# LS/EECS Building Ecommerce Applications

## Lab 3: OsapCalc-v3

**(update by the deadline)**

## Objectives

-understand model view controller programming model
-understand forms, servlets, view generating tags
-learn to program Forms, Servlets, EL, JSTL

# Before You Start

- Make sure you have OsapCalc-v2 working correctly (forms, UI.jspx, etc..). For samples of forms, check the sample available on Moodle.
- Export your OsapCalc-v2 project (make sure you include the source files in the export) to a war file OsapCalc-v2.`war` on the desktop.
- Import the war file just created to a new project named OsapCalc-v3.

For this lab, you need to add a dependency to your project, jstl.jar. add (copy and paste) jstl.jar to Web-INF->lib folder in Eclipse (check Moodle page for instructions. Do it manually. Make sure you can run the sample from Moodle, before you start working on Lab 3.

Note: In production and large projects, dependencies are handled by "build" frameworks such as "maven." They resolve the dependencies automatically, that is they download and place the jar files into the right folder. However, at the same time they make the development environment more complex. For the time being, we manage the dependencies manually.

# The Persistence Layers (for reading, this was explained in lecture)

Servlets come with three scopes that allow you to store data at various levels and for various durations:

1. **The Context (Application) Scope** stores data for all clients as long as the server is running. You access it using the get/set attribute methods of the `ServletContext` object (accessible from the servlet).
2. **The Session Scope** stores data per client as long as the session of the client has not expired. You access it using the get/set attribute methods of the `HttpSession` object (accessible in `doGet/Post` from `request`). Read the API of request object to see what else can be done with it.

3. **The Request Scope** stores data per client as long as the response of the current request has not been sent yet. You access it using the get/set attribute methods of the `ServletRequest` object (accessible in `doGet/Post` from `request`).

Notes:

a. All three scopes use setAttribute (name, value) and getAttribute(name) methods to set and access the attributes
b. all three scopes use a key-value map to store these attributes, and hence, they are typed.
c. Always ask yourself where does it make sense to store an attribute, in request, session or context?
d. Attributes are different than "Parameters." Parameters are part of the query string that comes from clients, they are strings and can be accessed with "getParameter(name)" method of HTTP Request object.

# EL, JSTL (for reading, this was explained in lectures)

The content of jspx files is predominantly from the html namespace but we can also add elements from:

1. ***EL**, the Expression Language*
2. ***JSTL**, the Java Standard Tag Library*

**EL**

**EL elements** are characterized by the `${ }` syntax and may include expressions with operators. The expressions are made up of constants, local variables, attributes from the Request Scope, or *implicit objects* that provide access to various request properties, such as: `param, header, pageContext.request`

Here are some examples of EL expressions. There were more samples in the previous lab, including a sample JSP page with EL expressions.

```
2+2:  ${2 + 2}

Is 5 greater than 3?: ${5 gt 3}

22 divided by 7: ${22 div 7}

Here is how to get parameter 'num' from the request object:

${param['num']}

Here is how you get the attribute 'firstName' that was set in the
request object:

${requestScope['firstName']}
```

*Here is how you get the request object:*

```
${pageContext.request}
```

*Here is how to get the remote address:*

```
${pageContext.request.remoteAddr}
```

*Here is how to get the value of the "period" context parameter (from web.xml):*

```
${initParam['period']}
```

Here is how to get the value of the "email" attribute saved in the servlet Context:

*${email}*

Here is how to get the value of the "x" attribute saved in the session object:

*${sessionScope['x']}*

Here is how to get the value of the "y" attribute saved in the application context:

*${applicationScope['y']}*

**JSTL**

enables us to add xml tags that hide code behind it. To use such a tag you will need to have the namespace of its library to the top of the jspx file (check also the Moodle Lectures and Labs page for a sample of jspx with EL and JSTL):

```
  xmlns:c="http://java.sun.com/jsp/jstl/core"
```

This namespace is for the core library that contains basic constructs such as variable declaration, selection, and iteration. Here are some JSTL examples: *set, for, if, choose*

```
<c:set var="number" value="54" />

Your var number = ${number} <br/>

<c:set var="customSize" value="${param.num}" />

<span style="font-size: ${customSize}">
        Here is your css-set font
</span>
```

```
<c:forEach var="i" begin="1" end="20" step="3">
     I am looping... the loop counter i = ${i}
</c:forEach>

 Here is the "header" map (from request):

 <ul>
   <c:forEach var="element" items="${header}">
     <li>${element.key} -- ${element.value}</li>
   </c:forEach>
 </ul>

<c:if test="${not empty errorMsg}">
 <p style="color:red"> The error message is: ${errorMsg}</p>
</c:if>

<c:choose>
   <c:when test="${number eq 20}">
     <p>The number is big!</p>
   </c:when>
 <c:when test="${number eq 10}">
    <p>The number is medium size!</p>
 </c:when>
 <c:otherwise>
   <p>The number is small!</p>
 </c:otherwise>
 </c:choose>
```

Note that your `UI.jspx` must adhere strictly to xml. In particular, it may not contain any illegally named tag, such as the deprecated `<%` or `<@` which you may find on the web.

# Requirement 1. Adding Persistence to UI.jspx

This requirement is about adding persistence to your application. A user enters some data into your UI.jspx form, then submits it to the servlet; a result page, Result.jspx shows the monthly payment and the grace period interest; the user presses "Restart;" *the UI.jspx page should show the input fields populated with the values entered previously*. The user can change some of the inputs and resubmits.

Hints on how to do it:
-You have to persist the parameters "principal," "interest," etc. in one of the scopes. In which one do you store it, request, session or context? Why?
-You use setAttribute to store parameters. Remember that you can store any type in the attribute, so you can store them as Strings, Doubles, int, etc...
-if the servlet stored the user's entries into **one of scopes** then your JSP can easily access these attributes using EL. For simple attributes, access them using

${attributeName}. For array or complex types, check the samples of lecture and on moodle or the EL web page. Also, be aware of the precedence among the scopes (check the lecture)
-If the values of the field have been previously entered, you can show them in Ui.jspx by setting the **`value`** attribute of the form's **input field,** as below.

**<input type="text" ....value="${interest}"/>**

-For checkbox and radio inputs, you will need to an attribute: **`checked="checked"`** to show the input is selected. Hint: use the **`choose`** tag to show the radio buttons or checkboxes correctly checked.

Checked:

```
<input type="checkbox" id="grace" name="grace" checked="checked"></input>
```

Unchecked:

```
<input type="checkbox" id="grace" name="grace"></input>
```

Test your application by having two concurrent sessions(clients) accessing your application:
-Case 1: the sessions are from the same browser
-Case 2: the sessions are in different browsers

What do you observe? Can you explain the difference?

# Requirement 2. Separating and Improving the View

a)The results of the computations should be displayed on a result page that has the fields properly formatted.

b)The "legend" name ( which is also the application name) should be shared among the jsp pages such a way that if the application owner wants to change it, the change should happen in one place and preferably does not imply a change of the code.

How to do it:

-This step was supposed to be done in previous Lab.

- Create a page named `Result.jspx` and transfer the results of your computation, such as montlhy payment and grace period interest. Using the request Scope, you can now pass the computed payment from the servlet to it (see the slides from Lecture 2, both servlet and jsp).

- In order to access an attribute named `x` from a jspx page, use the expression language (EL) syntax: **`${x}`.**

-In this step you format the numbers in the result page as "currency". In order to format numbers, use a second JSTL library whose namespace is:

```
xmlns:f="http://java.sun.com/jsp/jstl/fmt"
```

-This is how root tag should look like after the jstl core and fmt libraries are added:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:c="http://java.sun.com/jsp/jstl/core"
xmlns:f="http://java.sun.com/jsp/jstl/fmt" version="2.0">
```

-To format the payment with a thousands separator and rounding to two decimals, write:

```
<f:formatNumber type="currency">${payment}</f:formatNumber>
```

-To share the legend name among jsp pages, you might want to consider using web.xml to store it as a context parameter, and then the servlet will set an attribute in the context scope.

# Requirement 3. Separate the Model and the Controller

a) To separate the concerns, you have to separate the "Model" from views and controller. There should be only ONE model for your application.

b) To improve the maintenance of the code and to allow for team development, refactor the Servlet so it only performs controller tasks.

How to do it:

-Here you create the model component of your application. The model is created as a Java package.

- Create an ordinary Java class in a package named `model` and name it `Loan`.
- Create a regular constructor for your class and define any needed constants as *static final attributes*.
- Add the following methods to your model, so it takes all the parameters you used to compute the monthly rate:

```
public double computePayment(String p, String a, String i, String
g, String gp, String fi) throws Exception

public double computeGraceInterest(String p, String gp, String
i, String fi) throws Exception
```

-The methods should throw an exception if any parameter is invalid. You can create custom exceptions or just use standard ones.

-Move all the code that does computation and validation into the Loan class.

-Create a package named `ctrl` and move your servlet to it. Use "init" method of the servlet *(See the JEE Servlet API following the Resources link in our course site)*:

```
public void init() throws ServletException
```

-This method is invoked only once when the server starts (it gets called from the servlet constructor).
-You override this method and instantiate the "model" (the Loan class) in it. This will make the model a singleton. Remember from the lecture that an application has ONE model and MANY views and controller. Store the "model" reference in one of the scopes (which one makes sense?). Note the exception that this method throws. If the model's constructor throws any other exception (i.e. one that is not equal to, or not a subclass of, this exception) then your code should catch it and re-throw it with proper attribution.
Here is an example…where the model is saved in the context..

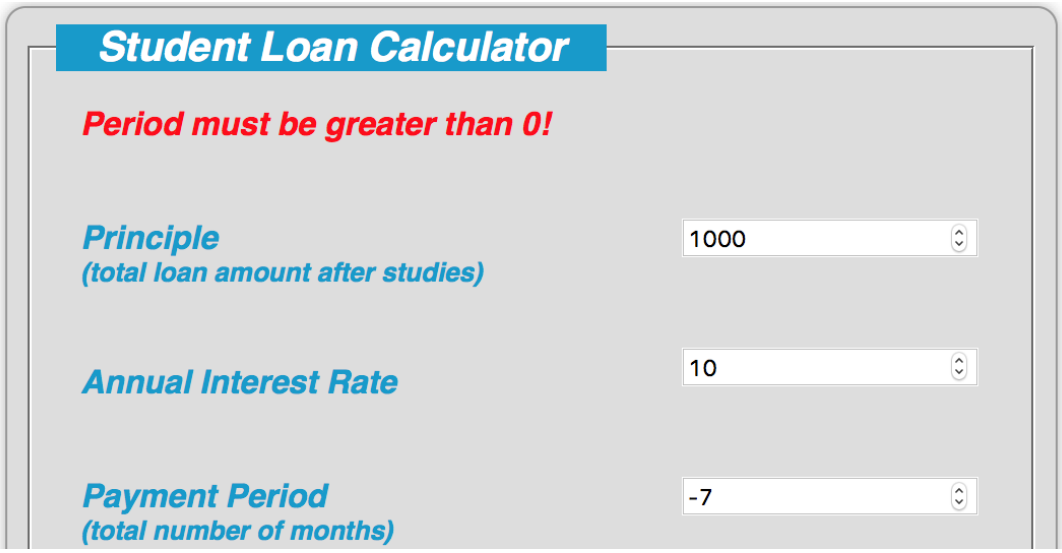
```
@Override
        public void init(ServletConfig config) throws ServletException {
                super.init(config);
                ServletContext context = getServletContext();

                appName = context.getInitParameter(CONTEXT_APP_NAME);

                fixedInterest = context
                                .getInitParameter(CONTEXT_FIXEDINTEREST);

                // Instantiate Loan object
…….
                context.setAttribute("mLoan", new Loan());
        }
```

- As before, your servlet starts by distinguishing a fresh visit from a submission visit and further from a re-compute visit.
- Your `doGet` and `doPost` methods must obtain **the model's reference** from the scope where it was saved.
- Remove all computation code from your servlet and replace it with an invocation of the model's methods.
- If all is well, forward to Result.jspx. If errors, see next task.

# Requirement 4. Handling errors

Handle any input errors and any calculation errors. If the user enters an input which is not valid you have to detect it and report it back to the user. For example, if the user enters a negative principal, you should inform the user that the principal should be greater than 0. You have to think of any invalid input that

affects your calculations. In case of an error, the UI page must show the error message in red at the top, and must pre-fill the form with whatever the user entered in the previous submission. See the example below.




How to do it:
-Verify the parameters for errors (do it in the model)
-If there is an error detected, set an attribute in one of the scopes and redirect the control to UI. Which scope do you use?
-Use JSTL **choose** or **when** in the UI.jspx to test if there is an error and then display the error.