a servlet that throws or sends an error.

Servlets typically run inside multithreaded servlet containers that can handle multiple requests concurrently. Developers must be aware to synchronize access to any shared resources such as files, network connections, and as well as the servlet's class and instance variables. More information on multithreaded programming in Java is available in the Java tutorial on multi-threaded programming.

Parameters:

`req` - the `ServletRequest` object that contains the client's request

`res` - the `ServletResponse` object that contains the servlet's response

Throws:

`ServletException` - if an exception occurs that interferes with the servlet's normal operation

`java.io.IOException` - if an input or output exception occurs

---

getServletInfo

Requests & Responses

- Sun's framework uses 'request' and 'response' wrapper classes (**HTTPServletRequest** and **HTTPServletResponse**) and for the HTTP input and output streams.
  - Instances of these classes are created by the framework code and passed to the **HttpServlet.service** method, and then to the doGet or doPost method that the servlet programmer defines.

- Typically, the response from a servlet is a dynamically generated HTML page.

javax.servlet.http

Interface HttpServletRequest

All Superinterfaces:

ServletRequest

All Known Implementing Classes:

HttpServletRequestWrapper

```
public interface HttpServletRequest
extends ServletRequest
```

Extends the `ServletRequest` interface to provide request information for HTTP servlets.

The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

Author:

Various

Field Summary

javax.servlet.http

Interface HttpServletResponse

All Superinterfaces:

ServletResponse

All Known Implementing Classes:

HttpServletResponseWrapper

```
public interface HttpServletResponse
extends ServletResponse
```

Extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies.

The servlet container creates an `HttpServletResponse` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

Author:

Various

See Also:

Servlets & Forms

- **Form data** can be split into name/value pairs.
  - But all this work is standard, independent of any application.
  - So this functionality is built into Sun's HttpServletRequest class.
  - In most cases, servlet programmers simply use **HttpServletRequest**.**getParameter** operations (inherited from **ServletRequest**) to pick up values entered in forms.

getParameter

`java.lang.String` **`getParameter`**`(java.lang.String name)`

Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

You should only use this method when you are sure the parameter has only one value. If the parameter might have more than one value, use `getParameterValues(java.lang.String)`.

If you use this method with a multivalued parameter, the value returned is equal to the first value in the array returned by `getParameterValues`.

If the parameter data was sent in the request body, such as occurs with an HTTP POST request, then reading the body directly via `getInputStream()` or `getReader()` can interfere with the execution of this method.

Parameters:

`name` - a `String` specifying the name of the parameter

Returns:

a `String` representing the single value of the parameter

See Also:

`getParameterValues(java.lang.String)`

---

getParameterNames

`java.util.Enumeration<java.lang.String>` **`getParameterNames`**`()`

The First Servlet Example

- The first example illustrates a simple servlet that handles data entry from a form.
  - The form uses an HTTP **get** request
    - It has only one data input field.
  - The servlet has a **doGet** method.
    - This method reads and processes the input from the form.
    - The input is supposed to be a positive number
    - The servlet returns the square root of that number.

Form and Servlet Code

- The form is defined as a static HTML page:

```
<html><head><title>Form Data Entry Page</title></head>
<body>
<h1 align=center >Fill in some data</h1>
<p>The demo servlet works out square roots, so feed it a number.
<p>
<form method=get action="/demo/sqrtservlet">
<input type=text name=number>
<input type=submit>
</form>
</body></html>
```

Form and Servlet Code

- The servlet class, **SqrtServlet.java**, is defined as an extension of Sun's **HttpServlet** class:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SqrtServlet extends HttpServlet {
  public void doGet (HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException
  {
    ...
  }
}
```

```
public void doGet (HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
{
  response.setContentType("text.html)
  // Pick up PrintWriter output stream for response
  PrintWriter out = response.getWriter();
  // Generate standard HTML header etc
  out.println("<html>" +
    "<head><title> Square roots </title></head>" );
  out.println("<body >" );

// Pick up name=value bindings for data from form, only
// the one parameter here
String data = request.getParameter("number");
double value = 0;
  // Does string represent a number?
  try { value = Double.parseDouble(data); }
  catch(Exception e) {
    // Error response, non-numeric input
    out.println("<p>Need NUMERIC data.");
    out.println("</body></html>");
    return;
  }
  // Must be non-negative
  if(value<0.0) {
    out.println("You are living in an imaginary world");
    out.println("</body></html>");
    return;
  }
  // Data are valid, process and respond
  value = Math.sqrt(value);
  out.println("Your square root value is " + value);
  out.println("</body></html>");
}
```

Form and Servlet Code

- The **doGet** function must extract the string entered in the form's 'number' input field and then try to convert this string to a numeric value.
  - If the string does not represent a number, or if it corresponds to a negative number, the servlet should generate a response page with an error message.
  - If the input data are valid, then they are processed and a results page is generated.

```
public void doGet (HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
{
  response.setContentType("text.html)
  // Pick up PrintWriter output stream for response
  PrintWriter out = response.getWriter();
  // Generate standard HTML header etc
  out.println("<html>" +
    "<head><title> Square roots </title></head>" );
  out.println("<body >" );

// Pick up name=value bindings for data from form, only
// the one parameter here
String data = request.getParameter("number");
double value = 0;
  // Does string represent a number?
  try { value = Double.parseDouble(data); }
  catch(Exception e) {
    // Error response, non-numeric input
    out.println("<p>Need NUMERIC data.");
    out.println("</body></html>");
    return;
  }
  // Must be non-negative
  if(value<0.0) {
    out.println("You are living in an imaginary world");
    out.println("</body></html>");
    return;
  }
  // Data are valid, process and respond
  value = Math.sqrt(value);
  out.println("Your square root value is " + value);
  out.println("</body></html>");
}
```

Form and Servlet Code

- The class **HttpServletRequest** supports a number of functions for getting parameter values with form data.

- The example code uses the basic **getParameter**(*name*) function that returns the value (as a String) for the parameter (or null if the parameter is not defined).

```
public void doGet (HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
{
  response.setContentType("text.html)
  // Pick up PrintWriter output stream for response
  PrintWriter out = response.getWriter();
  // Generate standard HTML header etc
  out.println("<html>" +
    "<head><title> Square roots </title></head>" );
  out.println("<body >" );

// Pick up name=value bindings for data from form, only
// the one parameter here
String data = request.getParameter("number");
double value = 0;
  // Does string represent a number?
  try { value = Double.parseDouble(data); }
  catch(Exception e) {
    // Error response, non-numeric input
    out.println("<p>Need NUMERIC data.");
    out.println("</body></html>");
    return;
  }
  // Must be non-negative
  if(value<0.0) {
    out.println("You are living in an imaginary world");
    out.println("</body></html>");
    return;
  }
  // Data are valid, process and respond
  value = Math.sqrt(value);
  out.println("Your square root value is " + value);
  out.println("</body></html>");
}
```

getParameter

`java.lang.String` **`getParameter`**`(java.lang.String name)`

Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

You should only use this method when you are sure the parameter has only one value. If the parameter might have more than one value, use `getParameterValues(java.lang.String)`.

If you use this method with a multivalued parameter, the value returned is equal to the first value in the array returned by `getParameterValues`.

If the parameter data was sent in the request body, such as occurs with an HTTP POST request, then reading the body directly via `getInputStream()` or `getReader()` can interfere with the execution of this method.

Parameters:

`name` - a `String` specifying the name of the parameter

Returns:

a `String` representing the single value of the parameter

See Also:

`getParameterValues(java.lang.String)`

---

getParameterNames

`java.util.Enumeration<java.lang.String>` **`getParameterNames`**`()`

Form and Servlet Code

- The response object can be asked to return a reference to an associated **PrintWriter**, as was done in this code, where PrintWriter out was set to allow responses to be written.

- Response information can be simply written to this stream.

```
public void doGet (HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
{
  response.setContentType("text/html");
  // Pick up PrintWriter output stream for response
  PrintWriter out = response.getWriter();
  // Generate standard HTML header etc
  out.println("<html>" +
    "<head><title> Square roots </title></head>" );
  out.println("<body >" );

// Pick up name=value bindings for data from form, only
// the one parameter here
String data = request.getParameter("number");
double value = 0;
  // Does string represent a number?
  try { value = Double.parseDouble(data); }
  catch(Exception e) {
    // Error response, non-numeric input
    out.println("<p>Need NUMERIC data.");
    out.println("</body></html>");
    return;
  }
  // Must be non-negative
  if(value<0.0) {
    out.println("You are living in an imaginary world");
    out.println("</body></html>");
    return;
  }
  // Data are valid, process and respond
  value = Math.sqrt(value);
  out.println("Your square root value is " + value);
  out.println("</body></html>");
}
```

getWriter

```
java.io.PrintWriter getWriter()
                            throws java.io.IOException
```

Returns a `PrintWriter` object that can send character text to the client. The `PrintWriter` uses the character encoding returned by `getCharacterEncoding()`. If the response's character encoding has not been specified as described in `getCharacterEncoding` (i.e., the method just returns the default value `ISO-8859-1`), `getWriter` updates it to `ISO-8859-1`.

Calling flush() on the `PrintWriter` commits the response.

Either this method or `getOutputStream()` may be called to write the body, not both.

Returns:

a `PrintWriter` object that can return character data to the client

Throws:

`UnsupportedEncodingException` - if the character encoding returned by `getCharacterEncoding` cannot be used

`IllegalStateException` - if the `getOutputStream` method has already been called for this response object

`java.io.IOException` - if an input or output exception occurred

See Also:

`getOutputStream()`, `setCharacterEncoding(java.lang.String)`

---

setCharacterEncoding

```
void setCharacterEncoding(java.lang.String charset)
```

java.io

Class PrintWriter

java.lang.Object
    java.io.Writer
        java.io.PrintWriter

All Implemented Interfaces:

Closeable, Flushable, Appendable, AutoCloseable

```
public class PrintWriter
extends Writer
```

Prints formatted representations of objects to a text-output stream. This class implements all of the `print` methods found in `PrintStream`. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println`, `printf`, or `format` methods is invoked, rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking `checkError()`.

Since:

JDK1.1

Tomcat

- Tomcat is a **Servlet container** (Web server that interacts with **Servlets**) developed under the Jakarta Project of Apache Software Foundation
  - Tomcat implements the Servlet and the Java Server Pages (JSP) specifications of Sun Microsystems
  - Tomcat is an open-source, non commercial project
    - Licensed under the Apache Software License
  - Tomcat is written in Java (OS independent)

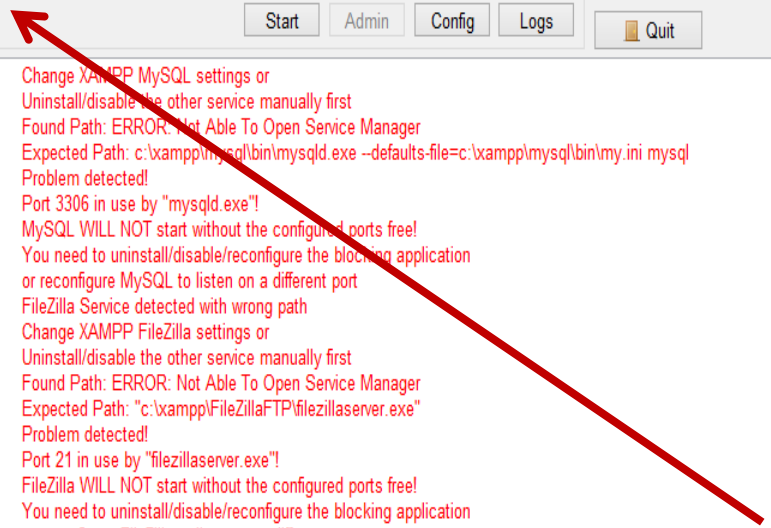
XAMPP Control Panel v3.1.0 3.1.0 [Compiled: September 20th 2012]

XAMPP Control Panel v3.1.0 3.1.0

Modules

Service	Module	PID(s)	Port(s)	Actions			
✗	Apache	1928	80, 443	Stop	Admin	Config	Logs
✓	MySQL	2232	3306	Stop	Admin	Config	Logs
✓	FileZilla	1596	21, 14147	Stop	Admin	Config	Logs
	Mercury			Start	Admin	Config	Logs
✗	Tomcat			Start	Admin	Config	Logs

Config
Netstat
Shell
Explorer
Services
Help
Quit



```
8:01:29 AM [mysql]  Change XAMPP MySQL settings or
8:01:29 AM [mysql]  Uninstall/disable the other service manually first
8:01:29 AM [mysql]  Found Path: ERROR: Not Able To Open Service Manager
8:01:29 AM [mysql]  Expected Path: c:\xampp\mysql\bin\mysqld.exe --defaults-file=c:\xampp\mysql\bin\my.ini mysql
8:01:29 AM [mysql]  Problem detected!
8:01:29 AM [mysql]  Port 3306 in use by "mysqld.exe"!
8:01:29 AM [mysql]  MySQL WILL NOT start without the configured ports free!
8:01:29 AM [mysql]  You need to uninstall/disable/reconfigure the blocking application
8:01:29 AM [mysql]  or reconfigure MySQL to listen on a different port
8:01:29 AM [filezilla]  FileZilla Service detected with wrong path
8:01:29 AM [filezilla]  Change XAMPP FileZilla settings or
8:01:29 AM [filezilla]  Uninstall/disable the other service manually first
8:01:29 AM [filezilla]  Found Path: ERROR: Not Able To Open Service Manager
8:01:29 AM [filezilla]  Expected Path: "c:\xampp\FileZillaFTP\filezillaserver.exe"
8:01:29 AM [filezilla]  Problem detected!
8:01:29 AM [filezilla]  Port 21 in use by "filezillaserver.exe"!
8:01:29 AM [filezilla]  FileZilla WILL NOT start without the configured ports free!
8:01:29 AM [filezilla]  You need to uninstall/disable/reconfigure the blocking application
8:01:29 AM [filezilla]  or reconfigure FileZilla to listen on a different port
8:01:29 AM [filezilla]  Problem detected!
8:01:29 AM [filezilla]  Port 14147 in use by "filezillaserver.exe"!
8:01:29 AM [filezilla]  FileZilla WILL NOT start without the configured ports free!
8:01:29 AM [filezilla]  You need to uninstall/disable/reconfigure the blocking application
8:01:29 AM [filezilla]  or reconfigure FileZilla to listen on a different port
8:01:29 AM [main]  Starting Check-Timer
8:01:29 AM [main]  Control Panel Ready
```

Apache Tomcat

The Apache Software Foundation
http://www.apache.org/

Administration

Status
Tomcat Manager

Documentation

Release Notes
Change Log
Tomcat Documentation

Tomcat Online

Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.html
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

NOTE: For security reasons, using the manager webapp is restricted to users with role "manager". Users are defined in `$CATALINA_HOME/conf/tomcat-users.xml`.

Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.

Tomcat mailing lists are available at the Tomcat project web site:

- **users@tomcat.apache.org** for general questions related to configuring and using Tomcat

Tomcat Directory Structure

- Take a quick look at the Tomcat installed directory. It contains the following sub-directories:
  - **bin**: contains the binaries; and startup, shutdown and other scripts.
  - **conf**: contains the system-wide configuration files, such as server.xml, web.xml, context.xml, and tomcat-users.xml.
  - **lib**: contains the Tomcat's system-wide JAR files, accessible by all webapps. You could also place external JAR file (such as MySQL JDBC Driver) here.
  - **logs**: contains Tomcat's log files. You may need to check for error messages here.
  - **webapps**: contains the webapps to be deployed. You can also place the WAR file for deployment here.
  - **work**: Tomcat's working directory used by JSP, for JSP-to-Servlet conversion.
  - **temp**: Temporary files.

Tomcat Directory Structure

- The main directory associated with a web application, **/webapps/demo** in this example, must contain a **WEB-INF** subdirectory.
- This WEB-INF subdirectory holds:
  - **web.xml** This file contains a specification of deployment parameters for the web application.
  - **lib** This optional subdirectory contains any special libraries that may be needed. These libraries will be Java archive files (.jar files). Later examples will illustrate cases where extra libraries are needed. This simple examples do not require extra libraries.
  - **classes** This subdirectory should contain the .class files for the servlet(s) and helper classes that are defined for a specific web application. The .java source files can be included but are not required. (During development, you would normally include the .java source files.)

Tomcat Directory Structure

- The Java code shown earlier should be created as:

  /webapps/demo/WEB-INF/classes/SqrtServlet.java

  - The javax.servlet classes are not part of the standard Java development libraries, so naive attempts to compile the SqrtServlet.java code will fail.
  - The required javax class definitions are available as the **servlet-api.jar** file included in the lib directory of the Tomcat installation.
  - This file must be added to your CLASSPATH prior to an attempt at compiling the servlet code.

web.xml

- Finally, you must create the file /webapps/demo/WEB-INF/**web.xml**.
- This should contain the following deployment description:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
<servlet>
<servlet-name>myservlet</servlet-name>
<servlet-class>SqrtServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>myservlet</servlet-name>
<url-pattern>/sqrtservlet</url-pattern>
</servlet-mapping>
</web-app>
```

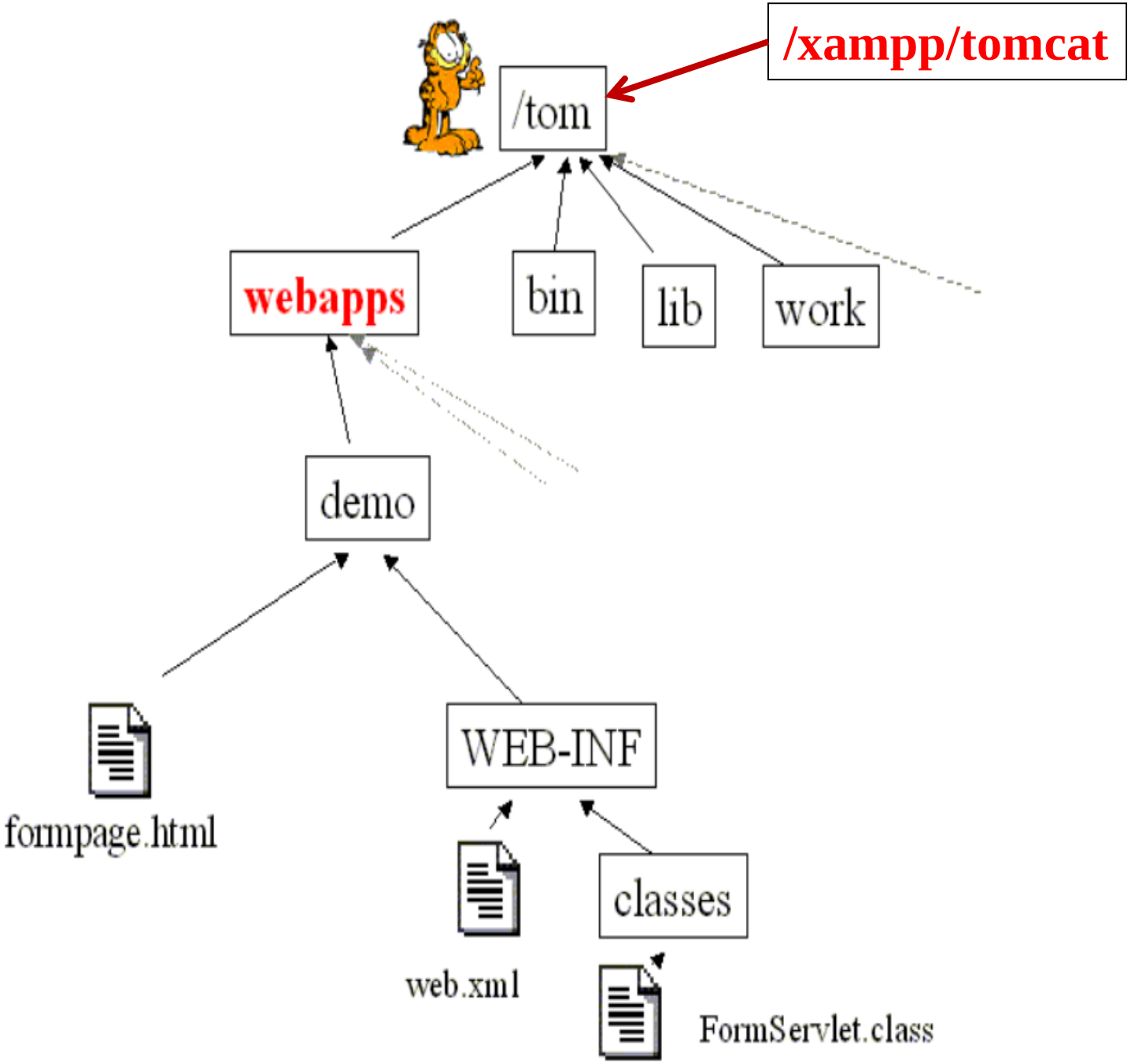
web.xml

- The first line (<?xml ... ?>) simply identifies this as an XML document.
- The ‘document type’ on the following lines provides a reference to the specification for Web Application documents.
- The application specific data are between the <web-app> start tag and the </web-app> end tag.
  - The first component, the data in the <servlet> ... </servlet> tag, relate a **servlet-name** to the implementation class.
  - The **servlet-name** is used by the container to identify the servlet
  - The **servlet-class** identifies the Java class that this servlet instantiates.
  - The **servlet-mapping** data relate the servlet’s registered name to the (partial) URL(s) that will be used to access it.

web.xml

- After creating all the directories and files, and after having successfully compiled the servlet code, you should restart your Tomcat.
  - Your new web-based ‘square-root service’ should be available at:

    http://localhost:8080/demo/FormPage.html


/xampp/tomcat
/tom
webapps
bin
lib
work
demo
WEB-INF
frompage.html
web.xml
classes
FormServlet.class

Servlet Examples with Code

This is a collection of examples which demonstrate some of the more frequently used parts of the Servlet API. Familiarity with the Java(tm) Programming Language is assumed.

These examples will only work when viewed via an http URL. They will not work if you are viewing these pages via a "file://..." URL. Please refer to the *README* file provide with this Tomcat release regarding how to configure and start the provided web server.

Wherever you see a form, enter some data and see how the servlet reacts. When playing with the Cookie and Session Examples, jump back to the Headers Example to see exactly what your browser is sending the server.

To navigate your way through the examples, the following icons will help:

 Execute the example

 Look at the source code for the example

 Return to this screen

Tip: To see the cookie interactions with your browser, try turning on the "notify when setting a cookie" option in your browser preferences. This will let you see when a session is created and give some feedback when looking at the cookie demo.

Hello World	Execute	Source
Request Info	Execute	Source
Request Headers	Execute	Source
Request Parameters	Execute	Source
Cookies	Execute	Source

JSP

JSP

- JSP (Java Server Pages) is an alternate way of creating servlets
  - JSP is written as ordinary HTML, with a little Java mixed in
  - The Java is enclosed in special tags, such as **<% ... %>**
  - The HTML is known as the **template text**
- JSP files must have the extension **.jsp**
  - JSP is **translated** into a Java servlet, which is then **compiled**
  - Servlets are run in the usual way
  - The browser or other client sees only the resultant HTML, as usual
- Tomcat knows how to handle both servlets and JSP pages

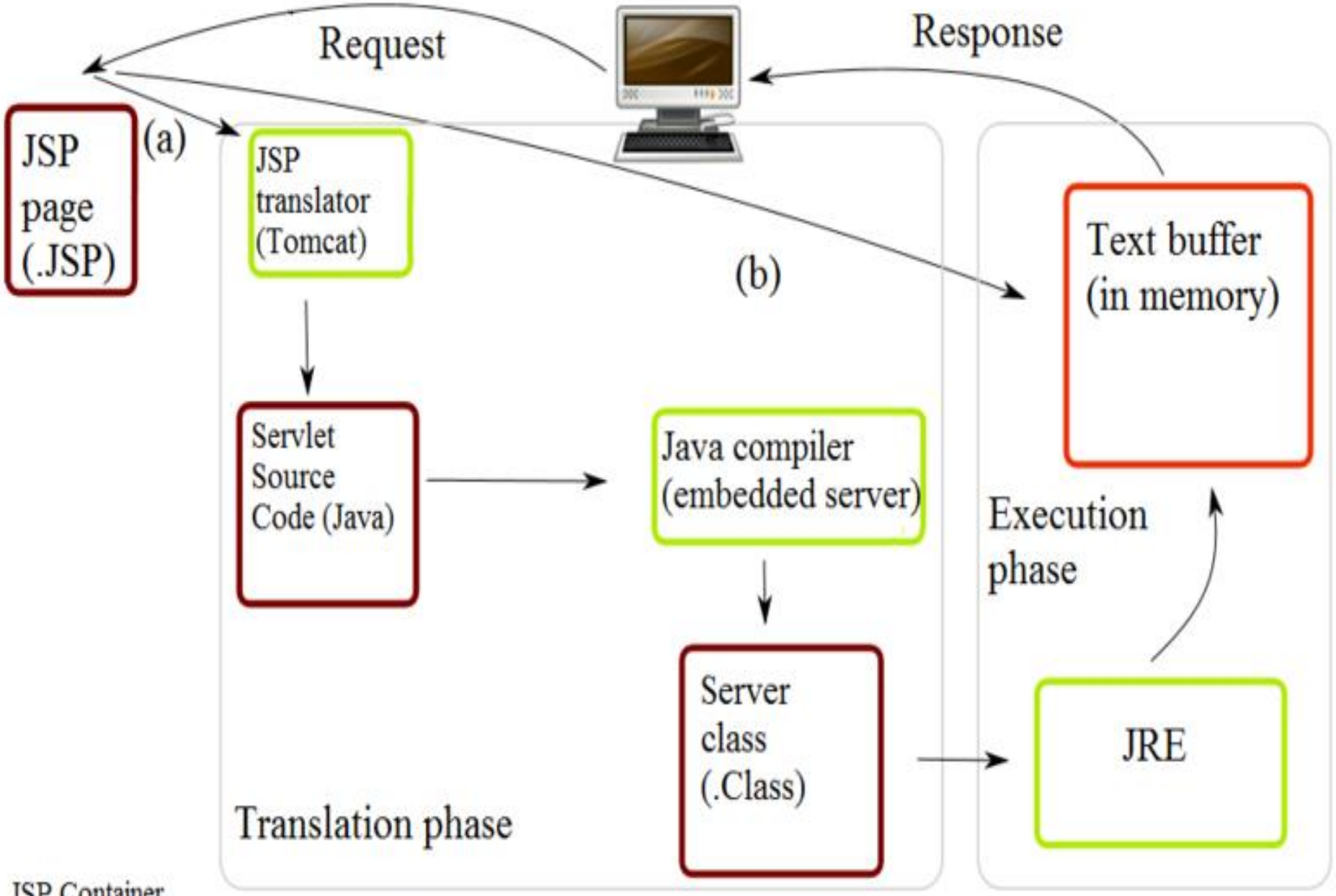
<% and %>

- There are a pair of tags for enclosing your JSP Code:

  <% %>

- An example:

```
<html>
  <body>
    <%
      out.println("Hello World, this is my first JSP page.");
    %>
  </body>
</html>
```



JSP Container

(a) Translation occurs at this point, if JSP has been changed or is new.

(b) If not, translation is skipped.

JSP

- Since JSPs are really servlets, they must run in a servlet container.
  - Tomcat is again the most appropriate container.
  - The deployment of JSPs on Tomcat is similar to the deployment of servlets.
  - A complete JSP web application will normally comprise some JSPs, static HTML pages, and possibly supporting servlet classes.
  - Such an application should be installed in the webapps directory of your Tomcat server.
    - This directory would hold the JSPs and static HTML pages
    - It would normally have a WEB-INF subdirectory containing a web.xml deployment file, libraries and a classes subdirectory for any supporting servlets and bean classes (servlets generated from the JSPs are created in subdirectories of the tomcat/work directory).

Elements in a JSP

- These are:
  - Directives
  - Action elements
  - Scripting elements
  - Template text

JSP Directives

- Directive
  - Message to JSP container
    - i.e., program that compiles/executes JSPs
  - Enable programmers to specify
    - Page settings
    - Content to include from other resources
    - Custom tag libraries used in the JSP

Directives

Directive	Description
page	Defines page settings for the JSP container to process.
include	Causes the JSP container to perform a translation-time insertion of another resource's content. As the JSP is translated into a servlet and compiled, the referenced file replaces the **include** directive and is translated as if it were originally part of the JSP.
taglib	Allows programmers to include their own new tags in the form of *tag libraries*. These libraries can be used to encapsulate functionality and simplify the coding of a JSP.

page Directives

- Directives affect the servlet class itself

- A directive has the form:

  *<%@ directive attribute="**value**" %>*

  or

  *<%@ directive attribute1="**value1**"*
  *attribute2="**value2**"*
  ...
  *attributeN="**valueN**" %>*

- The most useful directive is page, which lets you import packages
  - Example: <%@ page import="java.util.*" %>

page Directives: An Example

```
<html>
  <%@ page import="java.text.*" %>
  <%@ page import="java.util.*" %>
  <head>
    <title>Datet JSP</title>
  </head>
  <body>
    <h1>
      Today is
      <%
          DateFormat df = DateFormat.getDateInstance(DateFormat.FULL);
          Date today = new Date();
          String msg = df.format(today);
          out.println(msg);
      %>
    </h1>
    <h1>Have a nice day!</hl>
  </body
</html>
```

java.text

Class DateFormat

java.lang.Object
    java.text.Format
        java.text.DateFormat

All Implemented Interfaces:

Serializable, Cloneable

Direct Known Subclasses:

SimpleDateFormat

```
public abstract class DateFormat
extends Format
```

DateFormat is an abstract class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner. The date/time formatting subclass, such as SimpleDateFormat, allows for formatting (i.e., date -> text), parsing (text -> date), and normalization. The date is represented as a Date object or as the milliseconds since January 1, 1970, 00:00:00 GMT.

DateFormat provides many class methods for obtaining default date/time formatters based on the default or a given locale and a number of formatting styles. The formatting styles include FULL, LONG, MEDIUM, and SHORT. More detail and examples of using these styles are provided in the method descriptions.

DateFormat helps you to format and parse dates for any locale. Your code can be completely independent of the locale conventions for months, days of the week, or even the calendar format: lunar vs. solar.

FULL

```
public static final int FULL
```

Constant for full style pattern.

See Also:

Constant Field Values

LONG

```
public static final int LONG
```

Constant for long style pattern.

See Also:

Constant Field Values

MEDIUM

```
public static final int MEDIUM
```

Constant for medium style pattern.

See Also:

Constant Field Values

format

```
public final String format(Date date)
```

Formats a Date into a date/time string.

Parameters:

`date` - the time value to be formatted into a time string.

Returns:

the formatted time string.

parse

```
public Date parse(String source)
          throws ParseException
```

Parses text from the beginning of the given string to produce a date. The method may not use the entire text of the given string.

See the `parse(String, ParsePosition)` method for more information on date parsing.

Parameters:

`source` - A `String` whose beginning should be parsed.

Returns:

A `Date` parsed from the string.

Throws:

`ParseException` - if the beginning of the specified string cannot be parsed

java.util

Class Date

java.lang.Object
    java.util.Date

All Implemented Interfaces:

Serializable, Cloneable, Comparable<Date>

Direct Known Subclasses:

Date, Time, Timestamp

```
public class Date
extends Object
implements Serializable, Cloneable, Comparable<Date>
```

The class `Date` represents a specific instant in time, with millisecond precision.

Prior to JDK 1.1, the class `Date` had two additional functions. It allowed the interpretation of dates as year, month, day, hour, minute, and second values. It also allowed the formatting and parsing of date strings. Unfortunately, the API for these functions was not amenable to internationalization. As of JDK 1.1, the `Calendar` class should be used to convert between dates and time fields and the `DateFormat` class should be used to format and parse date strings. The corresponding methods in `Date` are deprecated.

Although the `Date` class is intended to reflect coordinated universal time (UTC), it may not do so exactly, depending on the host environment of the Java Virtual Machine. Nearly all modern operating systems assume that 1 day = 24 × 60 × 60 = 86400 seconds in all cases. In UTC, however, about once every year or two there is an extra

The include Directive

- The include directive inserts another file into the file being parsed
  - The included file is treated as just more JSP, hence it can include static HTML, scripting elements, actions, and directives
- Syntax: <%@ include file="***URL*** " %>
  - The ***URL*** is treated as relative to the JSP page
  - If the ***URL*** begins with a slash, it is treated as relative to the home directory of the Web server

The include Directive

- JSP **include** directive
  - Includes content of another resource at JSP translation time
    - Not as flexible as **<jsp:include>** action

include Directives: An Example

- You can include a file into your JSP page. The syntax is like that:

<jsp:include page="querybook.html" /> or:
<%@ include file="querybook.html" %>

```
<html>
  <body>
    <%
      out.println("Welcome to my online bookstore:");
    %>
    <jsp:include page="querybook.html" />
  </body>
</html>
```

Implicit Objects

- Implicit Objects
  - Provide access to many servlet capabilities within a JSP
  - Four scopes
    - Application scope
      - Objects owned by the container application
      - Any servlet or JSP can manipulate these objects
    - Page scope
      - Objects that exist only in page in which they are defined
      - Each page has its own instance of these objects
    - Request scope
      - Objects exist for duration of client request
      - Objects go out of scope when response sent to client
    - Session scope
      - Objects exist for duration of client's browsing session
      - Objects go out of scope when client terminates session or when session timeout occurs

Implicit Objects

Implicit Object	Description
Application Scope	
application	This **javax.servlet.ServletContext** object represents the container in which the JSP executes.
Page Scope	
config	This **javax.servlet.ServletConfig** object represents the JSP configuration options. As with servlets, configuration options can be specified in a Web application descriptor.
exception	This **java.lang.Throwable** object represents the exception that is passed to the JSP error page. This object is available only in a JSP error page.
out	This **javax.servlet.jsp.JspWriter** object writes text as part of the response to a request. This object is used implicitly with JSP expressions and actions that insert string content in a response.
page	This **java.lang.Object** object represents the **this** reference for the current JSP instance.
pageContext	This **javax.servlet.jsp.PageContext** object hides the implementation details of the underlying servlet and JSP container and provides JSP programmers with access to the implicit objects discussed in this table.

javax.servlet.jsp

Class JspWriter

```
java.lang.Object
  └java.io.Writer
      └javax.servlet.jsp.JspWriter
```

All Implemented Interfaces:

java.io.Closeable, java.io.Flushable, java.lang.Appendable

Direct Known Subclasses:

BodyContent

```
public abstract class JspWriter
extends java.io.Writer
```

The actions and template data in a JSP page is written using the JspWriter object that is referenced by the implicit variable out which is initialized automatically using methods in the PageContext object.

This abstract class emulates some of the functionality found in the java.io.BufferedWriter and java.io.PrintWriter classes, however it differs in that it throws java.io.IOException from the print methods while PrintWriter does not.

Implicit Objects

Implicit Object	Description
response	This object represents the response to the client. The object normally is an instance of a class that implements **HttpServlet-Response** (package **javax.servlet.http**). If a protocol other than HTTP is used, this object is an instance of a class that implements **javax.servlet.ServletResponse**.
Request Scope	
request	This object represents the client request. The object normally is an instance of a class that implements **HttpServletRequest** (package **javax.servlet.http**). If a protocol other than HTTP is used, this object is an instance of a subclass of **javax.servlet.ServletRequest**.
Session Scope	
session	This **javax.servlet.http.HttpSession** object represents the client session information if such a session has been created. This object is available only in pages that participate in a session.

Implicit Objects

- Example:

```
<%= request.getRemoteHost() %>
```

Action Elements

- Actions are XML-syntax tags used to control the servlet engine

<jsp:include page="***URL*** " />

- Inserts the indicated relative ***URL*** at execution time (not at compile time, like the include directive does)
- This is great for rapidly changing data

<jsp:forward page="***URL***" />
<jsp:forward page="<%= ***JavaExpression*** %>" />

- Jump to the (static) ***URL*** or the (dynamically computed) ***JavaExpression*** resulting in a URL

Action Elements

- There is a standard set of action elements (the **jsp tag library**).
  - These jsp actions include common operations on beans (use, get property, set property), some operations for communication with other JSPs or servlets (forwarding a request, including some output from another servlet), and a few miscellaneous things like help for setting up an applet that is to appear in a response page.
  - You are not limited to using only the standard JSP tags.
  - There are other tag libraries available from Apache, and also from O'Reilly publishers.
  - You can even define your own tag libraries and employ your own action tags.

Standard Actions

Action	Description
<jsp:include>	Dynamically includes another resource in a JSP. As the JSP executes, the referenced resource is included and processed.
<jsp:forward>	Forwards request processing to another JSP, servlet or static page. This action terminates the current JSP's execution.
<jsp:plugin>	Allows a plug-in component to be added to a page in the form of a browser-specific **object** or **embed** HTML element. In the case of a Java applet, this action enables the downloading and installation of the *Java Plug-in*, if it is not already installed on the client computer.
<jsp:param>	Used with the **include**, **forward** and **plugin** actions to specify additional name/value pairs of information for use by these actions.

Standard Actions

Action	Description
JavaBean Manipulation	
<jsp:useBean>	Specifies that the JSP uses a JavaBean instance. This action specifies the scope of the bean and assigns it an ID that scripting components can use to manipulate the bean.
<jsp:setProperty>	Sets a property in the specified JavaBean instance. A special feature of this action is automatic matching of request parameters to bean properties of the same name.
<jsp:getProperty>	Gets a property in the specified JavaBean instance and converts the result to a string for output in the response.

include Directives: An Example

- You can include a file into your JSP page. The syntax is like that:

<jsp:include page=“querybook.html" /> or:
<%@ include file="querybook.html" %>

```
<html>
  <body>
    <%
      out.println("This is my first JSP page.");
    %>
    <jsp:include page="querybook.html" />
  </body>
</html>
```

JSP Scripting Elements

- **Scripting elements** include:
  - <%= ***expression*** %>
    - The ***expression*** is evaluated and the result is inserted into the HTML page
  - <% ***code*** %>
    - The ***code*** is inserted into the servlet's **service** method
    - This construction is called a **scriptlet**
  - <%! ***declarations*** %>
    - The ***declarations*** are inserted into the servlet class, not into a method

Expressions: An Example

```
<html>
  <head>
    <title>Input JSP</title>
  </head>
  <body>
    <form action="InputJSP.jsp" method=post>
      Enter some text: 
      <input type="text" name="Text">
      <br><br>
      <input type="submit" value="Submit">
    </form><br>
    <h3>You entered: 
    <%= request.getParameter("Text") %></hl3>
  </body>
</html>
```

DateJSP.jsp

```
<html>
  <%@ page import="java.text.*" %>
  <%@ page import="java.util.*" %>
  <head>
    <title>Datet JSP</title>
  </head>
  <body>
    <h1>
      Today is
      <%
          DateFormat df = DateFormat.getDateInstance(DateFormat.FULL);
          Date today = new Date();
          String msg = df.format(today);
          out.println(msg);
      %>
    </h1>
    <h1>Have a nice day!</hl>
  </body
</html>
```

LoopyJSP.jsp

```
<html>
  <head>
    <title>Can't you see I'm trying to work here?</title>
  </head>
  <body>
    <% for (int i = 0; i < 30; i++) { %>
    All work and no play makes Jack a dull boy.<br>
    <% } %>
  </body
</html>
```

Declarations

- Use <%! ... %> for declarations to be added to your servlet class, not to any particular method
  - Caution: Servlets are multithreaded, so nonlocal variables must be handled with extreme care
  - If declared with <% ... %>, variables are local and OK
  - Data can also safely be put in the request or session objects

- Example:

```
<%! private int accessCount = 0; %>
    Accesses to page since server reboot:
<%= ++accessCount %>
```

- You can use <%! ... %> to declare methods as easily as to declare variables

CounterJSP.jsp

```
<html>
  <%@ page import="java.text.*" %>
  <%@ page import="java.util.*" %>
  <head>
    <title>Counter JSP</title>
  </head>
  <body>
    <h1>
      This JSP has been displayed <%= count++ %> time(s)!
  </body
</html>
<%!
private static int count = 1;
%>
```

DateJSP.jsp Revisted

```
<html>
  <%@ page import="java.text.*" %>
  <%@ page import="java.util.*" %>
  <head>
    <title>Date JSP</title>
  </head>
  <body>
    <h1>Today is <%= getDate() %></hl>
    <h1>Have a nice day!</hl>
</body
</html>
<%!
private String getDate() {
  DateFormat df = DateFormat.getDateInstance(DateFormat.FULL);
  Date today = new Date();
  return df.format(today);
}
%>
```

The ‘Guru’ – A JSP Example

- This example is a kind of ‘fortune cookie’ program - clients can request advice from the ‘Guru’ who will reply with a randomly chosen statement of advice, or a comment on the ways of the world.
  - The Guru system is comprised of a JSP, Advice.jsp, and a helper Guru class.
  - The JSP uses scriptlet style code to create a Guru and ask its advice.

The ‘Guru’ – A JSP Example

- The example relies on default parameters applying to JSPs, and so hides quite a number of details that will have to be exposed in later more realistic examples.

- The example should be constructed in a jspdemo subdirectory of your tomcat/webapps directory. This directory will be used to hold the Advice.jsp file, and a WEB-INF subdirectory.

- The WEB-INF subdirectory has a classes subdirectory; the Guru.java file is installed in this classes subdirectory.

- This simple example does not require a web.xml deployment file.

The scriptlet Guru

- The Advice.jsp file contains the following:

```
<%@ page import="mystuff.*" %>
<html><head><title>The Guru</title></head>
  <body bgcolor=white>
    <h1 align=center><font color=red>
    Today's advice from the Guru
    </font></h1>
    <% Guru theGuru = new Guru(); %>
    <p>
    <%= theGuru.enlightenMe() %>
  </body>
</html>
```

The scriptlet Guru

- The two highlighted lines are embedded scriptlet code.
- The first creates an instance of the Guru class; the second invokes the enlightenMe method of the new Guru object.
- The first scriptlet results in code in the generated servlet but nothing in the final resulting HTML page.
- The second scriptlet (actually, an '**expression**'), within the <%= ... %> tags, is to return a string – the resulting string is embedded in the final HTML page.

Guru.java

- The Guru class has the definition:

```
import java.util.Random;
public class Guru {
    private static Random rr;
    private static final String[ ] Sayings = {
        "Women like silent men. They think they're listening.",
        "When a man brings his wife flowers for no reason, there's a reason.",
        ...
        "Rich bachelors should be heavily taxed. It is not fair that some men should be happier than others."
    };
    public String enlightenMe() {
        int select = rr.nextInt(Sayings.length);
        return Sayings[select];
    }
    {
        rr = new Random();
    }
}
```

The Final Example: ListMovies.jsp

```
<%@ page import="movie.*" %>
<%@ page import="java.util.*" %>

<html>
  <head>
    <title>List Movies</title>
  </head>
  <body>
    <h1>DVD Movies On Sale Now</h1>
    <h3>
      <%= getMovieList() %>
    </h3>
  </body>
</html>
```

The Final Example: ListMovies.jsp, Pt.2

```
<%!
private String getMovieList() {
  String msg = "";
  ArrayList<Movie> movies = MovieIO.getMovies();
  for (Movie m : movies) {
    msg += m.year + ": ";
    msg += m.title + ", $";
    msg += m.price + "<br>";
  }
  return msg;
}
%>
```

The Final Example: Movie.java

```
public class Movie {
  public String title;
  public int year;
  public double price;

  public Movie(String title, int year, double price) {
    this.title = title;
    this.year = year;
    this.price = price;
  }
}
```

The Final Example: MovieIO.java

```
package movie;
import java.io.*;
import java.util.*;

public class MovieIO {
  public static ArrayList<Movie> getMovies() {
    ArrayList<Movie> movies = new ArrayList<Movie>();
    BufferedReader in = getReader("C:\\xampp\\tomcat\\webapps\\jspdemo\\movies.txt");
    Movie movie = readMovie(in);

    while (movie != null) {
      movies.add(movie);
      movie = readMovie(in);
    }
    return movies;
  }
```

The Final Example: MovieIO.java, pt.2

```
private static BufferedReader getReader(String name) {
    BufferedReader in = null;
    try {
      File file = new File(name);
      in = new BufferedReader(new FileReader(file));
    }
    catch (FileNotFoundException e) {
      System.out.println("The file doesn't exist.");
      System.exit(0);
    }
    return in;
  }
```

The Final Example: MovieIO.java, pt.3

```
private static Movie readMovie(BufferedReader in) {
    String title;
    int year;
    double price;
    String line = "";
    String[] data;
    try {
      line = in.readLine();
    }
    catch (IOException e) {
      System.out.println("I/O Error.");
      System.exit(0);
    }
    if (line == null) return null;
    else {
      data = line.split("\t");
      title = data[0];
      year = Integer.parseInt(data[1]);
      price = Double.parseDouble(data[2]);
      return (new Movie(title, year, price));
    }
  }
}
```

Source Code for HelloWorld Example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

javax.servlet.http

Class HttpServlet

```
java.lang.Object
  └javax.servlet.GenericServlet
      └javax.servlet.http.HttpServlet
```

All Implemented Interfaces:

java.io.Serializable, Servlet, ServletConfig

```
public abstract class HttpServlet
extends GenericServlet
implements java.io.Serializable
```

Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. A subclass of `HttpServlet` must override at least one method, usually one of these:

- `doGet`, if the servlet supports HTTP GET requests
- `doPost`, for HTTP POST requests
- `doPut`, for HTTP PUT requests
- `doDelete`, for HTTP DELETE requests
- `init` and `destroy`, to manage resources that are held for the life of the servlet

doGet

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
              throws ServletException,
                     java.io.IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a GET request.

Overriding this method to support a GET request also automatically supports an HTTP HEAD request. A HEAD request is a GET request that returns no body in the response, only the request header fields.

When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

Where possible, set the Content-Length header (with the `ServletResponse.setContentLength(int)` method), to allow the servlet container to use a persistent connection to return its response to the client, improving performance. The content length is automatically set if the entire response fits inside the response buffer.

When using HTTP 1.1 chunked encoding (which means that the response has a Transfer-Encoding header), do not set the Content-Length header.

The GET method should be safe, that is, without any side effects for which users are held responsible. For example, most form queries have no side effects. If a client request is intended to change stored data, the request should use some other HTTP method.

javax.servlet.http

Interface HttpServletRequest

All Superinterfaces:

ServletRequest

All Known Implementing Classes:

HttpServletRequestWrapper

```
public interface HttpServletRequest
extends ServletRequest
```

Extends the `ServletRequest` interface to provide request information for HTTP servlets.

The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

Author:

Various

Field Summary

javax.servlet.http

Interface HttpServletResponse

All Superinterfaces:

ServletResponse

All Known Implementing Classes:

HttpServletResponseWrapper

```
public interface HttpServletResponse
extends ServletResponse
```

Extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies.

The servlet container creates an `HttpServletResponse` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

Author:

Various

See Also:

HttpServletResponse

getWriter

```
java.io.PrintWriter getWriter()
                            throws java.io.IOException
```

Returns a `PrintWriter` object that can send character text to the client. The `PrintWriter` uses the character encoding returned by `getCharacterEncoding()`. If the response's character encoding has not been specified as described in `getCharacterEncoding` (i.e., the method just returns the default value `ISO-8859-1`), `getWriter` updates it to `ISO-8859-1`.

Calling flush() on the `PrintWriter` commits the response.

Either this method or `getOutputStream()` may be called to write the body, not both.

Returns:

a `PrintWriter` object that can return character data to the client

Throws:

`UnsupportedEncodingException` - if the character encoding returned by `getCharacterEncoding` cannot be used

`IllegalStateException` - if the `getOutputStream` method has already been called for this response object

`java.io.IOException` - if an input or output exception occurred

See Also:

`getOutputStream()`, `setCharacterEncoding(java.lang.String)`

---

setCharacterEncoding

```
void setCharacterEncoding(java.lang.String charset)
```

Source Code for Request Info Example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestInfo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Request Information Example</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h3>Request Information Example</h3>");
        out.println("Method: " + request.getMethod());
        out.println("Request URI: " + request.getRequestURI());
        out.println("Protocol: " + request.getProtocol());
        out.println("PathInfo: " + request.getPathInfo());
        out.println("Remote Address: " + request.getRemoteAddr());
        out.println("</body>");
        out.println("</html>");
    }

    /**
     * We are going to perform the same operations for POST requests
     * as for GET methods, so this method just sends the request to
     * the doGet method.
     */

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        doGet(request, response);
    }
}
```

Source Code for RequestHeader Example

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestHeaderExample extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration e = request.getHeaderNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = request.getHeader(name);
            out.println(name + " = " + value);
        }
    }
}
```

Source Code for Request Parameter Example

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestParamExample extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Request Parameters Example</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h3>Request Parameters Example</h3>");
        out.println("Parameters in this request:<br>");
        if (firstName != null || lastName != null) {
            out.println("First Name:");
            out.println(" = " + HTMLFilter.filter(firstName) + "<br>");
            out.println("Last Name:");
            out.println(" = " + HTMLFilter.filter(lastName));
        } else {
            out.println("No Parameters, Please enter some");
        }
        out.println("<P>");
        out.print("<form action=\"");
        out.print("RequestParamExample\" ");
        out.println("method=POST>");
        out.println("First Name:");
        out.println("<input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("Last Name:");
        out.println("<input type=text size=20 name=lastname>");
        out.println("<br>");
        out.println("<input type=submit>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse res)
    throws IOException, ServletException
    {
        doGet(request, response);
    }
}
```

Source Code for Cookie Example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CookieExample extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // print out cookies

        Cookie[] cookies = request.getCookies();
        for (int i = 0; i < cookies.length; i++) {
            Cookie c = cookies[i];
            String name = c.getName();
            String value = c.getValue();
            out.println(name + " = " + value);
        }

        // set a cookie

        String name = request.getParameter("cookieName");
        if (name != null && name.length() > 0) {
            String value = request.getParameter("cookieValue");
            Cookie c = new Cookie(name, value);
            response.addCookie(c);
        }
    }
}
```

javax.servlet.http

Class Cookie

```
java.lang.Object
  └javax.servlet.http.Cookie
```

All Implemented Interfaces:

java.io.Serializable, java.lang.Cloneable

```
public class Cookie
extends java.lang.Object
implements java.lang.Cloneable, java.io.Serializable
```

Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number. Some Web browsers have bugs in how they handle the optional attributes, so use them sparingly to improve the interoperability of your servlets.

The servlet sends cookies to the browser by using the `HttpServletResponse#addCookie` method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.

The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the `HttpServletRequest#getCookies` method. Several cookies might have the same name but different path attributes.

Cookies affect the caching of the Web pages that use them. HTTP 1.0 does not cache pages that use cookies created with this class. This class does not support the

The Cookie Class

- The javax.servlet.http package defines a Cookie class.
- Cookies are created with a name/value pair.
- The value string assigned to a cookie should not contain any characters like newlines, ampersands, question marks, whitespace etc.
- If you really need such characters in your cookies, then encode with java.net.URLEncoder before setting the value, and decode with java.net.URLDecoder after extracting a value.

The Cookie Class

- The Cookie class has a number of accessor and mutator methods:
  - String getName()
    - Returns the name of a cookie.
  - String getValue(); void setValue(String newValue)
    - Return the current and assign a new value to a cookie.
  - String getDomain(); void setDomain(String pattern)
    - Returns/sets the domain name specified for a cookie.
  - String getPath(); void setPath(String uri)
    - Returns/sets the path associated with a cookie (so that it is only returned to a subset of the servlets on the server).
  - int getMaxAge(); void setMaxAge(int expiry)
    - Returns/sets **the maximum age in seconds** associated with a cookie (by default, cookies are created as 'transient' with a maximum age of –1; these cookies are automatically deleted from the client when the browser terminates).

The Cookie Class

- The javax.servlet.http.HttServletRequest class has a

  Cookie[] getCookies

  method to retrieve any returned cookies.

- The HttpServletResponse class has addCookie and setCookie methods to put cookies in the header portion of a response.

getCookies

`Cookie[] getCookies()`

Returns an array containing all of the `Cookie` objects the client sent with this request. This method returns `null` if no cookies were sent.

Returns:

an array of all the `Cookies` included with this request, or `null` if the request has no cookies

---

getDateHeader

`long getDateHeader(java.lang.String name)`

Returns the value of the specified request header as a `long` value that represents a `Date` object. Use this method with headers that contain dates, such as `If-Modified-Since`.

The date is returned as the number of milliseconds since January 1, 1970 GMT. The header name is case insensitive.

If the request did not have a header of the specified name, this method returns -1. If the header can't be converted to a date, the method throws an `IllegalArgumentException`.

Parameters:

`name` - a `String` specifying the name of the header

Returns:

a `long` value representing the date specified in the header expressed as the number of milliseconds since January 1, 1970 GMT, or -1 if the named header was not included with the request

Throws:

`IllegalArgumentException` - If the header value can't be converted to a date

The Cookie Class

- You can create your own session management code based directly on cookies.
- The code would be analogous to that for PHP scripting.
- All the servlets in an application would have to incorporate the same code fragment.
- This code would check ‘get’ and ‘post’ requests for a returned session identification cookie
- If a cookie was not found, the code would generate a suitably randomized session identifier, and add it as a cookie to the response header.
- The session identifier would serve as a primary key for database records.

Source Code for Session Example

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionExample extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession(true);

        // print session info

        Date created = new Date(session.getCreationTime());
        Date accessed = new Date(session.getLastAccessedTime());
        out.println("ID " + session.getId());
        out.println("Created: " + created);
        out.println("Last Accessed: " + accessed);

        // set session info if needed

        String dataName = request.getParameter("dataName");
        if (dataName != null && dataName.length() > 0) {
            String dataValue = request.getParameter("dataValue");
            session.setAttribute(dataName, dataValue);
        }

        // print session contents

        Enumeration e = session.getAttributeNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = session.getAttribute(name).toString();
            out.println(name + " = " + value);
        }
    }
}
```

javax.servlet.http

Interface HttpSession

public interface **HttpSession**

Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session in many ways such as using cookies or rewriting URLs.

This interface allows servlets to

- View and manipulate information about a session, such as the session identifier, creation time, and last accessed time
- Bind objects to sessions, allowing user information to persist across multiple user connections

When an application stores an object in or removes an object from a session, the session checks whether the object implements `HttpSessionBindingListener`. If it does, the servlet notifies the object that it has been bound to or unbound from the session. Notifications are sent after the binding methods complete. For session that are invalidated or expire, notifications are sent after the session has been invalidated or expired.

When container migrates a session between VMs in a distributed container setting, all session attributes implementing the

Sessions

- The HttpServletRequest class has the session-related methods:
  - HttpSession getSession()
    - Returns a session for this request, creating one if none exists.
  - HttpSession getSession(boolean create)
    - If there is a session object associated with the request, it is returned. Otherwise, depending on the value of the create argument, either a session object is created or a null is returned.
  - public boolean isRequestedSessionIdFromCookie()
  - public boolean isRequested SessionIdFromURL()
- The last 2 functions let the servlet programmer determine whether cookies or rewritten URLs are being used (rare for a servlet programmer to really need to know this).

getSession

`HttpSession` **`getSession`**`()`

Returns the current session associated with this request, or if the request does not have a session, creates one.

Returns:

the `HttpSession` associated with this request

See Also:

`getSession(boolean)`

---

isRequestedSessionIdValid

`boolean` **`isRequestedSessionIdValid`**`()`

Checks whether the requested session ID is still valid.

If the client did not specify any session ID, this method returns `false`.

Returns:

`true` if this request has an id for a valid session in the current session context; `false` otherwise

See Also:

`getRequestedSessionId()`, `getSession(boolean)`, `HttpSessionContext`

---

isRequestedSessionIdFromCookie

Servlets-Related Classes

- The servlet code (as implemented in the **GenericServlet** and **HttpServlet** classes) defines the basic life cycle methods along with methods that can be used to obtain information about the servlet, the overall web application (context) and the container.
- The life cycle of a servlet involves creation, initialization, multiple uses via its service method, and final destruction.
- The container creates a servlet when the first request is received for that servlet.
- After instantiating a new servlet, the container calls its **init** method.
- The **GenericServlet** class defines an empty, no operation init method. You will often want to override this and provide an effective initialization function.

Servlets-Related Classes

- For example, it is sometimes appropriate for a (lightly used) servlet to own a connection to a database.
- This connection would be opened in the init function, closed in the destroy function, and be used (subject to a **mutex** lock) in some function called from the servlet's doGet or doPost action functions.
- You might want to have the name and password for the database provided as parameters (rather than have them encoded as strings in the Java source files).
- Initialization parameters can be part of a web.xml file:

Servlets-Related Classes

```
<servlet>
    <servlet-name>ServletA</servlet-name>
    <servlet-class>ExampleServlet</servlet-class>
    <init-param>
        <param-name>DBUser</para-name>
        <param-value>HSimpson</param-name>
    </init-param>
    <init-param>
        <param-name>DBPassword</para-name>
        <param-value>Duh</param-name>
    </init-param>
    <load-on-startup/>
</servlet> file:
```

Servlets-Related Classes

- An initialization function that creates a database connection using the specified name and password obtained from initialization data would be something like the following:

Servlets-Related Classes

```
...
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ExampleServlet extends HttpServlet {
    // Some constants
    private static final String dbDriverName = "...";
    private static final String dbURL = "...";
    // The database connection - implicitly shared by
    // concurrent clients (so use own locks!)

    private Connection theDBConnection;
```

Servlets-Related Classes

```
public void init() throws ServletException{
    // Code to pick up initialization data
    String userName = getInitParameter("DBUser");
    String userPassword = getInitParameter("DBPassword");
    try {
        theDBConnection = DriverManager.getConnection(dbURL, userName, userPassword);
    }
    catch(Exception e) {
        // Maybe print some trace statements
        // then throw a ServletException
        ...
    }
}
```

getInitParameter

`public java.lang.String` **`getInitParameter`**`(java.lang.String name)`

Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist. See `ServletConfig#getInitParameter`.

This method is supplied for convenience. It gets the value of the named parameter from the servlet's `ServletConfig` object.

Specified by:

`getInitParameter` in interface `ServletConfig`

Parameters:

`name` - a `String` specifying the name of the initialization parameter

Returns:

String a `String` containing the value of the initialization parameter

---

getInitParameterNames

`public java.util.Enumeration<java.lang.String>` **`getInitParameterNames`**`()`

Returns the names of the servlet's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the servlet has no initialization parameters. See `ServletConfig#getInitParameterNames`.

This method is supplied for convenience. It gets the parameter names from the servlet's `ServletConfig` object.

Servlets-Related Classes

- A **destroy** method complements an init method. If the init method claims resources, these should be released in the destroy method:

```
public void destroy() {
    if (theDBConnection != null) {
        try {
            theDBConnection.close();
        } catch(Exception e) { }
    }
}
```

HttpServletRequest

- The class javax.servlet.http.HttpServletRequest acts as a wrapper for the HTTP request data.
- The class defines a large number of methods that can be used to access specific parts of the request data.
- These access method include:
  - Methods returning general data about the request (standard HTTP information):
    - String getRemoteAddr
    - String getRemoteHost
    - String getRemoteUser – relies on use of HTTP authentication
    - String getMethod
    - int getServerPort
    - String getServerName

HttpServletRequest

- Methods returning data that form part of the HTTP request header:
  - Enumeration getHeaderNames
  - String getHeader(String name)
    - Ask for any specific HTTP header: e.g. CHARSET ACCEPT
  - Locale getLocale
    - Packaged version of Accept-Language header data
  - String getAuthType
    - Returns information about form of HTTP authentication (if any) used
  - Cookie[] getCookies
    - Returns array with all cookies that arrived with HTTP header. A Cookie object holds a name/value pair along with supplementary data such as domain, path and maximum age in seconds.

HttpServletRequest

- Methods relating to request data:
  - Enumeration getParameterNames
  - String getParameter(String name)
  - String[] getParameterValues(String name)
  - String getQueryString
  - int getContentLength
  - String getContentType
  - ServletInputStream getInputStream
    - Provides an input stream for reading binary data (e.g. upload of an image)
  - BufferedReader getReader
    - Provides a reader for text input.

HttpServletRequest

- Methods for passing data between servlets:
  - Object getAttribute(java.lang.String name)
  - void setAttribute(java.lang.String name, java.lang.Object o)

- Method relating to session support:
  - Session getSession()…

HttpServletRequest

- If a servlet encounters some problem – bad input data, some mishap with the database – it is going to have to return an error page rather than a dynamic response page.
- Quite often, the contents of the error page are fixed (just a prettied-up version of a message like '*Sorry, the database is unavailable. Please try later.*').
- It is always best to minimize the amount of HTML tags and content text buried in a servlet, so rather than have the error messages handled by the servlet you could simply redirect the request to a static web page:

```
if(dbConnection==null) {
    response.sendRedirect("/demo/NoDB.html");
    return;
}
```

HttpServletResponse

- The HttpServletResponse class defines (or inherits) a large number of methods including:
  - Methods relating to content output streams and buffering:
    - void setBufferSize(int size)
    - void flushBuffer
      - Forces any content in the buffer to be written to the client.
    - boolean isCommitted
      - Returns a boolean indicating whether the response has been 'committed' (i.e. header data have already been finalized and returned to web server and maybe to client as well).
    - ServletOutputStream getOutputStream
      - Get OutputStream suitable for binary data.
    - java.io.PrintWriter getWriter
      - Get PrintWriter to send textual response.

HttpServletResponse

- Methods for setting header data:
  - void setContentType(java.lang.String type)
    - Use to set any content type other than the default text/plain.
  - void setContentLength(int len)
  - void setLocale(java.util.Locale loc)
    - Sets headers such as charset and language
  - void addCookie(Cookie cookie)
  - void addDateHeader(java.lang.String name, long date)
  - void addHeader(java.lang.String name, java.lang.String value)
  - void sendError(int sc)
  - void sendError(int sc, java.lang.String msg)
  - void setStatus(int sc)
- The HttpServletResponse class defines a series of constants (SC_OK, SC_CREATED etc.) that correspond to the standard HTTP response codes.

HttpServletResponse

- In addition, there are helper methods like encodeURL(String).
  - The encodeURL method is used if it is necessary to use the URL rewriting approach for maintaining session state .