



CHAPTER 2

THE CORE ACTIONS



OBJECTIVES

After completing “The Core Actions,” you will be able to:

- Describe the purpose and role of the JSTL core library in JSP coding.
- Use core actions to complement standard actions, custom actions, and JSP expressions for seamless, script-free page logic.
 - Produce dynamic output based on JSP expressions.
 - Find or instantiate objects at various scopes, and set their values.
 - Direct conditional processing of page content based on dynamic tests.
 - Direct iterative processing of page content by looping through ranges of numbers, over elements in a collection, or over tokens in a master string.
 - Import external resources by URL for processing, or redirect the JSP container to an external resource to handle the current request.

The JSTL Core Library

- The most basic of the four libraries in the JSTL, the **core** library includes actions to perform some very simple tasks:
 - **Produce output** to the response stream
 - **Export variables** (objects) at various scopes, and set their values to static content or JSP expressions
 - Direct processing flow through **conditionals and loops**
 - **Import external resources** for use in page processing
 - **Redirect** processing to another page or resource
- In a page that uses any of the core tags, be sure to include the **taglib** directive that makes the library visible.

```
<%@ taglib prefix="c"  
    uri="http://java.sun.com/jsp/jstl/core" %>
```

- Again, the choice of prefix is the developer's; we will be adhering to the conventional prefixes in this module, and so we use **c:** for core actions.

A New Tool

- The core actions fill in some critical gaps between other JSP programming techniques.
- JSP expressions are wonderfully simple and offer a way to include dynamic content in a page without a lot of coding.
 - They can only read values, not write them.
- Standard actions have the advantages of the XML-like JSP action syntax, and they offer a simple interface to JavaBeans.
 - They can only work with beans, however.
 - Also they have no way of referencing indexed properties.
- JSTL core actions offer a simple way to work with objects of all types.
 - JSTs can manage primitive types as well as JavaBeans; primitives are represented as “boxed” objects of types such as **Integer** or **Float**.
 - JSTs can work with and also with indexed properties and collections: list/vector/array structures as well as key/value maps.

Documenting Standard Tags

- In this module, we'll document each of the standard tags by a standard notation, for easy reference.

```
<prefix:action
  attribute1 = "dynamic | static"
  requiredAttribute = "dynamic | static"
  ...
  attributeN = "dynamic | static"
/>
```

- Each attribute is shown.
 - The value **"dynamic"** will indicate that the attribute value can include JSP expressions; **"static"** indicates that it cannot.
 - **Required attributes** are underlined.
 - Notes and discussion of the behavior of the action occupy the rest of the page.
 - Sometimes no one attribute is required, but there are more complex constraints that dictate that one of a set must be used. These sorts of constraints escape this notation, but can be stated in the notes and discussion.
- **The summary notation will also indicate whether the tag can or must have body content.**
 - The template shown above is for an empty element.
 - If body content is possible (or mandatory), a short description will be included as the body of the summary tag, and a separate end tag will be shown.

<c:out>

```
<c:out
  value = "dynamic"
  default = "dynamic"
  escapeXml = "dynamic"
>
  default value
</c:out>
```

- The **<c:out>** action produces output to the response stream.
 - The **value** attribute defines the desired output, whether static or JSP expression content.
 - If the value is an expression that evaluates to **null**, then the default value is produced instead.
 - If the tag has a body, it provides the default value.
- In a JSP 2.0 container, the value expression alone would accomplish just about the same thing as using **<c:out>**.
 - This tag is handy when a page might need to function in either a **JSP 1.x or JSP 2.0 container**.
 - It is also good practice to use **<c:out>** when producing values derived directly from user input.
 - **<c:out>** automatically performs **output escaping**, which means that it can't generate HTML markup.
 - This feature guards against **cross-site scripting (XSS)** attacks.

<c:set>

```
<c:set
  var = "static"
  value = "dynamic"
  target = "dynamic"
  property = "dynamic"
  scope = "static"
>
  value
</c:set>
```

- This action finds or instantiates an object and sets its value.
- There are two distinct ways of invoking this action:
 - Identifying an object to set with the **var** attribute.

```
<c:set var="found" value="${true}" />
```

- Identifying an object **target** on which to set a specific **property**.

```
<c:set target="${team}" property="goalie"
  value="Rejean Lemelin" />
```

- In either case, the **value** attribute (or the body content) provide the value to set.
- In either case, the **scope** attribute qualifies the assignment target: either **var** or **target** must be found or instantiated under this scope.

Setting Properties

- In the **target/property** usage, the two attributes are evaluated in a way similar to the evaluation of a JSP expression **target.property**:
 - If **target** is a map, then **property** is treated as a key value in that map, and the corresponding value is set to **value**.
 - If the key does not exist, a new pair is added to the map: key=**property**, value=**value**.
 - If the key exists but the provided **value** is null, then the key is removed from the map.
 - If the **target** is not a map, then it is treated as a JavaBean and a mutator for **property** is sought under Beans naming conventions – that is, the action behaves very much like `<jsp:setProperty>`.
- Note that scalar collections – lists, vectors, arrays, etc. – are left out of this usage entirely.
 - That is, **property** will not be evaluated as a numeric index for a scalar collection type. If it were, one could use this action to set the Nth element of a collection to a new value.
 - Generally speaking, non-map collections are not addressed effectively by JSTL. JSP expressions do read such collections easily, but modifying collections is not well supported by standard actions or by JSTL.

Gotchas

- Note that both of these attributes accept EL expressions.
- This is a good thing, but a bit counterintuitive in the case of **target**.
 - You must use an EL expression to indicate the target object.
 - Otherwise, the expression will resolve to a string! You must provide a value for **target** that resolves to an object reference.
 - This is in contrast to the **var** attribute; **var** creates an object, where **target** references something that already exists.
 - Thus the first expression below will work but the second will not:

```
<c:set var="myNumber" value="5" />
<c:set target="myMap" property="five" value="5" />
```

- Correct is:

```
<c:set target="{myMap}" property="five"
        value="5" />
```

- There are various other examples of this throughout JSTL.
 - Each one makes sense when studied carefully, but does not make for intuitive usage.

<c:remove>

```
<c:remove
  var = "static"
  scope = "static"
/>
```

- This action removes a object at a given scope.
- Beyond good citizenship in cleaning up unneeded objects at session and application scopes, this action has a couple of common uses:
 - An object might indicate a state by its **absence or presence**. That is, instead of a separate boolean object such as “filteringEnabled”, one might rely on the presence of a “filter” object to indicate enabled and its absence to indicated a disabled state.
 - An object might be useful for some quick process but then be **hiding** a higher-scope object that needs to be used in a JSP expression. For instance, the output of the following code will be as desired only with the <c:remove> action as shown:

```
<c:set var="x" value="what we want"
      scope="session" />
<c:set var="x" >what we don't want</c:set>
<c:remove var="x" />
<c:out value="\${x}" />
```

Flow Control Actions

- The core library offers two ways of conditionally processing JSP content:
 - The `<c:if>` action controls processing of its body content by a single **test** attribute. This is a “yes or no” evaluation; there is no “if/else” construct.
 - The `<c:choose>` action provides “if/else” and “switch/case” logic, by including one or more `<c:when>` blocks and a single `<c:otherwise>` that functions as the “else” or “default” choice.
- It also provides two actions for creating loops over JSP content:
 - `<c:forEach>` allows looping over a range of numeric values or over a collection object.
 - `<c:forTokens>` will iterate over every token in a master string based on provided delimiters. This would be a specialty function in most languages, but it’s a natural basic function in JSP, where string processing and tokenizing is such common stuff.

<c:if>

```
<c:if
  test = "dynamic"
  var = "static"
  scope = "static"
>
  body content to process if test is true
</c:if>
```

- This action processes the tag body if the **test** condition evaluates to a boolean **true**.
 - The optional **var** and **scope** attributes identify a boolean object whose value will be set to the value of the test expression.
 - Either the target **var** or the body content to process, or both, may be included.

- The **test** attribute is another tricky one, like **target on <c:set>**.

- That is, since only JSP expressions as the value of the **test** attribute make any intuitive sense, it's easy to forget the **#{ }** characters in the test expression.

```
<c:if test="param.choice = 'Retry'" >
  You chose to retry the connection.
</c:if>
```

- This will always fail, since the literal string doesn't convert to boolean **true**. The correct usage is:

```
<c:if test="{param.choice = 'Retry'}" > ...
```

- In **Examples\LevelsBlind\Step1**, the code in **Tags\LevelsBlind\filter.tag** tests a given member against the query criteria:

```
<c:if test="{sex == param.sex &&
    age + 4 > param.age && age - 4 < param.age}">
    ...
</c:if>
```

- We'll see this code in action in a later example, as it combines with looping actions to complete the filtering action.

<c:choose> and Related Tags

<c:choose>

```
<c:when test = "dynamic" > <!-- 1 Mandatory -->
  possible content
```

```
</c:when>
```

```
<c:when test = "dynamic" > <!-- 2-N Optional -->
  possible content
```

```
</c:when>
```

```
<c:otherwise>default content</c:otherwise>
```

```
<!-- Optional -->
```

```
</c:choose>
```

- <c:choose> serves to organize invocations of two other actions, <c:when> and <c:otherwise>.
- <c:choose> may only contain these tags.
 - There must be a <c:when>; there may be more than one.
 - There may be a <c:otherwise>, but only one.
 - Neither of these two tags can be used legally except as the direct child of a <c:choose>.
- The handler will process the <c:when> actions in sequence, and will process the content of the first one whose test expression evaluates to **true**.
 - Successive <c:when> actions will be ignored after a hit.
 - If present, the <c:otherwise> content will be processed only if none of the <c:when> tests is successful.

- In **Examples\OddEven\choose**, the JSP shown in the previous chapter has been rewritten slightly to use the `<c:choose>` construct instead of a pair of `<c:if>`s.

```

${param["number"]} is
<c:choose>
  <c:when test="${param['number'] % 2 == 0}">
    even.
    <c:set var="totalEven" scope="session"
      value="${totalEven + 1}" />
  </c:when>
  <c:otherwise>
    odd.
    <c:set var="totalOdd" scope="session"
      value="${totalOdd + 1}" />
  </c:otherwise>
</c:choose>

```

- In this case, the switch saves a certain amount of trouble in writing two mutually exclusive test expressions.
 - Another advantage of choose/when/otherwise over two unrelated tests is seen when there is a range of possible values, only one of which should produce a certain result.
 - A closely related case is when there are only two legal values, but the tested input has yet to be validated; the `<c:otherwise>` will catch unexpected values and assure that one of two possible behaviors always obtains.
- We'll defer testing of this JSP, too, since it includes other exemplary code that we'll see later in this chapter.

<c:forEach>

```
<c:forEach
  items = "dynamic"
  begin = "dynamic"
  end = "dynamic"
  step = "dynamic"
  var = "static"
  varStatus = "static"
>
  content to process zero-to-many times
</c:forEach>
```

- This action effects a loop based on some combination of criteria as provided by attribute values.
 - **begin**, **end**, and **step** attributes, if present, will control the number of times the loop content is processed. **varStatus** will be provided to body content to describe the loop status.
 - **items** identifies a collection over which the loop will iterate. **var** will be provided to body content and will hold the current element in the collection.
 - Both **var** and **varStatus** have nested scope.
 - Either of these usages is viable independent of the other; they can also be combined, with the **begin** and **end** attributes setting boundaries and put to use as indices to the collection.
 - Also note that **varStatus** is viable in the absence of the **begin** and **end** attributes.
- Note that **items** is another in that category of attributes whose value is obviously dynamic ... and therefore dangerous. Don't forget the `#{ }`!

Using <c:forEach>

- Thus there are many common usages of the <c:forEach> action:
 - A traditional “for loop” to process content some number of times in a row:

```
<c:forEach begin="1" end="10" >...
```

- A simple “for-each loop” to iterate over a list of objects:

```
<c:forEach var="element" items="{myList}" >...
```

- A bounded iteration over a collection – say, to process the third through fifth elements:

```
<c:forEach var="each" items="{list}"  
begin="2" end="4" >...
```

The LoopTagStatus Interface

- The **varStatus** attribute provides information about the status of a loop in progress.
- It is of type **LoopTagStatus**, which (like all JSTL support interfaces) is JavaBeans-compliant. Properties are:

```
current : Object
index  : int
count  : int
first  : boolean
last   : boolean
begin  : int
end    : int
step   : int
```

- Many of these attributes reflect values provided to the loop action: **begin**, **end**, and **step**.
- The **current** property is just the object that would be referenced by the **var** object, if it were defined.
- The remaining attributes are the most commonly used, typically to modify the behavior of the body logic at the beginning and/or ending of an iteration:

```
<c:forEach var="word" items="${list}"
  varStatus="status" >
  ${var}
  <c:if test="${!varStatus.last}" >, and</c:if>
</c:forEach>
```

Iterating Over Maps

- The `<c:forEach>` action can iterate over scalar collections and also over maps.
 - In the first case, the **var** object will be the element in the collection, referenced directly.
 - In the case of a map, there are two objects that must be available through **var**: the key and the value.
 - Thus for maps **var** is of type **Map.Entry**, which offers properties for each of the two:

```
key : Object  
value : Object
```

- Thus iterating over a map is slightly different:

```
<c:forEach var="pair" items="param" >  
  <li>${pair.key} = ${pair.value}</li>  
</c:forEach>
```

<c:forTokens>

```
<c:forTokens
  items = "dynamic"
  delims = "dynamic"
  begin = "dynamic"
  end = "dynamic"
  step = "dynamic"
  var = "static"
  varStatus = "static"
>
  content to process zero-to-many times
</c:forTokens>
```

- This action is dedicated to looping over tokens parsed from a master string.
 - Thus the **items** attribute is required here; there is no point in using this action to implement a “for loop” over a range of numbers, since **<c:forEach>** will do that just as well.
 - The **delims** attribute is also required: each character in the value of this attribute will be treated as a delimiter for string tokenizing.
 - As in **<c:forEach>**, the **var** attribute defines a nested-scope variable; here, however, it is always of type **String**.

The Tomcat Web Server

- The Apache Software Foundation provides the **Tomcat** web server as the reference implementation for JSPs and servlets.
 - Tomcat is a fully-functioning production web server, and as an open-source tool is also a tremendous resource for any web-application or HTTP development.
- Tomcat 5.0 is installed on your machine.
 - Set the environment variable **CATALINA_HOME**, typically to **c:\Capstone\Tools\Tomcat5.0**.
- This root has the following subdirectories, among others:
 - **bin** holds the binaries, including scripts you will use to start and shut down the server.
 - **conf** holds configuration files, notably a file **server.xml** that you will modify to enable the course software.
 - **webapps** is the default **context root**, under which Tomcat will look for web applications as requested by HTTP clients.
- Also, for some of our exercises, you will need to define the environment variable **CC_MODULE** to direct compilation of Java class files.
 - Set this to **c:\Capstone\JSTL**, unless your instructor directs you otherwise.



Deploying Servlets and JSPs

- To facilitate our work in this course, we will deploy a single web application covering the entire file set.
 - This is not best practice: more properly each example and lab should be considered a self-contained application and deployed via a **web archive**, or **.war**, file.
 - For this module, identifying the whole tree as one application saves us a good deal of deployment work, allows browsing over the whole tree with URLs that loosely match the various file paths, and speeds up development and debugging.
- Tomcat looks to its own **webapps** directory for applications when clients make requests.
- It also will attempt to resolve a requested URL to a **context** as defined in its configuration.
 - A context has a **URI path**, which is relative to the server's host/port address or name in a requested URI.
 - The context definition allows Tomcat to map that relative path to a specific directory on the **file system**.
 - Other attributes attach to the context definition, but we won't be concerned with that in this module.

Adding a Context to Tomcat

- A file has been prepared with an XML `<Context>` element that advises Tomcat to serve the JSTL module as a distinct application.
 - Find this file in **WEB-INF\JSTL.xml**:

```
<Context
  path="/JSTL"
  docBase="c:\Capstone\JSTL"
  reloadable="true"
>
</Context>
```

- The attributes are:
 - **path**, which defines the application context under which Tomcat will look to our module for resources
 - **docBase**, which is the root directory to which this application context is mapped
 - **reloadable**, which when “true” tells Tomcat to monitor our class files and libraries, and to reload the application when changes are detected – this will save us having to stop and restart Tomcat in a few cases
- Copy this file to **%CATALINA_HOME%\conf\Catalina\localhost.**

Starting and Stopping Tomcat

- Tomcat comes with scripts for startup and shutdown.
 - DOS batch files are provided for Windows systems.
 - Shell scripts are provided for UNIX systems.
- To start the server, simply run **%CATALINA_HOME%\bin\startup.bat**.
 - You should see a separate console appear, and after a few seconds there will be some simple diagnostic information shown.
 - The server will run in this console, listening on a few predefined IP ports for web requests.
 - As part of the startup, necessary directories and JARs for the web server are added to the server's class path.
- Stop the web server by running **shutdown.bat**.
- You may find it convenient to create desktop shortcuts to these two script files.

- Let's take a closer look at the filtering logic in **Examples\LevelsBlind\Step1**.
 - We'll see several flow-control actions used in concert to test a given member against the stated query criteria.
 - The page will be seen to implement the following algorithm:

```
if sex = parameter "sex",  
and age is within 3 years inclusive  
of parameter "age", then:  
{  
  loop over all words in the parameter "interests"  
  {  
    loop over all words in the member's interests  
    {  
      test for a match  
    }  
  }  
  if there was a match, then this member qualifies.  
}
```

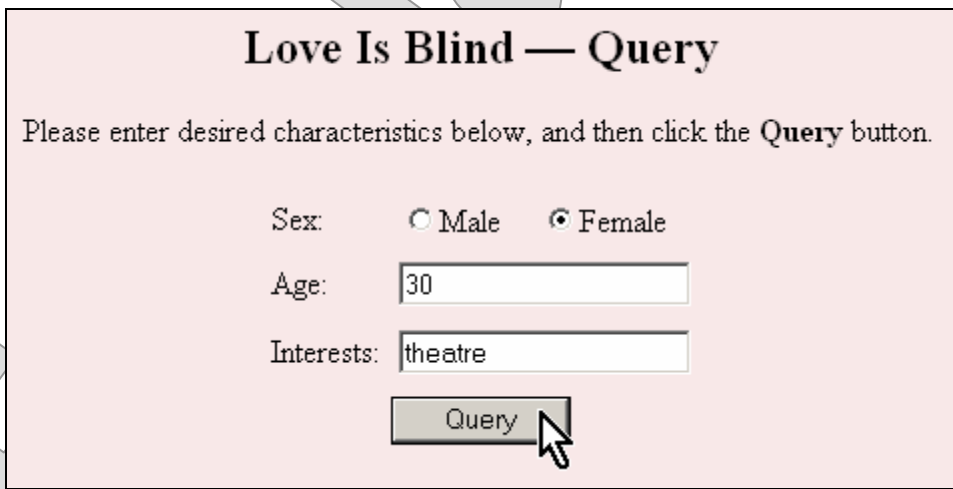
- This algorithm is implemented in `Tags\LevelsBlind\filter.tag`:

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="pres"
    tagdir="/WEB-INF/tags/LoveIsBlind/Table" %>

<c:if test="{sex == param.sex &&
    age + 4 > param.age && age - 4 < param.age}">
    <c:set var="match" value="false" />
    <c:forTokens var="ask" items="{param.interests}"
        delims=" " >
        <c:forTokens var="bid" items="{interests}"
            delims=" " >
            <c:if test="{ask == bid}">
                <c:set var="match" value="true" />
            </c:if>
        </c:forTokens>
    </c:forTokens>
    <c:if test="{match}" >
        <pres:member name="{name}"
            pseudonym="{pseudonym}" age="{age}"
            sex="{sex}" interests="{interests}" />
    </c:if>
</c:if>
```

- Test this example as follows:
 - Run the **update** script in the example directory. The **CC_MODULE** variable will be called into play now, as this script will copy **filter.tag** and other tag files to the module's **WEB-INF\tags** directory so that Tomcat can find them.
 - Start Tomcat with **startup**.
 - Start a web browser and navigate to the following URL:

`http://localhost:8080/JSTL/Examples/LoveIsBlind/Step1`



Love Is Blind — Query

Please enter desired characteristics below, and then click the **Query** button.

Sex: Male Female

Age:

Interests:

- Enter query criteria as shown above and click **Query**.
- The filter logic picks just two of the six listed members of the service, and these are listed in the table in **Results.jsp**:

<u>Name</u>	<u>Nickname</u>	<u>Age</u>	<u>Sex</u>	<u>Interests</u>
Alicia Grundel	Alicia	27	Female	music, painting, and theatre
Roberta Doss	Roberta	30	Female	cooking, wine, and theatre

Error Handling

- JSP error handling relies primarily on the **page** directive to specify an error page.
 - This will be served in lieu of the requested page if an exception is thrown during page processing.
- **This mechanism is coarse-grained, and for many purposes too “willing to fail.”**
 - The general attitude toward web page service is that it should be as error-tolerant as possible, providing page content if it can be managed, even if pieces are missing or incorrect due to errors, unavailable resources, etc.
 - Error pages don't fit all that well with this approach.

<c:catch>

```
<c:catch var = "static" >  
  content including actions that throw exceptions  
</c:catch>
```

- The `<c:catch>` action offers a means of error handling that is
 - Fine-grained
 - Able to recover from so-called “secondary processing” errors – that is, errors that should not compromise service of the requested page, even if some content will be missing or incorrect.
- Enclose a passage of JSP code that might produce exceptions – but whose success is not absolutely critical to page service – in a `<c:catch>` action.
 - The action will catch any exceptions that are thrown, preventing them from propagating to the JSP container.
 - Page processing can continue from the end tag of the catch action.
 - There is no “catch block” or “finally block” in which the author can include contingent actions – the error is always quietly discarded.
 - An object **var** can be exported – this will reference the exception that was caught, or will be **null** on success.
 - Successive code in the page can reference this object, and report it to the user or take other actions.

Buffering with `<c:set>`

- `<c:catch>` will not recover any response output created before an exception was thrown.
- That is, it is not **transactional**, but merely a robust way to keep page processing rolling.

– The following might produce an incomplete sentence:

```
<c:catch>
  The current enrollment is ${registrar.adults}
  adults and ${registrar.children}
  children.
</c:catch>
```

- A useful technique combines `<c:catch>` with `<c:set>`, using the latter as an **output buffer**.

– The buffer can then be flushed on success:

```
<c:catch>
  <c:set var="buffer" >
    The current enrollment is ${registrar.adults}
    adults and ${registrar.children}
    children.
  </c:set>
  ${buffer}
</c:catch>
```

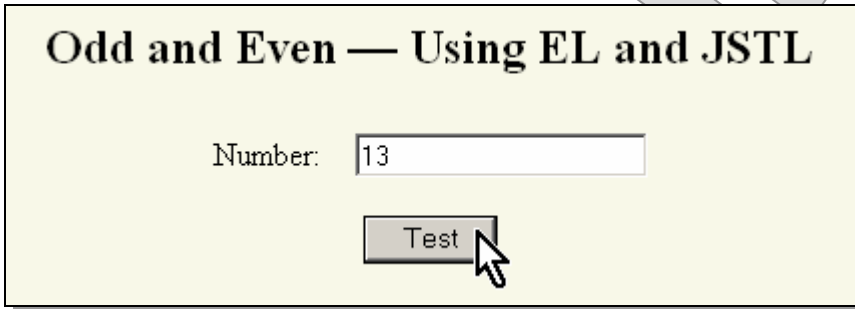
– If anything goes wrong, control will pass out of the `<c:catch>` action before the buffer is written to the response stream.

- Now we'll test out **Examples\OddEven\choose**.
- Note that **OddEven.jsp** has logic as discussed on the previous page:
 - It catches any exceptions in testing the number, and buffers output during that process.
 - It also checks the exception itself, and reports it:

```
<c:catch var="ex" >
  <c:set var="buffer">
    <p>
      ${param["number"]} is
      <c:choose>
        <c:when test="${param['number'] % 2 == 0}">
          even.
          <c:set var="totalEven" scope="session"
            value="${totalEven + 1}" />
        </c:when>
        <c:otherwise>
          odd.
          <c:set var="totalOdd" scope="session"
            value="${totalOdd + 1}" />
        </c:otherwise>
      </c:choose>
    </p>
  </c:set>
  ${buffer}
</c:catch>
<span style="font-size: 10pt; color: red;">
  ${ex}</span>
```

- Try out this page starting at

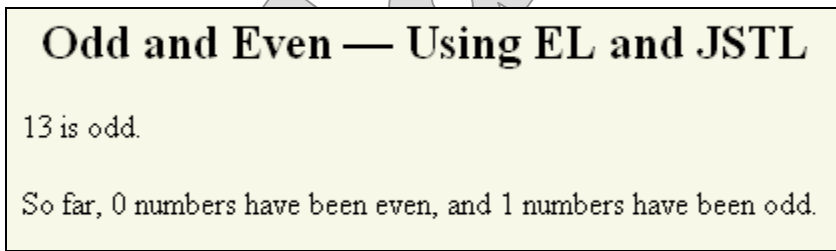
<http://localhost:8080/JSTL/Examples/OddEven/choose>



Odd and Even — Using EL and JSTL

Number:

- Enter a number and click **Test**.

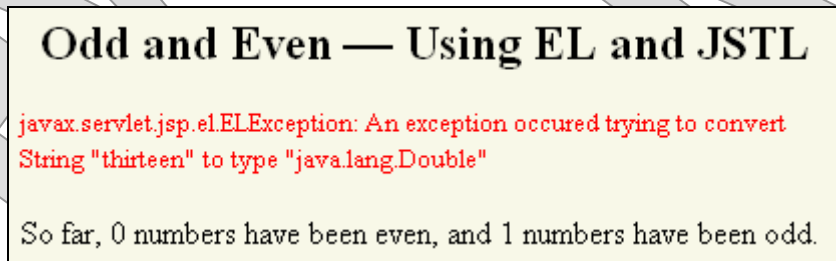


Odd and Even — Using EL and JSTL

13 is odd.

So far, 0 numbers have been even, and 1 numbers have been odd.

- This works fine; now go back and try entering something that can't be interpreted as a number.



Odd and Even — Using EL and JSTL

javax.servlet.jsp.el.ELException: An exception occurred trying to convert String "thirteen" to type "java.lang.Double"

So far, 0 numbers have been even, and 1 numbers have been odd.

- The `<c:catch>` action handled the exception, and reported it, and the rest of the page appears normally, including the current session's count of odds and evens.

<c:url>

```
<c:url
  var = "static"
  scope = "static"
  value = "dynamic"
  context = "dynamic"
>
  optional <c:param>s
</c:url>
```

- This action helps to form a URL for use in page logic.
 - This can be as simple as providing a string **value**.
 - Parameters can be specified either in the **value** or using **<c:param>** subtags – see the following page.
- URLs can be absolute, context-relative or page-relative.
 - Absolute URLs begin with a **scheme prefix** such as “http:”.
 - Context-relative URLs begin with a forward slash. Successive tokens will be evaluated from the JSP application context root – or from some foreign context as declared using the **context** attribute.
 - Page-relative URLs begin with alphanumeric tokens: directories or file names.
- URL objects resolve to strings, so that they can be used in producing HTML links:

```
<c:url var="link"
      value="Next.jsp?value=${computed}" />
<a href="${link}" >Click Here!</a>
```

<c:param>

```
<c:param
  name = "dynamic"
  value = "dynamic"
>
  value
</c:param>
```

- This action is invoked only within other actions, and defines a URL parameter as a name/value pair.
 - It functions much the way `<c:set>` does, except that it assigns a value to a **name** instead of to an exported **var**.
 - The value can be provided by the **value** attribute or by body content.

```
<c:url var="link" value="Next.jsp" >
  <c:param name="value" value="{calculated}" />
</c:url>
```

<c:import>

```
<c:import
  url = "dynamic"
  var = "static"
  scope = "static"
  varReader = "static"
  context = "dynamic"
  charEncoding = "dynamic"
>
  <c:param>s, or actions that process a Reader
</c:import>
```

- This action brings in an external resource for use in successive page logic.
 - The **url** is a string, which can be any well-formed URL, absolute or relative.
 - It can be encoded directly as a string, derived from an existing **<c:url>**-exported object, or take advantage of its own **<c:param>** subtags.
 - The action can export either a **var** as a string representation of the resource, or a **varReader** as a **Reader** for further processing, or both.
 - The **varReader** usage involves body content including actions that can consume information from the **Reader**.
- Imported resources can be produced as part of the JSP output, or can be fed into other page logic.

```
<c:import var="fragment" url="Snippet.html" />
<p>Some HTML here.</p>
${fragment}
<p>Some more HTML here.</p>
```

<c:redirect>

```
<c:redirect
  var = "static"
  scope = "static"
  url = "dynamic"
  context = "dynamic"
>
  <c:param>s
</c:redirect>
```

- This action redirects the container to an external resource and asks that resource to handle the current request.

In this lab you will complete the implementation of a JSP that calculates statistics on word usage in a text document. This will involve several stages of development: tokenize the document and build a map of word counts; remove commonly-occurring short words from the map; and produce the top ten words by occurrence counts in an HTML table.

Detailed instructions are found at the end of the chapter.

Suggested time: 60 minutes.

SUMMARY

- **The core library is aptly named; it provides fundamental actions that form the basis of practically all page logic that is based on custom actions.**
- **It fills gaps left by other JSP authoring styles:**
 - It can write where JSP expressions can only read information.
 - It can manage non-Bean objects, which standard actions cannot.
 - It provides a means of developing complex page logic in reusable tag files, which scripting cannot.
- **This is the most “horizontal” of the JSTL libraries, and perhaps of all JSP custom tag libraries.**
 - Core JSTs will be a part of nearly all page logic of any complexity.
 - More specialized actions will take advantage of exported objects, conditionals, and loops established by these core actions.