JSP 2.0 Expression Language

JSP 2.0 introduced a shorthand language for evaluating and outputting the values of Java objects that are stored in standard locations. This expression language (EL) is one of the two most important features of JSP 2.0; the other is the ability to define custom tags with JSP syntax instead of Java syntax. The EL cannot be used in servers that support only JSP 1.2 or earlier.

There are a number of strategies that JSP pages use to invoke Java code. One of the best and most flexible options is the MVC

- a servlet responds to the request
- invokes the appropriate business logic or data access code
- places the resultant data in beans
- stores the beans in the request, session or servlet context
- forwards the request to a JSP page to present the result

The one inconvenient part about this approach is the final step: presenting the results in the JSP page. You normally use **jsp:useBean** and **jsp:getProperty**, but these elements are a bit verbose and clumsy. Furthermore, **jspgetProperty** only supports access to simple bean properties; if the property is a collection or another bean, accessing the "subproperties" requires you to use complex Java syntax.

The JSP 2.0 expression language lets you simplify the presentation layer by replacing hard-to-maintain Java scripting elements or clumsy **jsp:useBean** and **jsp:getProperty** elements with short and readable entries of the following form.

```
${expression}
```

In particular, the expresion language supports the following capabilities.

- **Concise access to stored objects.**
  To output a "scoped variable" named **saleItem**, you use ${saleItem}
- **Shorthand notation for bean properties.**
  To output the companyName property (i.e., result of the getCompanyName method) of a scoped variable named company, you use ${company.companyName}. To access the firstName property of the president property of a scoped variable named company, you use ${company.president.firstName}.
- **Simple access to collection elements.**
  To access an element of an array, List, or Map, you use ${variable[indexOrKey]}.
- **Simple access to request parameters, cookies, and other request data.**
  To access the standard types of request data, you can use one of several predefined implicit objects.
- **A Small but useful set of simple operators.**
  To manipulate objects within EL expressons, you can use any of several arithmetic, relational, logical, or empty testing operators.
- **Conditional output.**
  To choose among output options, you can use `${test ? option1 : option2}`
- **Automatic type conversion.**
  The EL removes the need for most typecasts and for much of the code that parses strings as numbers
- **Empty values instead of error messages.**
  In most cases, missing values or NullPointerExceptions result in empty strings, not thrown exceptions.

Invoking the Expression Language

The EL elements can appear in ordinary text or in JSP tag attributes. For example:

```
<UL>
   <LI>Name: ${expression1}
   <LI>Address: ${expression2}
</UL>

<jsp:include page="${expr1}yusuf${expr2}"/>
```

Preventing Expression Language Evaluation

In JSP 1.2 and earlier, `${...}` had no special meaning. So, it is possible the characters `${` appear within a previously created page that is now being used on a server that supports JSP 2.0. In such a case, you need to deactivate the EL in that page. There are 4 options to do so:

1. **Deactivating the expression language in an entire Web application.**
   You use a web.xml file that refers to servlets 2.3 (JSP 1.2) or earlier.
2. **Deactivating the expression language in multiple JSP pages.**
   You use the jsp-property-group web.xml element to designate the appropriate pages.
3. **Deactivating the expression language in individual JSP pages.**
   You use the isELEnabled attribute of the page directive.
4. **Deactivating individual expression language statements.**
   In JSP 1.2 pages that need to be ported unmodified across multiple JSP versions (with no web.xml changes), you can replace $ with &#36;, the HTML character entity for $. In JSP 2.0 pages that contain both expression language statements and literal ${ strings, you can use \${ when you want ${ in the output.

1. Deactivating the Expression Language in an Entire Web Application

The JSP 2.0 expression language is automatically deactivated in Web applications whose deployment descriptor (i.e., WEB-INF/web.xml file) refers to servlet specification version 2.3 or earlier (i.e., JSP 1.2 or earlier). For example, the following empty-but-legal web.xml file is compatible with JSP 1.2, and thus indicates that the expression language should be deactivated by default.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

On the other hand, the following web.xml file is compatible with JSP 2.0, and thus stipulates that the EL should be activated by default.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd version="2.4">
</web-app>
```

2. Deactivating the Expression Language in Multiple JSP Pages

In a Web application whose deployment descriptor specifies servlets 2.4 (JSP 2.0), you use the el-ignored subelement of the jsp-property-group web.xml element to designate the pages in which the expression language should be ignored. Here is an example that deactivates the expression language for all JSP pages in the legacy directory.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd version="2.4">

<jsp-property-group>
  <url-pattern>/legacy/*.jsp</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-property-group>

</web-app>
```

3. Deactivating the Expression Language in Individual JSP Pages

To disable EL evaluation in an individual page, supply false as the value of the isELEnabled attribute of the page directive, as follows.

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

Note that the isELEnabled attribute is new in JSP 2.0 and it is an error to use it in a server that supports only JSP 1.2 or earlier. So, you cannot use this technique to allow the same JSP page to run in either old or new servers without modification. Consequently, the jsp-property-group element is usually a better choice than the isELEnabled attribute.

4. Deactivating Individual Expression Language Statements

Suppose you have a JSP 1.2 page containing ${ that you want to use in multiple places. In particular, you want to use it in both JSP 1.2 Web applications and in Web applications that contain expression language pages. You want to be able to drop the page in any Web application without making any changes either to it or to the web.xml file. Although this is an unlikely scenario, you simply replace the $ with the HTML character entity corresponding to the ISO 8859-1 value of $ (36). So, you replace ${ with &#36;{ throughout the page. For example,

```
&#36;{blah}
```

will portably display

```
${blah}
```

to the user. Note, however, that the character entity is translated to $ by the *browser*, not by the *server*, so this technique will only work when you are outputting HTML to a Web browser. Finally, suppose you have a JSP 2.0 page that contains both expression language statements and literal ${ strings. In such a case, simply put a backslash in front of the dollar sign. So, for example,

```
\${1+1} is ${1+1}.
```

will output

```
${1+1} is 2.
```

I. Accessing Scoped Variables

Objects in the *HttpServletRequest*, the *HttpSession*, or the *ServletContext*. are known as “scoped variables” and the EL has a quick and easy way to access them. Scoped variables may also be stored in the *PageContext* object, but this is much less useful because the servlet and the JSP page do not share *PageContext* objects. So, page-scoped variables apply only to objects stored earlier in the same JSP page, not to objects stored by a servlet.

To output a scoped variable, you simply use its name in an expression language element. For example,

```
   ${name}
```

means to search the PageContext, HttpServletRequest, HttpSession, and ServletContext (in that order) for an attribute named name. If the attribute is found, its toString method is called and that result is returned. If nothing is found, an empty string (not null or an error message) is returned. So, for example, the following two expressions are equivalent.

```
   ${name}
   <%= pageContext.findAttribute("name") %>
```

The problems with the latter approach are that it is verbose and it requires explicit Java syntax. It is possible to eliminate the Java syntax, but doing so requires the following even more verbose jsp:useBean code.

```
   <jsp:useBean id="name" type="somePackage.SomeClass" scope="...">
   <%= name %>
```

To illustrate access to scoped variables, the ScopedVars servlet stores a String in the HttpServletRequest, another String in the HttpSession, and a Date in the ServletContext. The servlet forwards the request to a JSP page that uses ${*attributeName*} to access and output the objects.

Notice that the JSP page uses the same syntax to access the attributes, regardless of the scope in which they were stored. This is usually a convenient feature because MVC servlets almost always use unique attribute names for the objects they store. However, it is technically possible for attribute names to be repeated, so you should be aware that the EL searches the PageContext, HttpServletRequest, HttpSession, and ServletContext *in that order*.

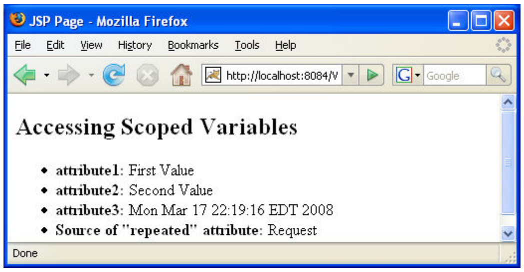
```
public class ScopedVars extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IO...
   {
      request.setAttribute("attribute1", "First Value");
      HttpSession session = request.getSession();
      session.setAttribute("attribute2", "Second Value");

      ServletContext application = getServletContext();
      application.setAttribute("attribute3", new java.util.Date());

      request.setAttribute("repeated", "Request");
      session.setAttribute("repeated", "Session");
      application.setAttribute("repeated", "ServletContext");

      RequestDispatcher dispatcher = request.getRequestDispatcher("index1.jsp");
      dispatcher.forward(request, response);
    }
}
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h2>Accessing Scoped Variables</h2>
        <UL>
            <LI><B>attribute1:</B> ${attribute1}
            <LI><B>attribute2:</B> ${attribute2}
            <LI><B>attribute3:</B> ${attribute3}
            <LI><B>Source of "repeated" attribute:</B> ${repeated}
        </UL>
    </body>
</html>
```



Accessing Scoped Variables

- **attribute1**: First Value
- **attribute2**: Second Value
- **attribute3**: Mon Mar 17 22:19:16 EDT 2008
- **Source of "repeated" attribute**: Request

forward(ServletRequest request, ServletResponse response) vs. sendRedirect(String location)

public interface RequestDispatcher

Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the RequestDispatcher object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

The forward method of RequestDispatcher will forward the ServletRequest and ServletResponse that it is passed to the path that was specified in getRequestDispatcher(String path). The response will not be sent back to the client and so the client will not know about this change of resource on the server. This method is useful for communicating between server resources, (servlet to servlet). Because the request and response are forwarded to another resource all request parameters are maintained and available for use. Since the client does not know about this forward on the server, no history of it will be stored on the client, so using the back and forward buttons will not work. This method is faster than using sendRedirect as no network round trip to the server and back is required. An example using forward:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOExcep.
{
  RequestDispatcher rd = request.getRequestDispatcher("pathToResource");
  rd.forward(request, response);
}
```

The sendRedirect(String path) method of HttpServletResponse will tell the client that it should send a request to the specified path. So the client will build a new request and submit it to the server, because a new request is being submitted all previous parameters stored in the request will be unavailable. The client's history will be updated so the forward and back buttons will work. This method is useful for redirecting to pages on other servers and domains. An example using sendRedirect:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOExcep.
{
  response.sendRedirect("pathToResource");
}
```

II. Accessing Bean Properties

When you simply use ${name}, the system finds the object named name, converts it to a String, and returns it. Although this behavior is convenient, you rarely want to output *the actual object* that the MVC servlet stored. Rather, you typically want to output *individual properties* of that object. The JSP expression language provides a simple but very powerful dot notation for accessing bean properties. To return the firstName property of a scoped variable named customer, you merely use `${customer.firstName}`. Although this form appears very simple, the system must perform reflection (analysis of the internals of the object) to support this behavior. So, assuming the object is of type NameBean, to do the same thing with explicit Java syntax, you would have to replace

```
${customer.firstName}
```

with

```
<%@ page import="com.yourPackage.NameBean" %>
<%
   NameBean person = (NameBean)pageContext.findAttribute("customer");
%>
<%= person.getFirstName() %>
```

```
package com.yusuf;

import javax.servlet.http.HttpSession;

public class YusufBean
{
   String user;

   public void setUser(String userName)
   {
      user = userName;
   }

   public String getUser()
   {
      return user;
   }
}
```

```
<jsp:directive.page session="true" />
<jsp:useBean id="bean" class="com.yusuf.YusufBean"/>

<HTML>
<HEAD>
   <TITLE>My JSP</TITLE>
</HEAD>

<BODY>

<%
   bean.setUser("Yusuf Ozbek");
%>

Welcome ${bean.user}

</BODY>
</HTML>
```

III. Accessing Collections

The JSP 2.0 expression language lets you access different types of collections in the same way: using array notation. For instance, if attributeName is a scoped variable referring to an array, List, or Map, you access an entry in the collection with the following:

```
${attributeName[entryName]}
```

If the scoped variable is an array, the entry name is the index and the value is obtained with theArray[index]. For example, if customerNames refers to an array of strings,

```
${customerNames[0]}
```

would output the first entry in the array.

If the scoped variable is an object that implements the List interface, the entry name is the index and the value is obtained with theList.get(index). For example, if supplierNames refers to an ArrayList,

```
${supplierNames[0]}
```

would output the first entry in the ArrayList. If the scoped variable is an object that implements the Map interface, the entry name is the key and the value is obtained with theMap.get(key). For example, if stateCapitals refers to a HashMap whose keys are U.S. state names and whose values are city names,

```
${stateCapitals["maryland"]}
```

would return "annapolis". If the Map key is of a form that would be legal as a Java variable name, you can replace the array notation with dot notation. So, the previous example could also be written as:

```
${stateCapitals.maryland}
```

However, note that the array notation lets you choose the key at request time, whereas the dot notation requires you to know the key in advance.

```
public class CollectionServlet extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException
   {
      String[] firstNames = { "Bill", "Scott", "Larry" };

      ArrayList lastNames = new ArrayList();
      lastNames.add("Ellison");
      lastNames.add("Gates");
      lastNames.add("McNealy");

      HashMap companyNames = new HashMap();
      companyNames.put("Ellison", "Sun");
      companyNames.put("Gates", "Oracle");
      companyNames.put("McNealy", "Microsoft");

      request.setAttribute("first", firstNames);
      request.setAttribute("last", lastNames);
      request.setAttribute("company", companyNames);

      RequestDispatcher dispatcher = request.getRequestDispatcher("collections.jsp");
     dispatcher.forward(request, response);
   }
```

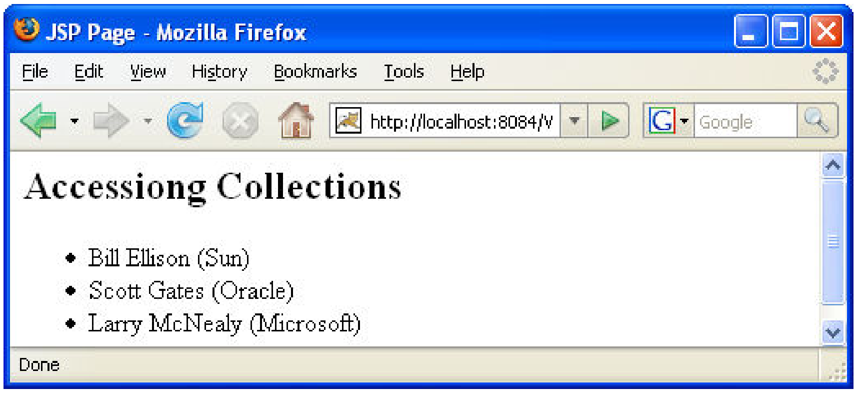
```
<html>
<head>
 <title>JSP Page</title>
</head>

<body>

<h2>Accessiong Collections</h2>

<UL>
 <LI>${first[0]} ${last[0]} (${company["Ellison"]})
 <LI>${first[1]} ${last[1]} (${company["Gates"]})
 <LI>${first[2]} ${last[2]} (${company["McNealy"]})
</UL>

</body>
</html>
```



Accessiong Collections

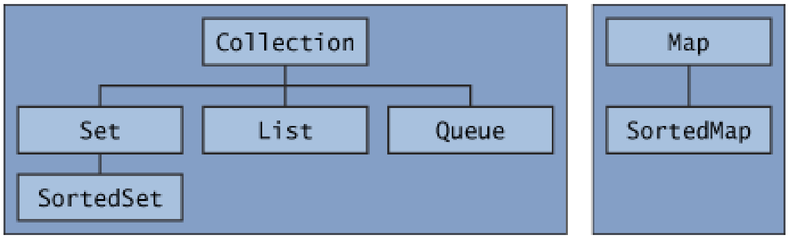
- Bill Ellison (Sun)
- Scott Gates (Oracle)
- Larry McNealy (Microsoft)

Java Collections Framework

The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees — a Map is not a true Collection.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated *optional* — a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an UnsupportedOperationException. Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

- Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.
- Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.
  Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.
- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.

The last two core collection interfaces are merely sorted versions of Set and Map:

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.
- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

IV. Referencing Implicit Objects

In most cases, you use the JSP expression language in conjunction with the Model- View-Controller architecture in which the servlet creates the data and the JSP page presents the data. In such a scenario, the JSP page is usually only interested in the objects that the servlet created, and the general bean-access and collection-access mechanisms are sufficient.

However, the expression language is not restricted to use in the MVC approach; the expression language can be used in any JSP page. To make this capability useful, the specification defines the following implicit objects.

- **pageContext**
  The pageContext object refers to the PageContext of the current page. The PageContext class, in turn, has request, response, session, out, and servletContext properties (i.e., getRequest, getResponse, getSession, getOut, and getServletContext methods). So, for example, the following outputs the current session ID.

  ```
  ${pageContext.session.id}
  ```

- **param and paramValues**
  These objects let you access the primary request parameter value (param) or the array of request parameter values (paramValues). So, for example, the following outputs the value of the custID request parameter (with an empty string, not null, returned if the parameter does not exist in the current request).

  ```
  ${param.custID}
  ```

- **cookie**
  The cookie object lets you quickly reference incoming cookies. However, the return value is the Cookie object, not the cookie value. To access the value, use the standard value property (i.e., the getValue method) of the Cookie class. So, for example, either of the following outputs the value of the cookie named userCookie (or an empty string if no such cookie is found).

  ```
  ${cookie.userCookie.value}
  ${cookie["userCookie"].value}
  ```

- **pageScope, requestScope, sessionScope, and applicationScope**
  These objects let you restrict where the system looks for scoped variables. For example, with

  ```
  ${name}
  ```

  the system searches for `name` in the `PageContext`, the `HttpServlet-Request`, the `HttpSession`, and the `ServletContext`, returning the first match it finds. On the other hand, with

  ```
  ${requestScope.name}
  ```

  the system only looks in the `HttpServletRequest`.

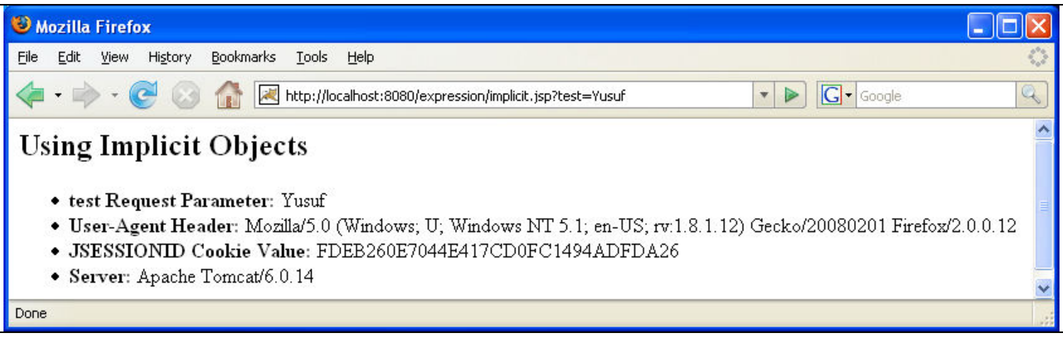
- **header and headerValues**
  These objects access the primary and complete HTTP request header values, respectively. Remember that dot notation cannot be used when the value after the dot would be an illegal property name. So, to access the Accept header, you could use either

  `${header.Accept}` or `${header["Accept"]}`

  But, to access the Accept-Encoding header, you must use

  ```
  ${header["Accept-Encoding"]}
  ```

```
<H2>Using Implicit Objects</H2>
<UL>
<LI><B>test Request Parameter:</B> ${param.test}
<LI><B>User-Agent Header:</B> ${header["User-Agent"]}
<LI><B>JSESSIONID Cookie Value:</B> ${cookie.JSESSIONID.value}
<LI><B>Server:</B> ${pageContext.servletContext.serverInfo}
</UL>
```



Using Implicit Objects

- **test Request Parameter:** Yusuf
- **User-Agent Header:** Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.12) Gecko/20080201 Firefox/2.0.0.12
- **JSESSIONID Cookie Value:** FDEB260E7044E417CD0FC1494ADFDA26
- **Server:** Apache Tomcat/6.0.14

Scope

Any object in a JSP or Servlet has an area within the web application in which it is visible to other objects, expressions, or scriptlets. There are 4 different levels of scope:

1.Page scope

- The first level is page scope, which is restricted to a single JSP page.
- Attributes are stored in the **PageContext** object
- Any object created with page scope is only accessible within the page where it is created.
- Once the response has been sent to the client, all references to objects with page scope are released.

2.Application scope

- All servlets in the web application have access
- Attributes are stored in the **ServletContext** object
- Available for the lifetime of the servlet

3.Session scope

- Available to servlets that have access to this specific session
- Attributes are stored in the **HttpSession** object
- Available for the life of the session

4.Request scope

- Available to servlets that have access to this specific request
- Attributes are stored in the **ServletRequest** object
- Available for the life of the request (until your doGet or doPost method completes)

Page-scoped data is accessible only within the JSP page and is destroyed when the page has finished generating its output for the request.

Request-scoped data is associated with the request and destroyed when the request is completed.

Session-scoped data is associated with a session and destroyed when the session is destroyed.

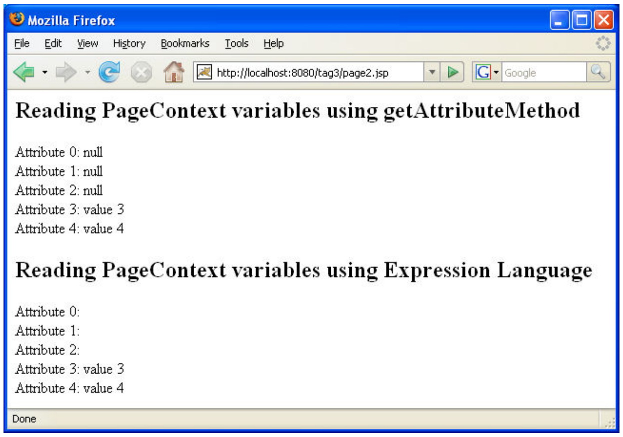
Application-scoped data is associated with the web application and destroyed when the web application is destroyed.

```
PageContext variables are set.
<%
      pageContext.setAttribute("attrib0", "value 0");  // PAGE_SCOPE is the default
      pageContext.setAttribute("attrib1", "value 1", PageContext.PAGE_SCOPE);
      pageContext.setAttribute("attrib2", "value 2", PageContext.REQUEST_SCOPE);
      pageContext.setAttribute("attrib3", "value 3", PageContext.SESSION_SCOPE);
      pageContext.setAttribute("attrib4", "value 4", PageContext.APPLICATION_SCOPE);
%>

<P><A HREF='page2.jsp'>Go to page 2</A></P>
```

```
<HTML>
<BODY>
<H2>Reading PageContext variables using getAttributeMethod</H2>
Attribute 0: <%= pageContext.getAttribute("attrib0") %><br>
Attribute 1: <%= pageContext.getAttribute("attrib1", PageContext.PAGE_SCOPE) %><br>
Attribute 2: <%= pageContext.getAttribute("attrib2", PageContext.REQUEST_SCOPE) %><br>
Attribute 3: <%= pageContext.getAttribute("attrib3", PageContext.SESSION_SCOPE) %><br>
Attribute 4: <%= pageContext.getAttribute("attrib4", PageContext.APPLICATION_SCOPE) %><br>

<H2>Reading PageContext variables using Expression Language</H2>
Attribute 0: ${attrib0}<br>
Attribute 1: ${attrib1}<br>
Attribute 2: ${attrib2}<br>
Attribute 3: ${attrib3}<br>
Attribute 4: ${attrib4}
</BODY>
</HTML>
```



Reading PageContext variables using getAttributeMethod

```
Attribute 0: null
Attribute 1: null
Attribute 2: null
Attribute 3: value 3
Attribute 4: value 4
```

Reading PageContext variables using Expression Language

```
Attribute 0:
Attribute 1:
Attribute 2:
Attribute 3: value 3
Attribute 4: value 4
```

Creating Tag Files

JSP specification version 2.0 introduced a JSP-based way to create custom tags using tag files. One of the key differences between Java-based custom tags and JSP-based custom tags (tag files) is that with Java-based tags the tag handler is a Java class, whereas with JSP-based tags the tag handler is a JSP page. Tag files are also a bit simpler to write because they don't require you to provide a TLD. If there is a lot of HTML formatting, use tag files to create output. If there is a lot of logic, use Java to create output.

In general, there are two steps to creating a JSP-based custom tag.

1. Create a JSP-based tag file

This file is a fragment of a JSP page with some special directives and a .tag extension. It must be placed inside the /WEB-INF/tags or a subdirectory thereof.

2. Create a JSP page that uses the tag file

The JSP page points to the directory where the tag file resides. The name of the tag file (without the .tag extension) becomes the name of the custom tag.

A tag file is simply a plain text file with a file extension of .tag. Other than the JSP page directive, all the usual JSP elements can be used within this file. Let's start off really simple: Create a file called *tagDemo.tag* and put the following text inside:

```
<%
 String whoAmI = "I am a String within a scriptlet";
%>
Hello There,  <%= whoAmI %>
```

The next step is to create your test jsp page and add the following line:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
```

The taglib directive above simply tells the jsp container where to find your tag (the tagdir attribute) and what prefix you will be using to identify your tag (the prefix attribute). To use the tag requires a combination of the prefix and the tag file name in the following form:

```
<tags:tagDemo/>
```

Using doBody to access body content

One of the more useful things about custom tags is the fact that you can access the tag's body content and manipulate or make use of it.

```
<tags:uppercase>I want to be in capital letters.</tags:uppercase>
```

This is where the <jsp:doBody/> action comes in to play. <jsp:doBody/> is an example of a jsp action that can only be used inside a tag file.

```
<jsp:doBody var="theBody"/>
<%
   String bc = (String) jspContext.getAttribute("theBody");
%>
<%= bc.toUpperCase() %>
```

The first line is the <jsp:doBody/> declaration. What is important here is the *var* attribute, which places a String variable into the pageContext of the tag file. This variable contains the actual body content of the doBody tag. The <jsp:doBody/> tag has a couple other features you may find interesting.

```
<%@ tag body-content="scriptless"  %>
<jsp:doBody var="theBody" scope="page"/>
<%
    String bc = (String) jspContext.getAttribute("theBody");
%>
<%= bc.toUpperCase() %>
```

The code above does exactly the same thing as the previous code.

Notice that the <jsp:doBody/> tag now has a scope attribute; you can use this to export the variable to any of the usual JSP scopes. The default is 'page'.

You should also know about the *varReader* attribute—you can use this instead of the *var* attribute to have the variable stored as a java.io.Reader instead of a String. In some circumstances this may be more efficient when processing large amounts of body content.

doBody without any attributes

What happens if you use <jsp:doBody/> without any attributes at all? In this case the body content is printed directly back to the page.

Adding attributes

Attributes are the mechanism that custom tags use to allow you, or the users of your tags, to customize the way that the tag behaves.

```
<%@attribute name="color" required="true" %>
```

This lets your tag file know that you are expecting an attribute called 'color' to be passed along. Here is how the box tag is used in a jsp:

```
<tags:box color="yellow" >I want to be inside a colored box!</tags:box>
```

Supplying the attribute is straightforward, but how do you access the value from inside our tag file? It shows up in the page scope.

```
<%
   String theColor = (String) jspContext.getAttribute("color");
%>
```

With the JSP 2.0 expression language, here's a little leaner version of the box.tag file:

```
<%@ attribute name="color" required="true"  %>

<table width='400" cellpadding="5"  bgcolor="${pageScope.color}" >
    <tr>
        <td>
            <jsp:doBody/>
        </td>
    </tr>
</table>
```

The values you supply to tag file attributes can be dynamic in nature—you are not limited to static string values. Having said that, there is an attribute you can use with the *attribute* action called *rtexprvalue*—set this to false if you want to turn off this feature and use just static values.