Chapter 4

Working with Forms

OBJECTIVES

After completing "Working with Forms," you will be able to:

- **Build HTML forms into your JSPs.**
- **Process form input in your servlets:**
  - **Single-value** inputs such as text fields
  - **Multi-value** inputs such as from multi-select option lists
  - **Checkbox** and **radio-button** inputs
  - **Submit buttons** themselves, for example to distinguish between multiple buttons on a form and to know which one was clicked
- **Validate form input before applying it to the application's model.**

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

HTML Forms

- **In the previous chapter we emphasized the need for dynamic applications – which simply means those that act differently from one use to the next.**
- **It's possible to have a dynamic application that does not vary based on user input – only on external or server-side stimuli.**
  - DateTime is a small example of this; a news site would be another.
- **But most applications will want to accept user input in some form, and the most common form is, well ... the form!**
- **The HTML `<form>` element tells the browser what it needs to know in order to submit user-entered data to the website.**

```
<form action="MyNextServlet" method="post" >
```

  - The **action** attribute is a URL to which a request will be sent when the user "submits" the form.
  - The **method** chooses the HTTP method for the request.
- **Descendant elements of the `<form>` tell the browser to render various sorts of user-interface "widgets" from text fields to lists to pushbuttons.**
- **Many of these components are capable of submitting the form, but the most common means of doing so is via a submit button:**

```
<input type="submit" name="OK" value="OK" />
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Form Parameters

- **When a form is submitted, input components in the form contribute their data to the resulting request as form parameters.**
  - Each component has a **name** attribute, and this is used as the name of a form parameter.
  - And each component has a value, which is sometimes expressed initially as a **value** attribute, and sometimes in other ways including character content and child elements, but which ultimately reflects user actions since the page was rendered.
- **Different sorts of components use this basic name-value system in different ways.**
  - **Single-value** components, such as text fields, are the most obvious: there will be a single form parameter in the submission.
  - **Single-selection** components such as lists and sets of radio buttons, result in single form parameters as well.
  - **Multi-value** components, such as lists with multiple-selection features, will do something perhaps a little surprising: there will be zero to many parameters, all with the same name.
  - **Checkboxes** will submit single parameters with the value "on" – or they will not submit anything at all!
  - **Submit buttons** will provide single parameters, too. They have meaningful names, and their values are their visible captions – not changeable by the user, but values nonetheless.
  - This is more useful that it might seem at first: you can look for the presence of a parameter name as a clue that the user clicked one button rather than another.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Form Submit Method

- We've seen a bit of the **query string** that can appear in an HTTP URL, carrying some number of name-value pairs to the server.
- If a `<form>` has a **method** of "get" (this is case-insensitive, by the way, but usually expressed in lower case), then a query string will be synthesized, just like the ones we've seen, and appended to the **action** URL to inform an HTTP GET request.
  - Certain special characters in the form data will be **escaped** to characters that are legal as part of a URL.
- For a **method** of "post", there is still a query string.
  - But it is carried in the **body** of an HTTP POST.
- The servlets API – and JSP expressions using the **param** object – will read the parameters the same way regardless of method.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Form Submit Method

- **POST is generally preferred for form submission, for a few reasons:**
  - GET is meant to be **idempotent** – a term which is somewhat misinterpreted in the HTTP specification to mean that it should be possible to repeat the request endlessly with equivalent results – all around – to sending it just once.
  - POST is understood to be capable of changing application state – and is not expected to be repeatable. Browsers often warn users when they try to reload the result of a POST, or go back through a POSTed URL, etc.
  - So it's largely a question of conventions and expectations.
  - Also GET carries the form data in the URL, which can have significant security implications since the full URL – query string and all – will thenceforth be available in the browser's history, can be auto-completed, bookmarked, etc.
- **Oddly, though, GET is the default!**
- **So it's important to remember the method attribute if you do indeed want to get an HTTP POST when the user submits the form.**

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

GET vs. POST

DEMO

- **We'll now demonstrate some of the differences between GET and POST in form submission, by modifying our Hello application from an earlier chapter.**
  - Do your work in **Hello_Step4**.
  - The completed demo is in **Hello_Step5**.

1. Open **docroot/index.html**.
   - This is the home page for the application, for which we didn't have much use in earlier labs as we were experimenting with various hand-written request URLs.
2. Add two HTML forms to the existing content, as shown below:

```
<div style="padding-left: .4in;" >
  <p><a href="Servlet" >/Servlet</a></p>
  <p><a href="Servlet/Birthday" >
    /Servlet/Birthday</a></p>
  <p><a href="Servlet/Valentine" >
    /Servlet/Valentine</a></p>
  <p><a href="Servlet?name=Sonia" >
    /Servlet?name=Sonia</a></p>
  <form action="Servlet" >
    <input type="text" name="name"
      value="Frannie" />
    <input type="submit" name="button"
      value="GET IT" />
  </form>
  <form action="Servlet" method="post" >
    <input type="text" name="name" value="Zooey" />
    <input type="submit" name="button"
      value="POST IT" />
  </form>
</div>
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

GET vs. POST

DEMO

3. Open **src/cc/hello/Servlet.java**, and rename the **doGet** method to **handleRequest**. (And, it should no longer be an **@Override**.)
4. Now create a new **doGet** method, and have it call **handleRequest**.

```
@Override
protected void doGet (HttpServletRequest request,
    HttpServletResponse response)
  throws IOException, ServletException
{
  handleRequest (request, response);
}
```

5. Create a **doPost** that does the same thing:

```
@Override
protected void doPost (HttpServletRequest request,
    HttpServletResponse response)
  throws IOException, ServletException
{
  handleRequest (request, response);
}
```

- **So now you have a servlet that responds in the same way to either a GET or a POST – not a good practice generally, but useful for our purposes here.**
- **And you can trigger that servlet via two different methods of form submission.**

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

GET vs. POST

DEMO

6. Run the HTTPSniffer – again, you can just type **start sniff** from the command line, or run the HTTPSniffer configuration in Eclipse.
7. Build and test the application at a "sniffable" URL:

```
http://localhost:8079/Hello
```

Hello, Servlets!

Try the following links to make requests of the **cc.hello.Servlet**:

/Servlet

/Servlet/Birthday

/Servlet/Valentine

/Servlet?name=Sonia

Frannie GET IT

Zooey POST IT

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

GET vs. POST

DEMO

8. Click the **GET IT** button, and see the familiar response page – hmm, well, mostly familiar:

Hello there, Frannie&button!

request.getMethod():	GET
request.getContentLength():	-1
request.getContentType():	null
request.getRequestURI():	/Hello/Servlet
request.getContextPath():	/Hello
request.getServletPath():	/Servlet
request.getQueryString():	name=Frannie&button=GET+IT

– In the sniffer you see the query string in the URL, as expected:

```
GET /Hello/Servlet?name=Frannie&button=GET+IT
  HTTP/1.1
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

GET vs. POST

DEMO

9. Go back and try the **POST IT** button.
   - Again, something's not quite right in the response page. The title is supposed to be "Hello there, Zooey!"

Hello from the Servlet itself!	
request.getMethod():	POST
request.getContentLength():	25
request.getContentType():	application/x-www-form-urlencoded
request.getRequestURI():	/Hello/Servlet
request.getContextPath():	/Hello
request.getServletPath():	/Servlet
request.getQueryString():	null

   - Now in the sniffer you see the information in the POST body:

```
POST /Hello/Servlet HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 25

name=Zooey&button=POST+IT
```

   - And notice that **getContentLength** and **getContentType** are now showing interesting values, because the request has a body.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

GET vs. POST

DEMO

- **So, most of this is as it should be. But, what's wrong with our request processing that we're getting these weird titles?**

10. Look in **handleRequest** and see a couple issues:
    - Our test for the presence of a query string doesn't hold up for an HTTP POST – this explains why the POSTed name is ignored and the page title is "Hello from the Servlet itself!"

```
if (queryString != null)
```

    - Then, even for a GET, our parsing for request parameters is sloppy, and breaks down because there are more than one parameter in the string – and this is why we get "Frannie&button" as the name:

```
  String name = queryString.split ("=")[1];
```

11. Fortunately there are much more robust ways to parse user input, that work for GET and POST and for any number of parameters! Fix the code as shown below:

```
if (request.getParameter ("name") != null)
{
  String name = request.getParameter ("name");
```

12. Build and test again, and you should see good results all around.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Geometry

LAB 4A

Suggested time: 15 minutes

In this lab you will implement a servlet that reads user inputs through an HTML form that describe the dimensions of a three-dimensional ellipsoid. The servlet will set these values into a JavaBean which knows how to calculate values about the ellipsoid – volume, type, and description – and will pass that bean along to a JSP which presents a full report.

Detailed instructions are found at the end of the chapter.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Reading List Selections

- A <select>/<option> list in an HTML form will contribute to the HTTP submission in one of two ways:

```
<select name="level" >
  <option value="1" >One</option>
  <option value="2" >Two</option>
  <option value="3" >Three</option>
</select>
```

  - If **single-select**, it will act much like a text field, providing a single name and value:

```
level=2
```

  - If **multi-select**, it will provide zero to many values, all under the same name:

```
level=1&level=3
```

- To process a multi-value list, you must do more than just call **getParameter** on the request object; this method assumes a single value and will only return a single value.
- There is a separate method **getParameterValues**, which takes a single name but returns an array of strings.

```
String[] selections =
  request.getParameterValues ("level");
```

  - Note that this approach is entirely safe for single-value inputs as well; it's just not usually as convenient as **getParameter**.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Reading Checkboxes and Radio Buttons

- A checkbox control is just an input of type “checkbox”:

```
<input type="checkbox" name="mailingList" >
  Send me email!</input>
```

- A checkbox will send a value of “on” when checked.

```
mailingList=on
```

- When unchecked, it will not contribute to the request at all.
- This takes some getting used to!
  - You can test for the value “on” by name.
  - You can test for the simple presence of the name, and disregard the value.
  - You can’t test for the value “off” – it won’t be there!
- Only one radio button of the same name can be selected, so a group of them will act as a single-value component.

```
<input type="radio" name="level" value="1" >
  One</input>
<input type="radio" name="level" value="2" >
  Two</input>
<input type="radio" name="level" value="3" >
  Three</input>
```

  - The input here will be either nothing (no buttons selected) or a single parameter:

```
level=2
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Reading Submit Buttons

- **Submit buttons are often ignored when processing form input, because their purpose was just to trigger the submission.**
- **But it can be useful to know which of several submit buttons in a form was clicked.**
- **Submit buttons have names and values – but the values are used as user-readable button labels, so they are often not interesting on the server side.**
- **This leaves two strategies:**
  - Give all buttons the same name, and look at the value to determine which was clicked:

```
<input type="submit" name="cmd" value="OK" />
<input type="submit" name="cmd" value="Cancel" />

if (request.getParameter ("cmd").equals ("OK")) ...
```

  - Give each button a different name, and test for the presence or absence of the name in the list returned by **getParameterNames**:

```
<input type="submit" name="OK" value="Oh-kay!" />
<input type="submit" name="cancel" value="Nope" />

List parameterNames = Collections.list
  ((Enumeration) request.getParameterNames());
if (parameterNames.contains ("OK")) ...
```

  - You did something like this in the previous chapter's lab, with more use of type parameters for better type safety.
  - This approach involves a bit more code, but allows clean separation of user-friendly captions and logical values.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

UIComponents

EXAMPLE

- In **UIComponents** we exercise the primary control types in an HTML form, and process their inputs in a servlet.
- See **docroot/index.html**:
  - There is a form that POSTs to a servlet at the URL "Submit":

```
<form method="post" action="Submit" >
```

  - Then each component type is illustrated in a table row with a descriptive label:

```
<table>
  <tr>
    <td class="code" >Text</td>
    <td><input type="text" name="text" /></td>
  </tr>
  ...
```

  - And since it won't be obvious in testing the way the others will, take an extra look at the "hidden" input component:

```
  <tr>
    <td class="code" >Hidden</td>
    <td><input type="hidden" name="hidden"
          value="special code" /></td>
  </tr>
```

  - Another piece we've not seen by example yet is a multi-select list:

```
<select name="fruit" size="3"
        multiple="true" >
  <option value="1" >Apple</option>
  <option value="2" >Orange</option>
  <option value="3" >Banana</option>
</select>
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

UIComponents

EXAMPLE

- The submit button has the name "bn":

```
<input type="submit" class="button" name="bn"
  value="Submit" />
```

- And, another new trick, here's a link that sends an HTTP GET to the same URL as the form:

```
<a href="Submit?text=Hello&password=guest&
  hidden=secret&flag=on&dayOrNight=night&
  direction=3&fruit=1&fruit=3" >LINK</a>
```

- See **src/cc/servlets/ReadData.java:**
  - We'll see in a moment that a JSP will be able to echo all the single-value inputs by itself.
  - But the multi-select list requires some special attention, so the servlet's **doPost** method reads that data and shapes it up in a JSP-friendly form:

```
String[] values =
  request.getParameterValues ("fruit");
StringBuilder fruits = new StringBuilder ();
for (int v = 0;
      values != null && v < values.length; ++v)
{
  fruits.append (values[v]);
  if (v != values.length - 1)
      fruits.append (", ");
}

request.setAttribute ("fruits", fruits);
```

  - **doGet** takes the unusual tack of calling **doPost**, since the query string in the link is functionally equivalent to that from the form.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

UIComponents

EXAMPLE

- Finally, see **docroot/results.jsp**.
  - It can echo most values like this:

```
<table>
  <tr>
    <td>text:</td>
    <td>${param.text}</td>
  </tr>
```

  - In the case of the multi-select list, it instead presents the servlet's assembled string:

```
  <tr>
    <td>fruits:</td>
    <td>${fruits}</td>
  </tr>
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

UIComponents

EXAMPLE

- Build and test the application, running traffic through the sniffer:

```
http://localhost:8079/UIComponents
```

UI Components

Text	Able was I
Password	••••••••••••••
Hidden	
Checkbox	True?
Radio buttons	Day Night
Single-select	South
Multiple-select	Apple Orange Banana
Submit button	Submit

Or, try this handy LINK.

– Enter values and click **Submit** ...

UI Components

text:	Able was I
password:	ere I saw Elba
hidden:	special code
flag:	on
dayOrNight:	day
direction:	3
fruits:	2, 3
bn:	Submit

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

UIComponents

EXAMPLE

- The sniffer shows the full query string:

```
POST /UIComponents/Submit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 120

text=Able+was+I&password=ere+I+saw+Elba&hidden=spec
ial+code&flag=on&dayOrNight=day&direction=3&fruit=2
&fruit=3&bn=Submit
```

- If you go back and click the LINK, you'll see a similar string, but now in an HTTP GET URL:

```
GET /UIComponents/Submit?text=Hello&password=guest&
hidden=secret&flag=on&dayOrNight=night&direction=3&
fruit=1&fruit=3 HTTP/1.1
```

  – ... and of course different results in the response page as well:

UI Components

text: Hello
password: guest
hidden: secret
flag: on
dayOrNight: night
direction: 3
fruits: 1, 3
bn:

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Editing Invoices

LAB 4B

Suggested time: 45 minutes

In this lab you will refactor your invoice JSP to allow the user to edit invoice data as well as to view it; and you will enhance the backing servlet so that it can process form input and update invoice records as requested.

Detailed instructions are found at the end of the chapter.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validation

- Whenever an application relies on user input, it is important to **validate** that input – meaning that we want to take reasonable steps to assure that the data we're given is:
  - **Present**! – the most common user error is that of omission
  - Of the **right type** – not a string where a number was expected
  - Of the **right length** – will it fit in the database column?
  - **Well-formed** – uses only certain expected characters, fits an expected pattern such as two letters then three digits, etc.
- Validation is a form of error handling that is applied early in the request-handling process, so as to save time and trouble later.
- Without validation, a given datum can trip up code much deeper in the application, with any number of possible consequences:
  - **Request crash** – an unrecoverable exception that forces us to display an error page to the user
  - **Application crash**
  - (eek) **Server crash**
  - Or perhaps the worst of them all ... no visible errors, but **bad data** in the database, just waiting to be discovered or cause bigger trouble later
- Validation also has a security aspect, because bogus inputs may result not from simple user ineptitude but from outright malice: this is a common tool used by hackers to break into web applications.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validation

- **There are many possible validation strategies ...**
  - Checking minimum and maximum **length**
  - Checking minimum and maximum **value**, or comparing values
  - **Regular expressions**
  - **"Known good"** validation, which means checking against a list of known legal values
  - **"Known bad"** validation, meaning scans for bad values or character patterns
- **... and many validation tools and utilities:**
  - **JSF** offers built-in validation support
  - So do **Struts**, **Spring**, and most other web frameworks
  - The **Apache Commons Validator** can be integrated, without using the entire Apache/Struts framework
- **Or you can build your own library of validation code for use in and around your application.**
  - **Front-controller** servlets and servlet **filters** can be good options.
  - Or sometimes a simple set of **utility classes** can ease the burden on individual servlets.
- **The important thing is to have a standard for validation practice as part of your development process, and to be diligent about implementing that standard.**
  - Make validation one of the criteria of your **code reviews**.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validating Origin and Destination

EXAMPLE

- **Notice the validation logic built into the Flights application, in Flights_Step2.**
- See **src/cc/travel/web/FlightSearchServlet.java**:
  - It uses a variable **message** to capture any necessary error message to the user, and another **goBack** to express whether it should re-serve the submitting page (which it might do either on an error or because the user was just searching for an airport by name):

```
String message = null;
boolean goBack = false;
```

  - If the user was searching for the origin airport by name, and the airport is not found, it sets a message:

```
if (request.getParameter ("origin-cmd") != null)
{
  Airport found = ...;
  if (found != null)
  {
    goBack = true;
    route.setOrigin (found);
  }
  else
    message = "Couldn't find that airport.";
}
```

  - It does the same thing for the destination airport.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validating Origin and Destination

EXAMPLE

- Otherwise, the user is looking to query for flights between two airport codes, which must be provided. But, were they?

```
else if (origin == null)
  message = "Please select an origin ...";
else if (destination == null)
  message = "Please select a destination ...";
```

- And, what if they're the same airport?

```
else if (origin.getCode() == destination.getCode())
  message = "Umm ...";
```

- Having validated these inputs, the servlet then decides what to do next:

```
if (message != null)
  request.setAttribute ("message", message);

if (message != null || goBack)
{
  request.getRequestDispatcher ("airports.jsp")
    .forward (request, response);
  return;
}
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

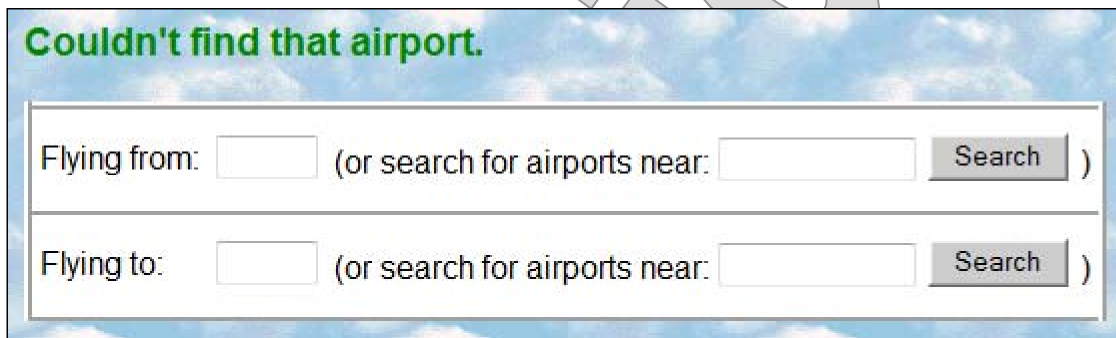
Validating Origin and Destination

EXAMPLE

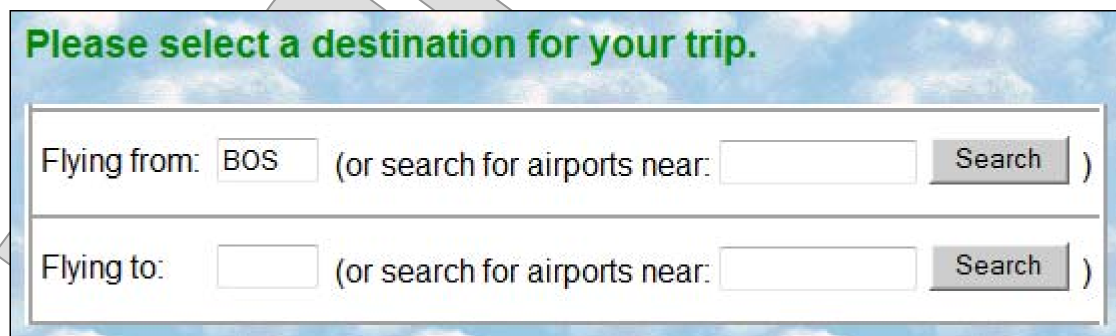
- **Build and test the application, with an eye to missing or incorrect inputs through the initial form.**

`http://localhost:8080/Flights`

  - See that failed searches are reported ...



  - ... and that missing or matching airport codes cause the page to be served again, rather than moving forward:



© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validating Invoice Data

DEMO

- **Let's add some validation logic to the Billing application.**
  - Do your work in **Billing_Step4**.
  - The completed demo is found in **Billing_Step5**.

1. First, build and test the application and provide bad data. See how it does not react all that well to negative invoice amounts, un-parseable amounts, or un-parseable dates.
2. Open **src/cc/billing/web/Records.java**. In **editInvoice**, catch any parsing exceptions when reading the invoice amount:

```
try
{
  invoice.setAmount (Double.parseDouble
    (request.getParameter ("amount")));
}
catch (NumberFormatException ex)
{
  request.setAttribute
    ("amountError", "Amount must be a number.");
}
```

3. While we're at it, check that the invoice amount is positive:

```
try
{
  invoice.setAmount (Double.parseDouble
    (request.getParameter ("amount")));
  if (invoice.getAmount () <= 0)
  {
    request.setAttribute ("amountError",
      "Amount must be a positive number.");
  }
}
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validating Invoice Data

DEMO

4. In **docroot/invoice.jsp**, add a third cell to the table row for invoice amount, to show any error message that might have been posted by the servlet:

```
<tr>
  <td>Amount:</td>
  <td><input type="text" ... /></td>
  <td class="errorMessage" >${amountError}</td>
</tr>
```

5. Build and test the application, and try submitting un-parseable values and negative values for invoice amounts.

http://localhost:8080/Billing

Customer:	Customer Two	
Invoice number:	7890550	
Invoice date:	Tue May 10 20:50:52 EDT 2011	
Amount:	2048.53	**Amount must be a number.**
Date paid:		
Payment type: ()	Cash Check Credit card	

– Notice the user's input is lost, replaced by the original value. This isn't ideal, and will require more work in any validation strategy.

Amount:	-500.0	**Amount must be a positive number.**

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validating Invoice Data

DEMO

6. But, notice this: if you enter a negative amount, get the error message, and then navigate away from the invoice and navigate back ... the negative amount is there!

Invoice 7890550

Customer:	Customer Two
Invoice number:	7890550
Invoice date:	Tue May 10 20:50:52 EDT 2011
Amount:	-500.0
Date paid:	
Payment type: ()	Cash Check Credit card

Submit Changes

Previous Next Record 2 of 3

   – It has apparently been applied despite the error message – and if you think about it, we haven't yet added any logic to the servlet that would prevent this.

7. At the top of the **editInvoice** method, set a variable to track whether there are any validity errors:

```
boolean errors = false;
```

8. Then declare and initialize a variable **amount**:

```
double amount = 0;
```

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Validating Invoice Data

DEMO

9. Instead of calling **setAmount** immediately, defer this until we can validate the value. Just assign that value to **amount** for now. And, wherever we set an error message into the request, set **errors** to **true**:

```
try
{
  amount = Double.parseDouble
    (request.getParameter ("amount"));
  if (amount <= 0)
  {
    request.setAttribute ("amountError",
      "Amount must be a positive number.");
    errors = true;
  }
}
catch (NumberFormatException ex)
{
  request.setAttribute ("amountError",
    "Amount must be a number.");
  errors = true;
}
```

10. Now, after the **try/catch** block, call **setAmount**:

```
if (!errors)
  invoice.setAmount (amount);
```

11. Test again and you should see that there are two possible outcomes from submitting an invoice amount – and they have simple names! “Success” means a good value was saved in the data set; “failure” means an error message was shown and the data was not saved.

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Evaluation Only

Validating Invoice Data

DEMO

- **As an optional extension of this demonstration, you might effect the same improvements for the payment date.**
  - This will require some more complex logic, because you already have some conditional code in there for processing payment date and type together if the date is given.
  - And, more importantly, you'll want to check all values for validity before you pass any values to the model. That is, don't set the invoice amount or the date or the payment type, unless all submitted values are found to be valid.
  - Remember: the worst outcome is partial failure!
- **The answer code in Step5 validates both fields, as described above, so you may also want to review the code in that step.**
- **Finally, notice that the user loses his or her input value if there is a validation error.**
  - This is not cool, and if it happens a lot in an application it is guaranteed to make the user angry.

    "Wait: you won't take my input, and you've lost all my work?!?"
  - Fixing this issue requires a more muscular infrastructure that includes a "backing bean" to carry the submitted data – separately from the actual model bean, since we don't want to pollute that with invalid data – and code in the JSP to echo values from that intermediate bean onto the form.
  - We can get this from JSF, Struts, Spring, etc.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

SUMMARY

- **HTML forms are essential to most web applications.**
- **Building them is simple enough, although working with the raw HTML code can be time-consuming and error-prone.**
  - A **custom tag library** can ease the pain here considerably, and bring additional features to the application in the process.
  - Recommended tools here are the usual suspects: **JSF**, **Struts**, **Spring**, and other web frameworks.
- **Processing form input is similarly straightforward – and similarly clunky – in servlet code.**
  - This is another "pain point" addressed by most web frameworks.
- **It is important to validate form inputs – and all user inputs – before carrying out requested work.**
- **There are many types of validation constraints, and many strategies and tools for enforcing those constraints.**
  - And, yep – web frameworks provide some help with this as well – although the level of support varies much more widely for validation than it does for basic form-building and request processing.

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Geometry

LAB 4A

In this lab you will implement a servlet that reads user inputs through an HTML form that describe the dimensions of a three-dimensional ellipsoid. The servlet will set these values into a JavaBean which knows how to calculate values about the ellipsoid – volume, type, and description – and will pass that bean along to a JSP which presents a full report.

Lab project: **Ellipsoid_Step1**

Answer project: **Ellipsoid_Step2**

Files: **src/cc/math/EllipsoidServlet.java**

Instructions:

1. Open **EllipsoidServlet.java** and write your code into the **doGet** method, as follows.
2. Create a local variable **delegate** of type **Ellipsoid**, and initialize it to a new instance. The constructor takes no parameters.
3. Call **request.getParameter**, passing "a" as the parameter name. Pass the returned value to **Double.parseDouble**, and pass those results to **delegate.setA**.
4. Do the same for parameters and properties **b** and **c**.
5. Set the **delegate** reference as the value of a request-scope attribute called "ellipsoid". This is the name the JSP will use to read information.
6. Forward to **result.jsp**.

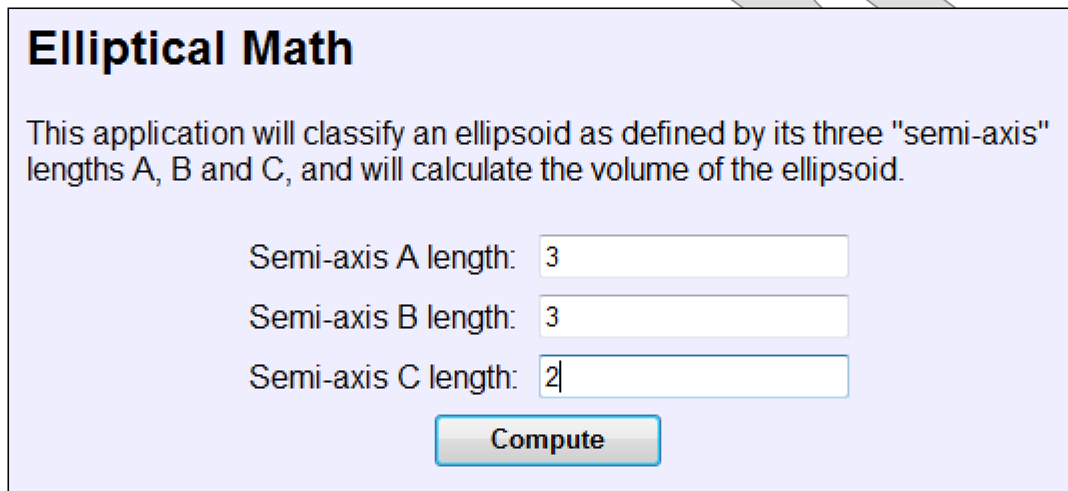
Evaluation Only

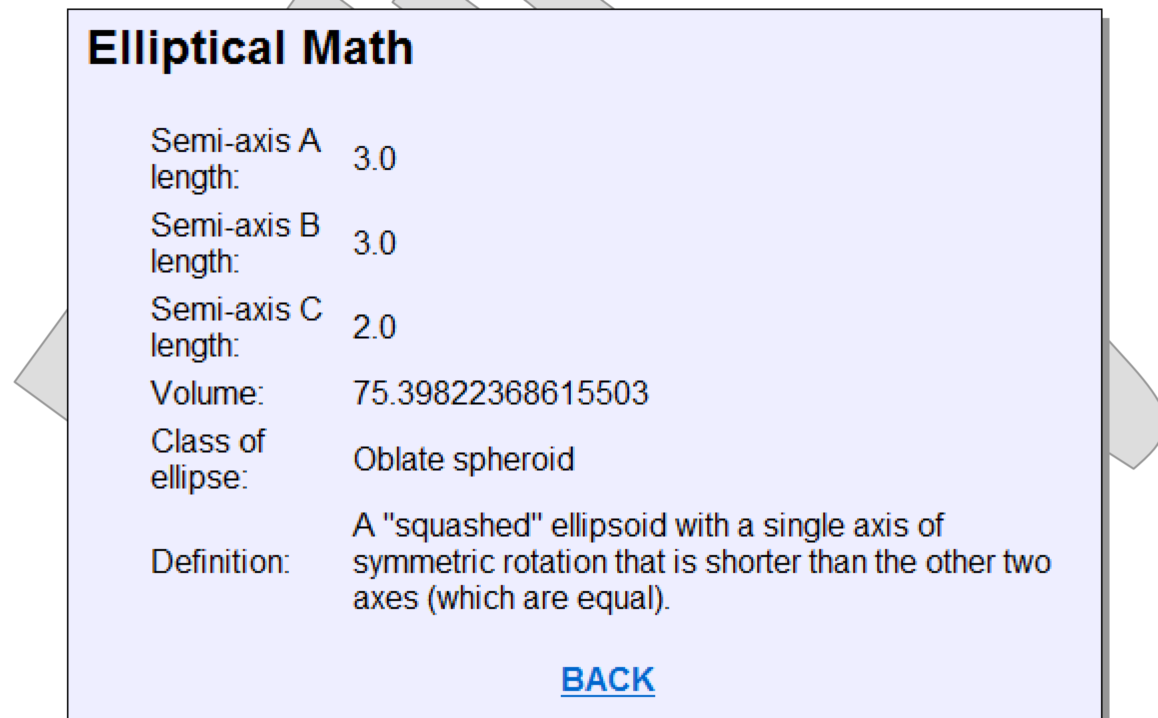
© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Geometry LAB 4A

7. Build and test, and you should see that you can enter values for the ellipsoid semi-axis lengths and see classifications and calculations based on your input:

```
http://localhost:8080/Ellipsoid
```





One glaring shortcoming of this application is that it does nothing to validate user inputs. We'll start to look at input-validation strategies later in this chapter.

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Editing Invoices

LAB 4B

In this lab you will refactor your invoice JSP to allow the user to edit invoice data as well as to view it; and you will enhance the backing servlet so that it can process form input and update invoice records as requested.

Lab project: **Billing_Step3**

Answer project: **Billing_Step4**

Files: **docroot/invoice.jsp**
src/cc/billing/web/Records.java

Instructions:

1. Open **invoice.jsp** and find the table row for the invoice amount.
2. Wrap the existing expression in a complete text field, like this:

```
<td><input
      type="text"
      name="amount"
      value="${invoice.amount}"
    /></td>
```

3. Do the same for the next row and the **paidDate** value.
4. See that radio buttons are already defined for the payment type. This will do for input, though we'll see later that it will take some JSTL logic to re-fill these radio buttons correctly when showing a selected invoice. So as to be able to confirm changes that you make when testing in this lab, add an expression to the text that precedes the radio buttons:

```
<td valign="top" >Payment type: (${invoice.paymentType})</td>
```

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Editing Invoices LAB 4B

5. Now, after the HTML table, but before your existing **<div>** that includes the **Next** and **Previous** buttons, add another **<div>** for a submit button, as shown below:

```
  ...
</table>
<div>
  <input
    type="submit"
    class="button"
    id="submit"
    name="submit"
    value="Submit Changes"
  />
</div>
<div>
  ...
```

6. Now open **Records.java**, and define a field that we'll use later to parse date input:

```
private static DateFormat dateFormatter =
    new SimpleDateFormat ("M/d/yy");
```

   You'll need to import the two types above, and also **ParseException**, from **java.text**.

7. Add logic to the existing **doPost** method:

```
else if (parameterNames.contains ("previous"))
    ...
else if (parameterNames.contains ("submit"))
{
    Invoice toEdit = getByNumber (number);
    editInvoice (toEdit, request);
    goToDetailPage (number, request, response);
}
else
    throw new IllegalArgumentException ("...");
```

8. Create a new private method **editInvoice**. Give it two parameters – an **Invoice** reference and an **HttpServletRequest** – and a **void** return type. Declare that it throws the **ServletException**.

9. Read the "amount" request parameter, parse it as a **double**, and pass this to **setAmount** on the given invoice object.

10. You might build and test and this point, and you would see that any changes submitted for invoice amounts would "stick," such that you could navigate away from the invoice and come back, and see the same value. Any other changes would be lost, because you have yet to read those inputs.

11. Still in **editInvoice**, initialize string variables **paidDate** and **paymentType** to the values of request parameters of the same names.

Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Editing Invoices LAB 4B

12. We're going to process both payment date and type only if the user enters a payment date. So, check to see if **paidDate** is **null** or the empty string: if so, call **setPaidDate** and **setPaymentType** on the invoice, passing **null** for both properties.

13. If both values are provided, then pass real values to each of the **setPaidDate** and **setPaymentType** methods. Both values will require some type conversion. and the date parsing can throw an exception. So:

```
else
    try
    {
        invoice.setPaidDate (dateFormatter.parse (paidDate));
        invoice.setPaymentType
            (Invoice.PaymentType.valueOf (paymentType));
    }
    catch (ParseException ex)
    {
        throw new ServletException
            ("Couldn't understand the paid date", ex);
    }
```

14. Build and test, and you should now be able to adjust amount, payment date (try "4/5/11" or "10/10/10"), and payment type for any invoice; navigate away; navigate back, and see your changes reflected in the invoice presentation. See though how the payment type uses radio buttons for input but plain text for output. Also the dates both show in a pretty junky way. We'll improve on both of these in a later chapter.

```
http://localhost:8080/Billing
```


Evaluation Only

© 2007-2013 Will Provost.
All rights reserved by Capstone Courseware, LLC.