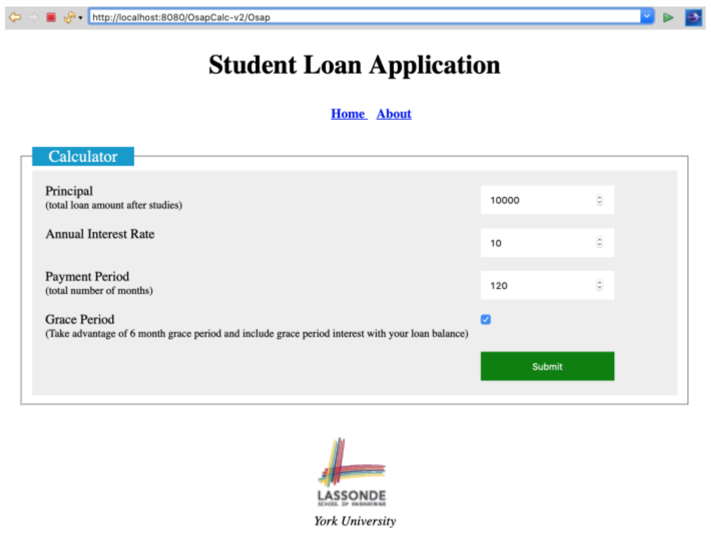
LS/EECS Building Ecommerce Applications

Lab 2: OsapCalc-v2.0

(upload before the deadline)

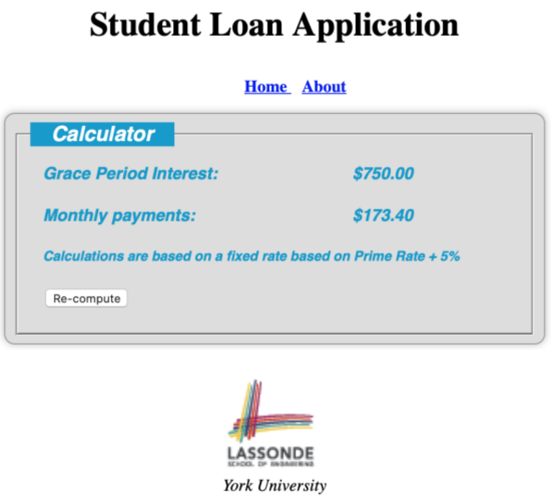
Objectives

-understand forms, servlets, jsp
-learn to program forms, servlets, jsp, css
-create an application that has these 2 main UIs and use cases described by the tasks below.



http://localhost:8080/OsapCalc-v2/Osap
Student Loan Application
Home About
Calculator
Principal
(total loan amount after studies)
10000
Annual Interest Rate
10
Payment Period
(total number of months)
120
Grace Period
(Take advantage of 6 month grace period and include grace period interest with your loan balance)
Submit
LASSONDE
York University

Fig. 1. UI.jspx page, at the end of the lab



Student Loan Application
Home About
Calculator
Grace Period Interest: $750.00
Monthly payments: $173.40
Calculations are based on a fixed rate based on Prime Rate + 5%
Re-compute
LASSONDE
York University

Fig. 2. Results.jpx page at the end of your Lab 2.

Tasks

Before You Start this Lab

- Make sure your `OsapCalc-v1` project is working correctly. It should allow the client to provide the needed parameters through the query string and it should return the monthly payment. No need for any validation (such as principal is a positive number) or presentation formatting (beyond the rounding) at this stage. Do *not* proceed with the next step until you tested this functionality.
- Export your `OsapCalc-v1` project (make sure you include the source files in the export) to a war file `OsapCalc-v1.war` on the desktop. Submit it through Moodle, if you have not done so.
- Import the war file just created to a new project named `OsapCalc-v2`.
- Clean up your *Osap* servlet so it only contains parameter extraction, payment computation, and response output. We don't need any code related to networking explorations.

Task A: HTML Forms

HTML forms enable the user to input data to your application. It is highly recommended that you create your first form in another project to get a feel for it. Use the samples from the lectures as a starting point. It is important you type each line, do not use copy and paste…in this way you better learn the syntax

- Right-click the `WebContent` folder of `studentCalc1` to create a new *jsp* file. Name the file `UI.jspx` and select its template so it complies with the latest JSP tags (2.0).
- Note that `jspx` files are just `html` files with two extras: strictness (they follow to xml, hence "x") and richness (thanks to JSTL). For now, think of it as ordinary `html`.
- Organize your page following the typical HTML content, that is make sure you have the following elements
    - Header
    - Nav
    - Section
    - Article
    - Footer
    - Aside

  (check the course resources and Fig 1. and 2)

- Create a form to take the inputs for our webapp. If you have not created html forms before, you may want to look at course resources but the key features are covered in the bullets below (and in the lecture sample). Look at the picture above for how your form should look like in term of text (Note: at this stage, do not pay attention to style, we will address that in the css section)

- The `form` tag has `action` and `method` as attributes. Set the first to the empty string and the second to `GET`.
- Use a **`label`** tag for prompts and an `input` tag for user inputs.
- The `input` tag for text fields must have a `type` attribute with value `text`, a `value` attribute for the default (pre-filled) value, and a `name` attribute for query string construction.
- The `input` tag for the submit button must have a `type` attribute with value `submit`, a `value` attribute for the button's caption, and a `name` attribute for query string construction.
- Note that *all* html tags take an optional `id` attribute or/ad a *class* attribute. Do not confuse `id` (which is used to facilitate the addressing of the tag from within the html document and its stylesheets and scripts) with `name` (which is used for assembling the needed elements of the query string upon form submission). These attributes id, class, name serve different purposes.
- In order to pair a `label` with its corresponding `input` field, add a `for` attribute to it and have it point to the `id` of the input field tag. Example

```
<form action="Osap" method="GET" class="osapForm">
   <label for="principal"> Principal <BR /> <small>(total loan amount after studies)</small></label>
  <input type="number" step="0.01" id="principal" name="principal"></input>
…….
  <button action="submit" name="calculate" value="true">Submit</button>
</form>
```

Complete the form so it has all elements from Fig. 1

- HTML (and particularly html5) has a rich set of form tags in addition to the basic ones mentioned above.
- To test your work incrementally as you build your form, simply select UI. jspx and "Run on Server"
- If everything works fine, when you press the button "Submit" the browser will invoke the servlet "Osap" and its method "GET" according to the attributes in the above form. your servlet will return the same content as in Lab 1.
- Make sure you have this functionality working.

Task B: Serving the Form

- Now the question is how does the servlet retrieves the data from the form?
  - Simply, by using request.getParameter("name"), where "name" is the name of the *input* field, e.g. "calculate" , "principal" in the form above.
  - Try to get those parameters and return them in the servlet response (as a testing step)
- Modify your `Osap` servlet so it serves the UI form when it starts.

Hint: in doGet method, use RequestDipatcher object to switch data and control to another artifact, e.g. to UI.jspx..see below

```
String target = "/UI.jspx";
request.getRequestDispatcher(target).forward(request, response);
```

Verify that your client will see the form upon visiting by selecting the servlet and then "Run on Server." You can also use the url, http://localhost:8080/OsapCalc-v2/Osap, which, according to the web.xml will invoke the servlet "Osap"

- Add code at the top of your servlet to distinguish a fresh visit (to which you respond by serving the form) and a submission visit (to which you respond by computing and sending the payment). Hint: the name of the submit button, "calculate" , is sent as part of the query string when the servlet is invoked by the form.
- To serve the computed payment, do the same thing you did in the previous release (Lab1); i.e. get a writer and output an html fragment containing a brief message and the rounded amount.

Task C: Changing the Computation

- Add a new parameter `fixedInterest` within our context (that is the web.xml file) and set it equal to `5`, this new parameter is the set interest that gets added to the prime interest rate that has been supplied by the user. i.e. if the user supplied 3.5 as the interest, the total interest becomes `User Supplied Interest + fixedInterest`
- Add a new parameter `grace`, this new parameter indicates whether grace period is accounted for in the monthly payments
- Add a new context parameter (in web.xml file) `gracePeriod` and set it to 6, this defines the standard grace period in months
- Add a computation for the `grace interest`, the formula is as follows: `Grace Interest = Principal * ((Interest + fixedInterest) / 12) * gracePeriod`
- Our new monthly payment formula is as follows `Monthly Payment Formula = Monthly Payment Formula + (graceInterest / gracePeriod)`. Note: if you do not know the monthly payment formula, refer to the previous Lab.

Task D: Form Submission - POST

- In the form, change the value of the `method` attribute to `POST`. Now the parameters are not sent in the URL query anymore
- In the doPost() method on the servlet, just call doGet(request, response)
- Reload the form in your browser and activate its network monitoring tool in a persistent mode (i.e. so it does not clear the log after a form submission).
- Insert some data and click the submit button. The page will reload this time but you should see the POST request and the associated content that was uploaded to the server.

- In addition to the uploaded payload, note also the `Content-Length` and `Content-Type` request headers. What does the length measure? Verify its value.
- In general, what are the advantages and disadvantages of using POST versus GET?
- If everything works fine, you should have the same functionality as at the end of Task D.

Task E: Adding Results.jspx page

- Writing html tags from java code in the doGet method is not the correct. We have done it for testing purposes.It is time to separate the java code from the view.
- Create a new page, Results.jspx and add the header, footer, a Nav bar and a button "Re-compute" The button should invoke the Servlet Osap, and its method POST.
- Test it..
- In Osap, servlet, in doGet method, after you finish the computation, use RequestDispatcher to invoke "/Results.jspx"
- `String resultPage = "/Results.jspx";`
- `request.getRequestDispatcher(resultPage).forward(request, response);`
- The question is how do you get the data from servlet into the results page
    - In the servlet, save all data you want in the page into the session object, something request.getSession().setAttribute("`GI`", `graceInterest`) method
    - In the jspx page, use EL language expressions, such as ${GI} to get the value of the attribute named "GI"

```
 <form action="" method="POST" >
   <strong>Grace Period Interest: </strong> ${graceInterest}<br/>
    <strong>Monthly payments: </strong> ${payment}<br/>
    button action="restart" name="restart" value="true">Re-compute</button
</form>
```

Task G: Cascading Style Sheets

- WHAT: Separating Semantics from Presentation.
- WHY: Scalability; customization; consistency; empowerment
- How: Two separate documents: html & css. Link the two by putting this link tag in the head of the html file:

```
    <link rel="stylesheet" type="text/css" href="..." title="..."
media="..." />
```

  Note that media can be all, screen, print, handheld, aural, braille, tv, projection, etc. The title is used if multiple style sheets are linked.

- Style Rule Syntax: The css file contains one or more style rule. Each rule has a *selector* followed by one or more *declaration* surrounded by braces. The declaration syntax is: property: value followed by ;.
- Selector Syntax: Can be a tag name, an id (prefixed with #), a class name (prefixed with .), or a pseudo class name (prefixed with :). You can also combine these building blocks by a comma (implying the rule apply to several selections), by a space (implying hierarchy), or by a dot (tag.class).

Incorporating Styles in this Release

- Add the following to the `head` section of `UI.jspx`:

```
<link rel="StyleSheet"
        href="${pageContext.request.contextPath}/res/mc.css" type="text/css"
title="cse4413" media="screen, print" />
```

- Create a folder under `WebContent` of your project and name it `res` (for resources).

- Right-click the `res` sub-folder of WebContent and select a new css file and name it *mc.css*.
- Start adding rules and checking the browser after each. Don't just copy and paste a whole set of rules; you will understand better and a deeper level if you do it one rule at a time. Here is one sample rule:

```
.osapForm {
  display: grid;
  grid-template-columns: [labels] auto [controls] 1fr [info] 1fr;
  grid-auto-flow: row;
  grid-gap: 0.8em;
  background: #eee;
  padding: 1.0em;}
```

It says that the children of an element that has an attribute class='osapForm' are displayed in a grid with 3 columns. The columns and their width are: *labels-auto, controls-1fraction from the rest*, *info* - *1fraction*;

Another rule, this time for a child of osapForm class:

```
.osapForm > label {
  grid-column: labels;
```

```
   grid-row: auto;

 }
```

The child with the tag ‘label’ is placed under the column, “labels” of the grid. The row is determined automatically (auto).

- Here are highlights of other rules you may want to add:

```
@CHARSET "UTF-8";


article
{
    padding:5px;
    margin-top:5px;
}

header
{
    padding:16px;
    text-align:center;
    font-weight:bold;
    font-size:2.0rem;
}

aside
{
    margin-top:5px;
    background-color:white;
    padding:5px;
    text-align:center;
    font-style:italic;
}

section
{
    padding:5px;
    margin-top:5px;
}

footer
{
    padding:5px;
    text-align:center;
    font-weight:bold;
}

nav
{
    text-align:center;
}

ul li
{
    display:inline;
    padding-left:5px;
    padding-right:5px;
    text-align:left;
    font-size:16px;
    font-weight:bold;
}

legend
  {
    font-size: 125%;
    padding-left: 1em;
    padding-right: 1em;
    background: #09C;
    color: #fff;
  }


  .resultForm
  {
    width:500px;
    margin:auto;
    background-color: #dddddd;
    font-family: Calibri, Ariel, sans-serif;
    font-size: 16px;
    font-style: italic;
    font-weight: bold;
    color: #09C;
    border-radius: 10px;
    padding: 8px;
    border: 1px solid #999;
    border: inset 1px solid #333;
    box-shadow: 0px 0px 8px rgba(0, 0, 0, 0.3);
  }

  tr, td {
      padding: 10px;
  }


 .error{
  color:red;
  font-weight: bold;
  }


  .osapForm {
    display: grid;
    grid-template-columns: [labels] auto [controls] 1fr [info] 1fr;
    grid-auto-flow: row;
    grid-gap: 0.8em;
    background: #eee;
    padding: 1.0em;
  }

  .osapForm > label  {
    grid-column: labels;
    grid-row: auto;
  }
```

```
.osapForm > .error{
grid-column:info;
grid-row:auto;
color:red;
font-weight: bold;
}


.osapForm > button {
  background: green;
  color: white;
  }

.osapForm > input,
.osapForm > button {
  grid-column: controls;
  grid-row: auto;
  border: none;
  padding: 1em;
}

.radio label, .radio input
{
  display: inline;
}
```