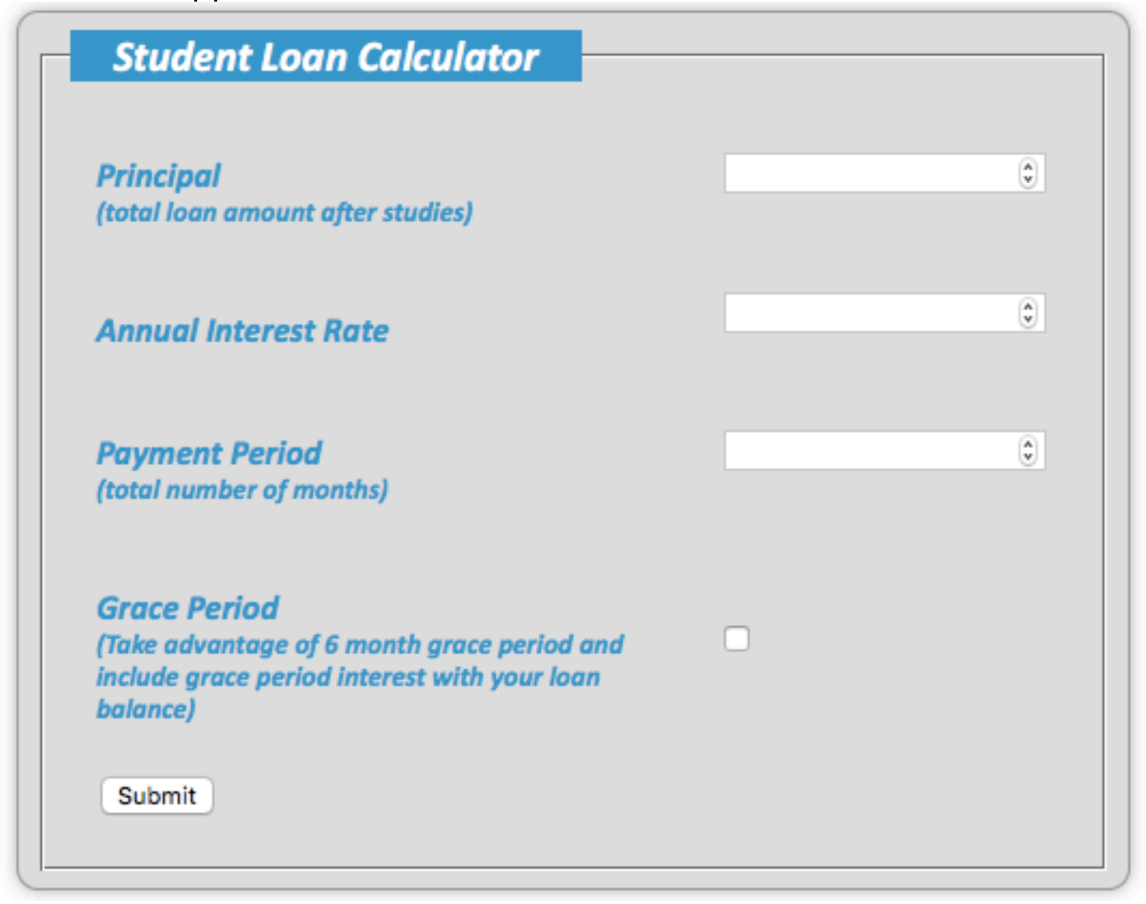
LS/EECS Building Ecommerce Applications

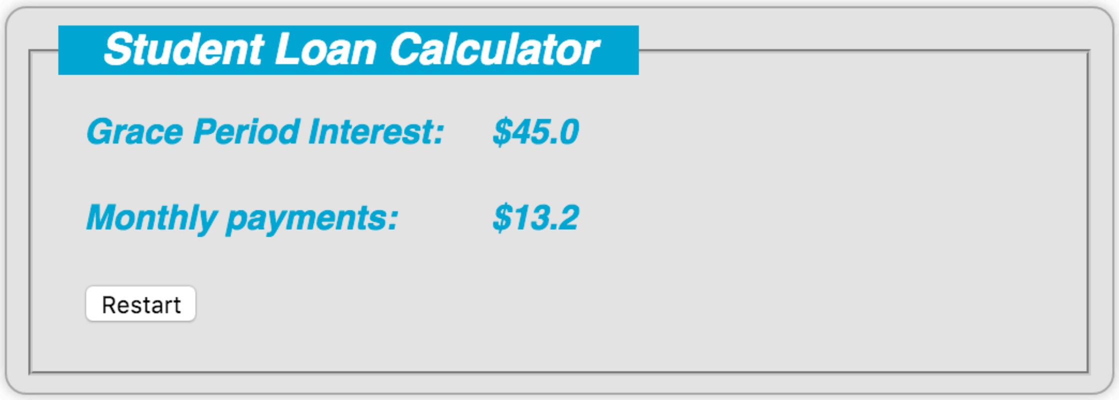
Lab 2: StudentCalc R2.0

(show your implementation to the TA at beginning of Lab 3)

Objectives

-understand forms, servlets, jsp
-learn to program forms, servlets, jsp, css
-create an application that has these 2 main UIs and use cases


Student Loan Calculator
Principal
(total loan amount after studies)
Annual Interest Rate
Payment Period
(total number of months)
Grace Period
(Take advantage of 6 month grace period and include grace period interest with your loan balance)
Submit



Tasks

Before You Start this Lab

- Make sure your `studentCalc1 (release 1)` 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 persistence support (such as re-computing with a different interest rate) or 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 `studentCalc1` project (make sure you include the source files in the export) to a war file `studentCalc1.war` on the desktop.
- Import the war file just created to a new project named *studentCalc2*.
- Clean up your *Start* 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

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`.
- 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 lectures). 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)

- 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. 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).
- 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

```
<tr>
    <td><label for="interest">Annual Interest Rate</label></td>
    <td><input type="number"  step="0.01" id="interest" name="interest"></input></td>
</tr>
```

- 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 visit it from your browser. Since it is stored in the root of your webapp, you visit it via the URL: `http://host:port/studentCalc2/UI.jspx`.

Task B: Serving the Form

- The jspx file will be translated into a Java servlet and compiled and deployed just in time upon the first visit. Hence, think of it as a servlet.
- Modify your `Start` servlet so it serves the UI form when it starts. Hint: forward to it. See the forward method in the lecture samples..
- Verify that your client will see the form upon visiting `http://host:port/studentCalc2/Start`.
- Activate the http monitoring tool of your browser and enter some input data in the form. Click the submit button. Verify that the browser has correctly built and sent the query string.
- Add a **`legend`** tag as the first child of `form` and type in *Student Loan Application* between the opening and closing tags (look at the samples in the lecture).
- Reload the form in your browser and examine the change.
- Surround all your label and input fields with a **`fieldset`** tag (again, see the lecture).
- Reload the form in your browser and examine the change.

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: The Server Side

- Note that setting the `action` attribute to an empty string means the browser must submit the form to the same server from which the form was retrieved.
- 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 is sent as part of the query string.
- 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.
- Add a button with caption *Restart* to enable the user to return to the main form with all its fields reset to their default (empty) values. Note that the button should be named in such a way that the controlling servlet will handle requests coming from clicking it as if they were fresh visits.
- Write a brief sentence to explain why it is awkward to intermix html output and Java code.
- Write a brief paragraph to explain why mixing validation, payment computation, and presentation violates the separation of concern principal.

Task E: Form Submission - POST

- Change the value of the `method` attribute to `POST` and change the empty `action` attribute to `http://www.cse.yorku.ca/`
- 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?

Task F: The Server Side - POST

- Keep the form's `method` attribute set to `POST` but set its `action` attribute to the empty string.
- Keep everything else the same in your servlet.
- Test your webapp.
- You will notice that you can get the form from your browser, as before, but the submit button does not display the payment. Why?
- Use your browser's http monitor to see if the button did indeed post the parameters to the server.
- If the client is behaving correctly then it must be the server. Use `System.out.println` to see where is the problem in your servlet.
- Once you have identified the problem, fix it and test. The POST version should produce the same result as the GET version.
- Again, but this time on the server side, what are the advantages and disadvantages of using POST versus GET?

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="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:

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

- Visit the form and examine the changes.
- Here are highlights of other rules you may want to add:

```
  legend   {
      font-size: 125%;
      padding-left: 1em;
      padding-right: 1em;
      background: #09C;
      color: #fff;   }
  form   {
       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;
  }
  label {
    display: block;
    margin-top: 16px;
  }
  .radio label, .radio input    {
  display: inline;
}
```

- You will need to add ID's and/or classes to `UI` so that the css selectors can be correctly linked with their targets.