

HW4 Amazon.edu

For HW4, we'll use servlets to build a little bookstore called Amazon.edu. The conceptual structure of Amazon.edu is similar to the CGI homework, but using servlets makes things a little easier. Amazon.edu is due midnight ending Thu June 7th. You may use at most four late days on this homework.

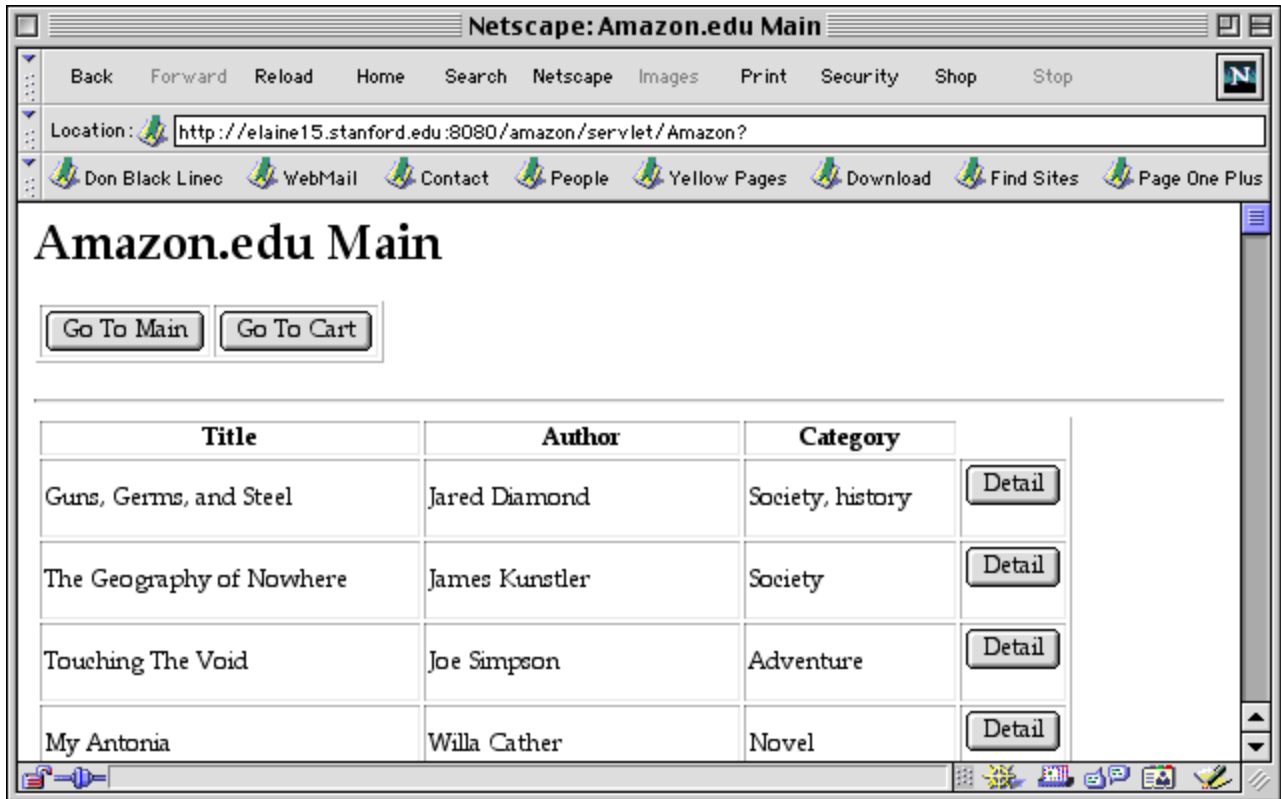
Part A

For Part A, we will build the main part of a little store. Part B will add the check-out feature. The solution will use CGI-type logic to create the main page and the cart page, and a JSP to create the detail page. (P/NC students may work in teams of two and do not need to do part B)

1. Main

Part A has a simple 3-page interface. The first page you see is the "main" page which features

- The title and h1 "Amazon.edu Main"
- "Go To Main" and "Go To Cart" navigation buttons (described below)
- A table listing all the books in the database by Title, Author, and Category (but omitting the description). There is a "Detail" button for each book. The top row of labels uses "<th>" while the others use "<td>".



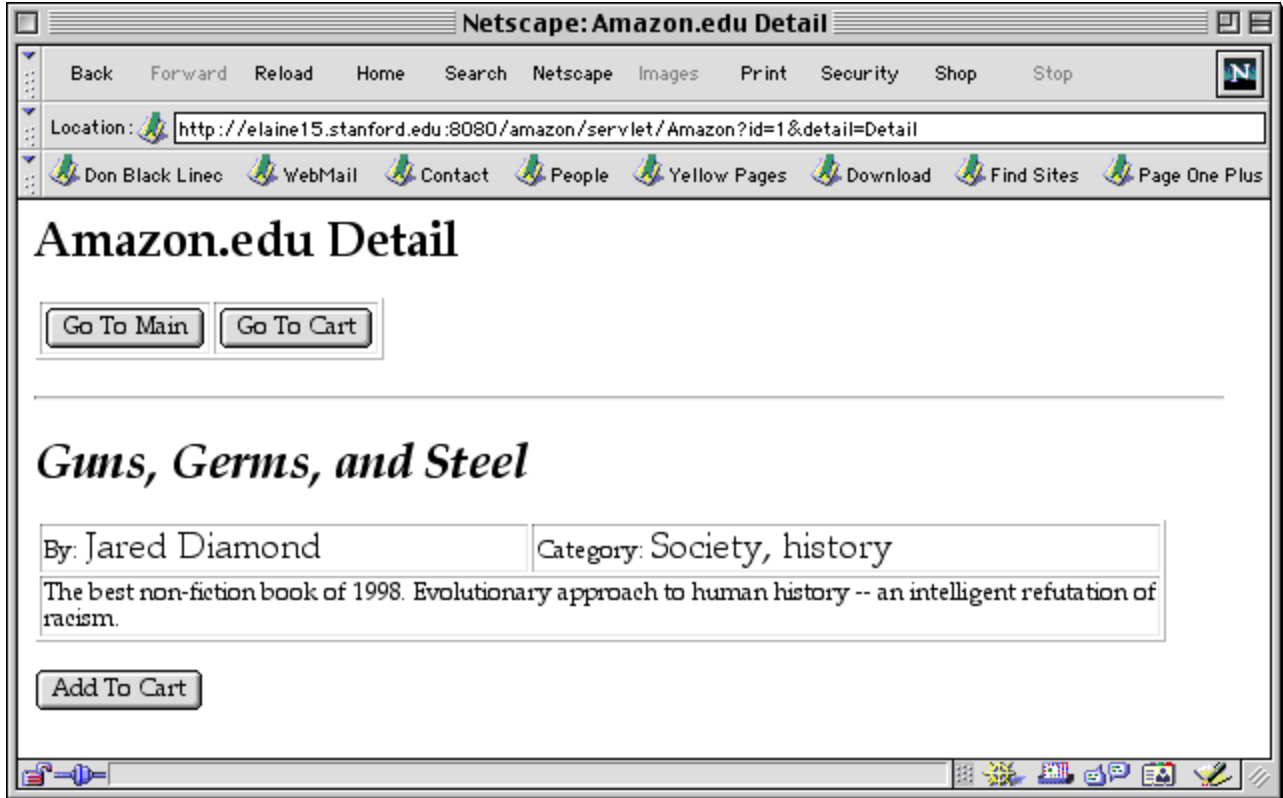
The TextDB class provided for you takes care of the details of reading the book database into memory.

2. Detail

Clicking the detail button for a book leads to a page showing...

- The title and h1 "Amazon.edu Detail"
- The standard buttons
- The information for that book presented with the following format...
 - The title is in a <cite> tag inside an <h2> tag
 - The author and category are in two cells in the first row one of a table. Use ... to make the author and category fonts a little bigger.
 - The description fills the second row of the table. Use <td colspan=2> to get the td to fill out both columns
- Finally, there's a "Buy it" button that can add the book to the cart

The detail page will be implemented as a little JSP. The JSP is well suited to building the nested HTML structure of the detail page.

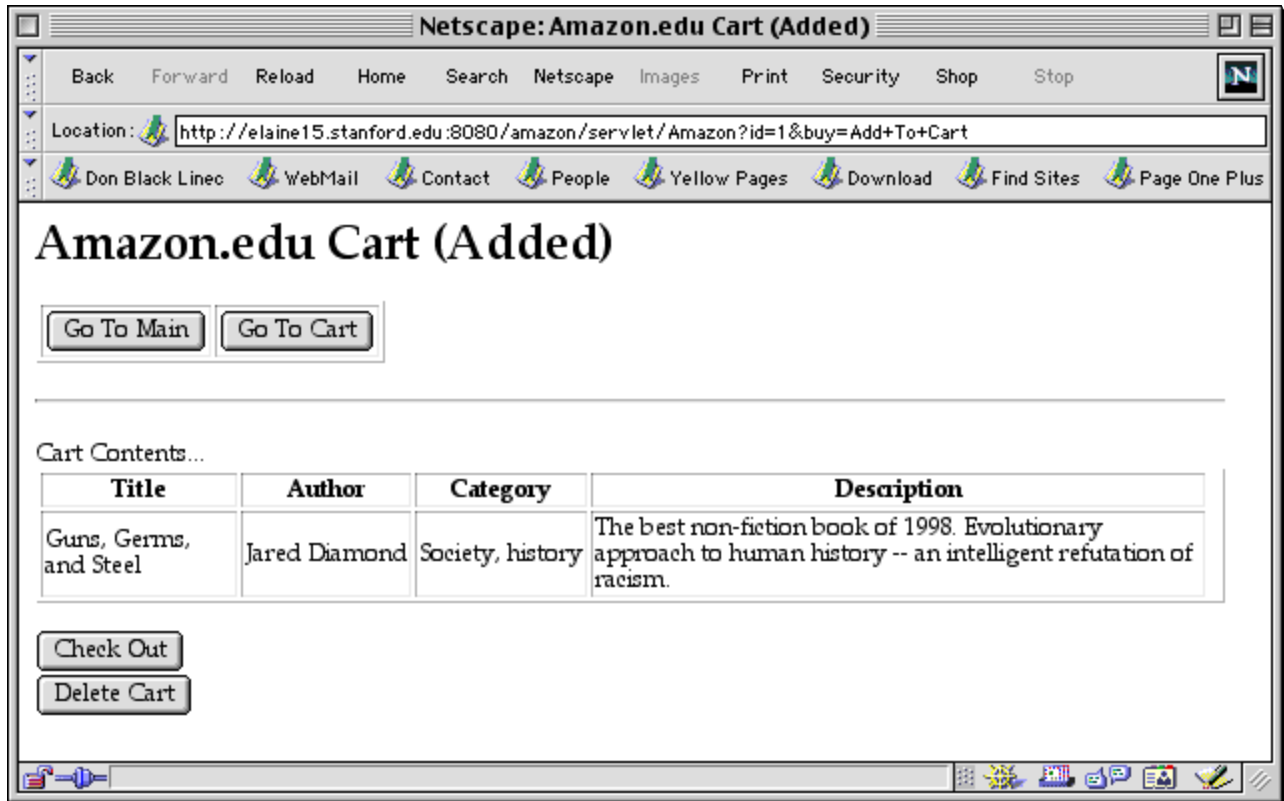


Clicking the Add To Cart button adds the book to the session shopping cart, and goes to the shopping cart page (below) to show the cart state. Clicking the "Go To Cart" button from here and from the main page are both similar to clicking Add To Cart from here. The difference is that the Add To Cart button changes the cart state. Adding a book which has already been added should go to the cart page but without changing the quantity—the only quantities allowed in our little binary shopping hell are 0 and 1.

3. Cart

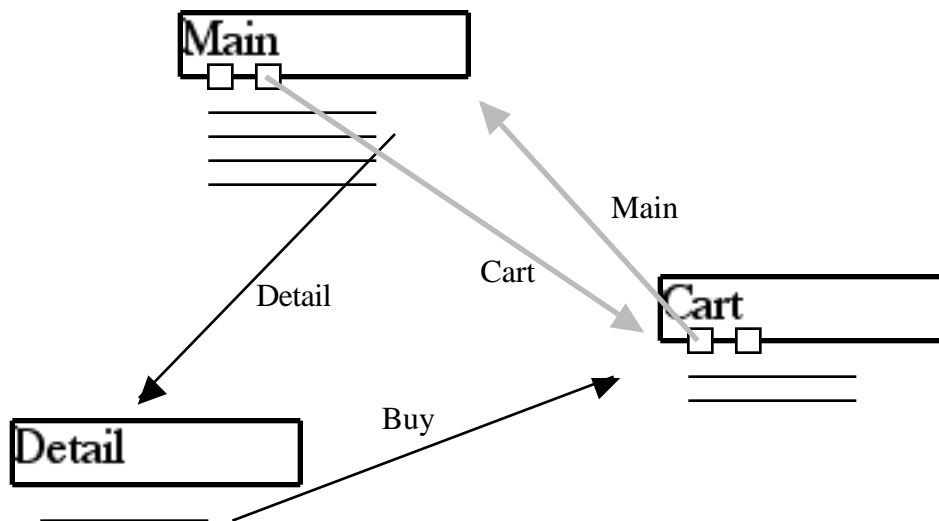
The Cart page shows a table similar to the main page, but containing the subset of the books that have been bought, and with the word "bought" in bold in the rightmost column. The cart page shows...

- The title and h1 "Amazon.edu Cart" if coming from the Go To Cart button. If we are coming here because of a click on the buy button, the string "(Bought)" should be added to the titling to provide more rational feeling feedback for the buy button.
- The standard buttons
- The table should show the current set of added books — Title, Author, Category, and Description for each one.
- The "Delete Cart" should change the cart to contain zero books and return to the main page. We'll ignore the "Check Out" button for now.



Navigation

Here's a map showing the most common transitions — the book buying path is in black, and the navigation button paths between main and the cart are in gray...



Implementation Ideas

As with the CGI homework, the solution will be to build a single servlet which looks at the request bindings to see what page to generate. Internally, the servlet session feature will be the perfect way to keep track of the current state of the cart.

doGet()

As with the CGI homework, the servlet needs to figure out which page it's supposed to send back. Use `(request.getParameter("buttonName") != null)` to check for bindings in the request.

To keep things organized, separate out the `doDetail()`, `doCart()`, and `doMain()` methods to generate the individual pages.

System.out.println()

Use `System.out.println("string")` to print out debugging information. This will show up in your server terminal. It's fine to print out lots of stuff -- this is the only debugging tool we have.

Code Re-Use

There's great potential for code re-use here. Don't be afraid to make up your own private utilities with a few parameters if the utility can then be called from several places. Create a utility method to generate the common HTML at the top of every page. Create a row-writing method that you can use to create the table rows for the main page and the cart page.

Session

The session is the perfect way to track the current collection of bought books. At the start of `doGet()`, detect if there is a session, and if not create a new empty one. The lecture example installed the "Tracker" object in the session. In this case, an even simpler strategy will work — just install a `Vector` object directly in the session since a single `Vector` is all that's needed to store the shopping cart state. In the `Vector`, store the row numbers of the bought books. Use `session.setAttribute("string", object)` to put things in the session.

Buying books and printing out the contents of the cart can use the cart `Vector` stored in the session. Every time the servlet is invoked, the same session will be present. As a result, the Amazon servlet should not store session specific state in traditional instance variables. The session object provides all the necessary storage for the servlet. This solves the traditional consistency problems with the forward and back buttons, and multiple windows aimed at the store at the same time (try it — they'll automatically share the session as with the lecture example).

Vector/String/int Operations

Since the database never changes, we do not need to do the nasty "id" scheme as in the CGI homework. The row number will be sufficient to identify each book. It turns out to be convenient to store and manipulate the row numbers in their `String` form, and convert them to `int` only when necessary. Here are some handy code snippets for `Vector` and `String` operations...

```

Vector cart = new Vector();

String str = "6";

cart.removeElement(str); // search for str in the vector and remove it
cart.addElement(str);    // add str to the end of the cart Vector

///// Do something with all the Strings in the cart Vector
for (int i=0; i<cart.size(); i++) {
    str = (String) cart.elementAt(i);
    // do something with str
}

cart.clear();           // remove all the elements from a vector

///// String/int conversions
int n = Integer.parseInt(str); // figure the int value for a string
String s = Integer.toString(n); // figure the string form of an int

///// String compare (both strings must be non-null)
string1.equal(string2)

///// Concat strings and ints using +
int n;
String string;

("hello" + string + n) // Converts the int to string
                        // and concatenates it all together.

```

TextDB

The TextDB classes provided for you does the work of reading the text database into memory. The servlet should have a single "db" instance variable that points to a TextDB object that contains all the row data. The first time doGet() is called, create the TextDB object, passing the name of the file it should read from...

```

if (db == null) {
    db = new TextDB("books.txt");
}

```

The TextDB is created the first time and stored in the db ivar. For subsequent calls to doGet(), the db is ready. Your code can send two messages to the db to get data out of it...

- int size() -- the number of rows in the db.
- String[] getRow(int num) -- return an array of strings representing one row. Row 0 is the column headers for the database, and the subsequent rows are data.

Detail Page

There's a simple BookBean class provided for you. Create an instance of BookBean on the request for the JSP. Your JSP should be called `"/detail.jsp"`. If there are errors in your JSP, the error messages generated will be quite cryptic. JSPs are very sensitive to syntax -- compare your JSP with the lecture example carefully to check your syntax.

Setup Logistics

Here's how to create your tomcat/servlet setup. The setup is a little complex, since the open-source "tomcat" system we are using is intended for commercial deployments.

1. Java

Edit your `.cshrc` file in your home directory. Add the following line to set your java classpath.

```
setenv CLASSPATH /usr/class/cs193i/tomcat/lib/servlet.jar
```

Then, edit your unix path variable includes the directory `"/usr/pubsw/apps/jdk-1.2.2/bin"`

Here's what the relevant part of my `.cshrc` looks like afterwards...

```
# Now we actually set the path.
# We add your home directory and bin directory to the system list.
# You may add additional path customizations here.
set path=( \
    /usr/pubsw/apps/jdk-1.2.2/bin \
    /usr/class/cs108/staff/bin \
```

Do a "source `.cshrc`" to load the changes. Afterwards, you can check that it's done correctly...

```
elaine21:~/> which javac
/usr/pubsw/apps/jdk-1.2.2/bin/javac
elaine21:~/> echo $CLASSPATH
/usr/class/cs193i/tomcat/lib/servlet.jar
```

2. Directory Structure

In order to run servlets, your directories need to be set up just so — we'll use the `"/usr/class/cs193i/materials/servlet"` directory as a model. Use the following command from your home directory to copy the model servlet directory. The servlet directory can be located anywhere.

```
~/> cp -r /usr/class/cs193i/hw/servlet .
~/>
```

- The `servlet` directory contains the "books.txt" file. This is where you start and stop the server.

- The **servlet/conf** directory contains the `server.xml` configuration file
- The **servlet/webapps/amazon** directory contains an `index.html` that you may access at `http://server/amazon/`. This is also where JSPs go.
- The **servlet/webapps/amazon/WEB-INF/classes** directory contains all the `.java` and `.class` files.

1. Edit `server.xml`

Edit the `server.xml` file to use your own random port numbers. There are comments in the file showing you where to do this.

2. Server

CD to the `servlet` directory. Run the "startcat" command there. This should start the tomcat server and echo a lot of configuration. You will get an error message if an instance of tomcat is already blocked on your chosen port #. This terminal is where you will see `println`s from your servlet and other error messages. Use the `stopcat` command to stop the server.

3. Java Classes

Use a separate terminal down in `/servlet/webapps/amazon/WEB-INF/classes` to edit your `.java` files. Use "`javac *.java`" to compile them. If `javac` cannot be found, your unix path is wrong. If the servlet classes cannot be found, your java classpath is wrong (check the steps above).

4. Browser

From the browser, connect to your tomcat server...

`http://server:port/amazon/ --- get index.html`

`http://server:port/amazon/servlet/Amazon --- run the servlet`

You can recompile your `.java` file, and the server should notice and use the new `.class` file automatically -- you should not need to stop and start the server every time.

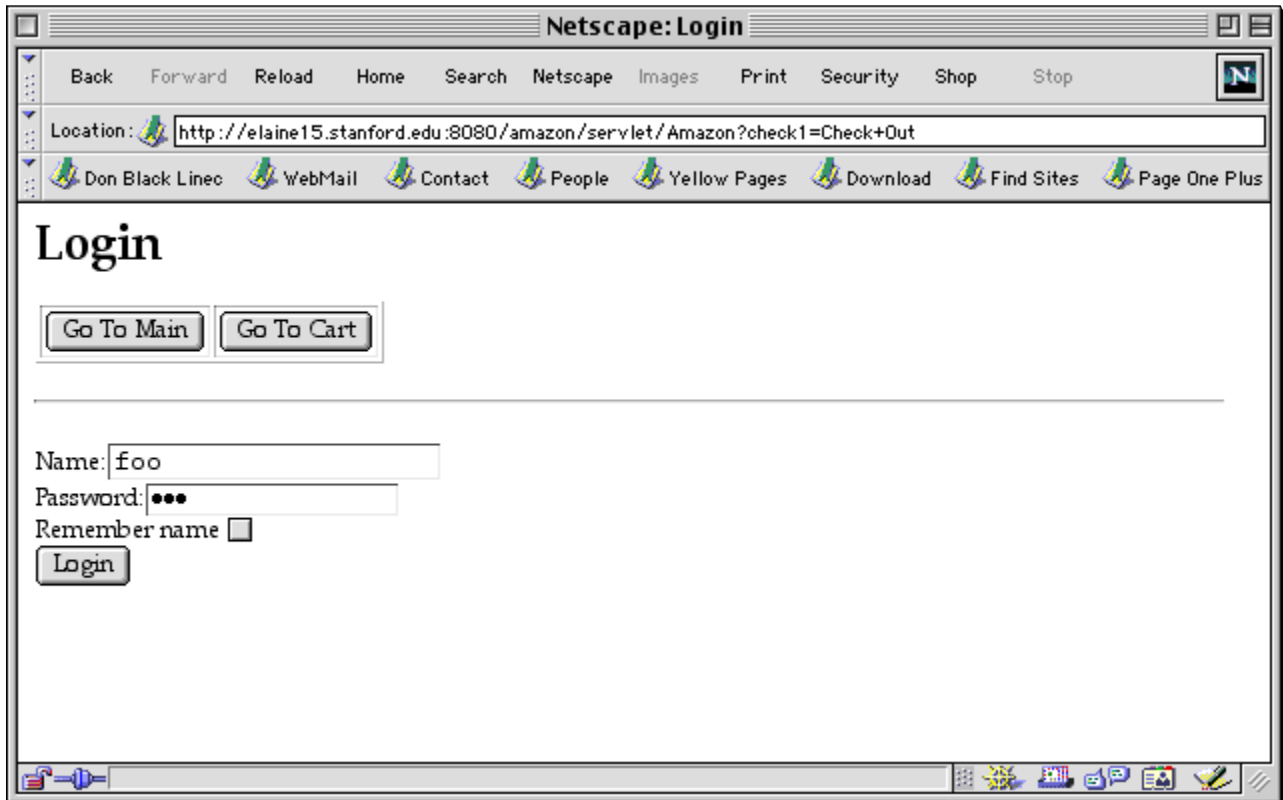
5. JSP

When it's time for the detail JSP, put it in `servlet/webapps/amazon` alongside the `index.html`.

Part B

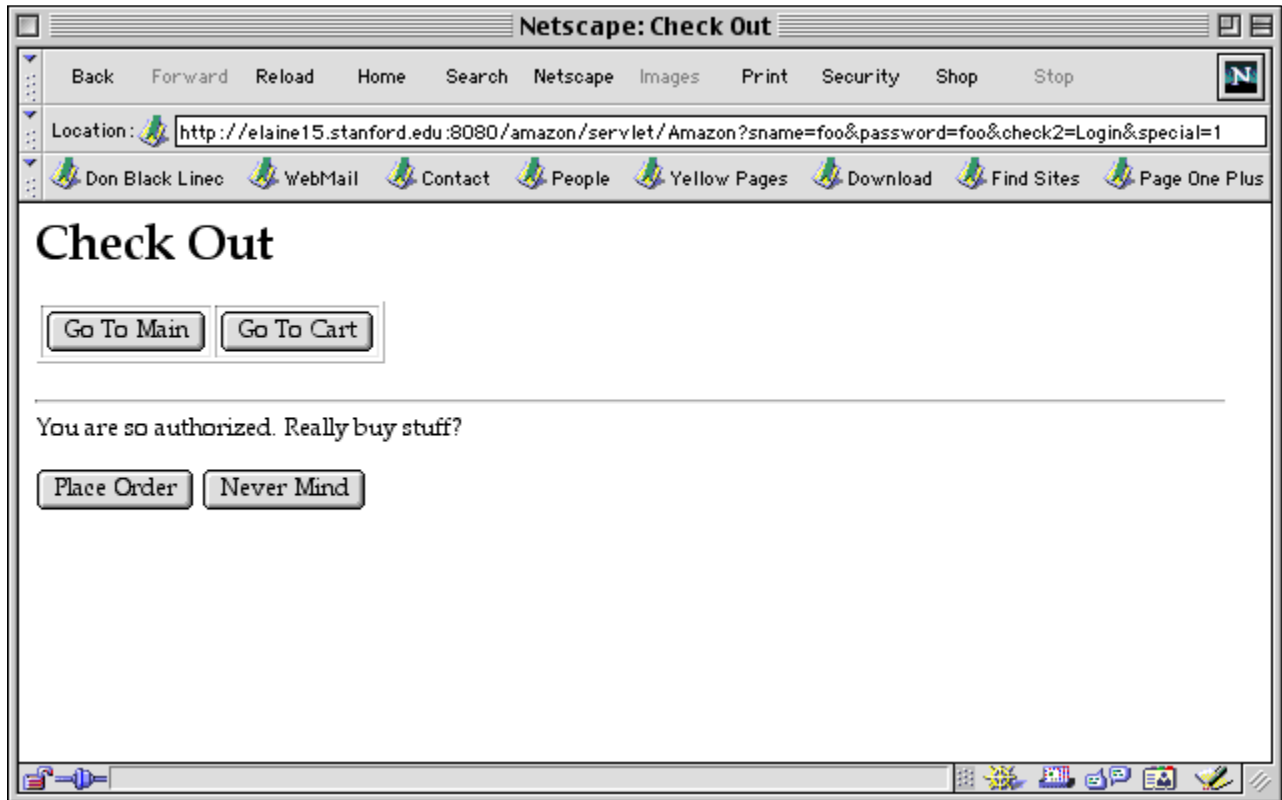
For Part B, we will make the check-out feature work.

There are three pages in the checkout process, which we'll call c1, c2, and c3. In the simplest case, the user is lead through the 3 pages in order. The Check Out button from the cat page leads to c1 where the user can put in their username and password...

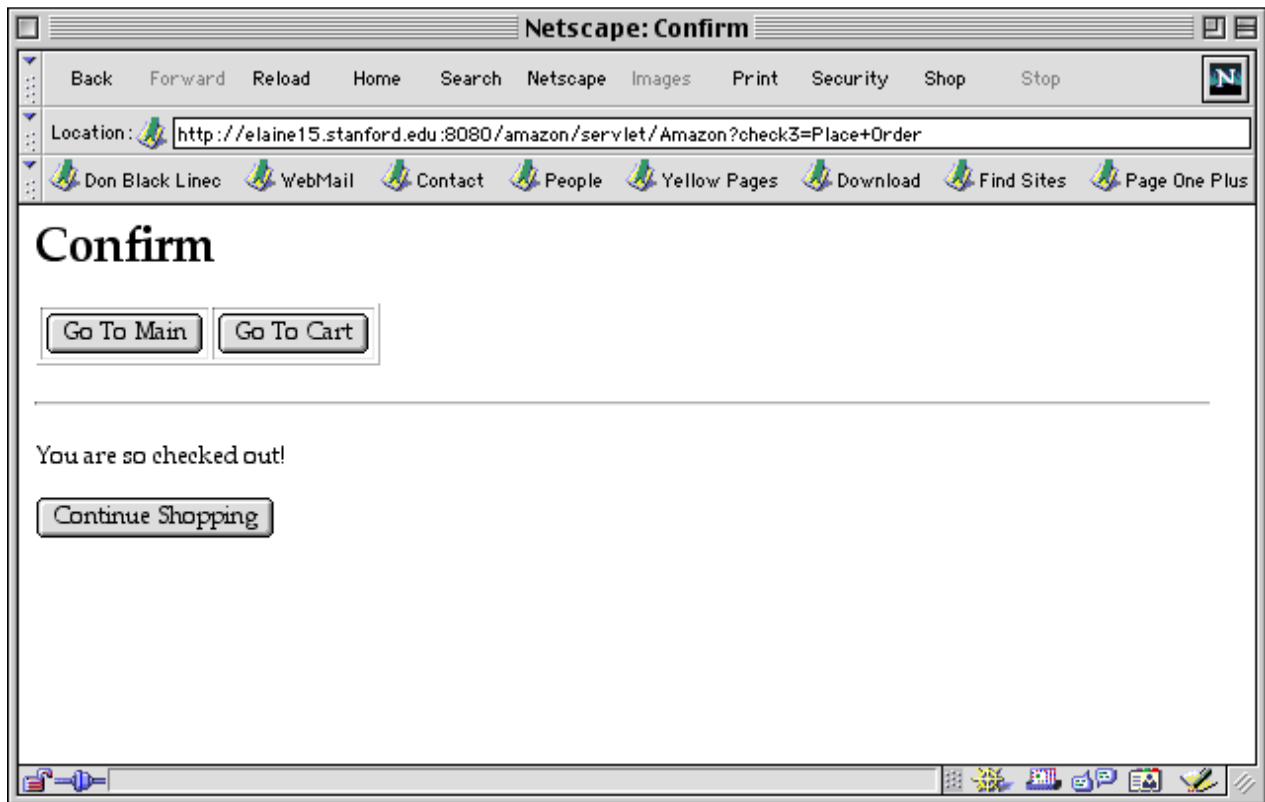


If the user tries to login, but the password is not correct, return page c1 again. At first, just let the password be the same as the username -- so if user "foo" logs in with password "foo", the password is good.

The c2 page confirms that the username and password are correct (the user is "authorized"), and the user can now choose to complete the purchase...



Placing the order leads to c3 which confirms that the order has happened and allows the user to return to the main page...



JSP

The c1, c2, c3 pages should all be coded as JSPs. They are mostly just static HTML, so it's regarded as good style to code them as JSPs instead of clogging up the servlet code with HTML.

Here are a few other cases for the check out pages...

Authorized

Suppose the user is seeing c2 -- they are authorized. Instead of checking out, the user returns to the main page and does some more shopping. When, later, they hit the Check Out button, the servlet should know that they are already authorized, and so skip c1 and go right to c2. Whether or not the user is authorized should be stored in the server side session, not in a hidden field or cookie.

Reset On Confirm

If the user confirms the order and so gets to c3...

- Their cart should be emptied out, so going to the cart page should show no books

- Their authorization state should be set back to not authorized, so going to c1 should ask for the password again. (This is for the "library" problem.)

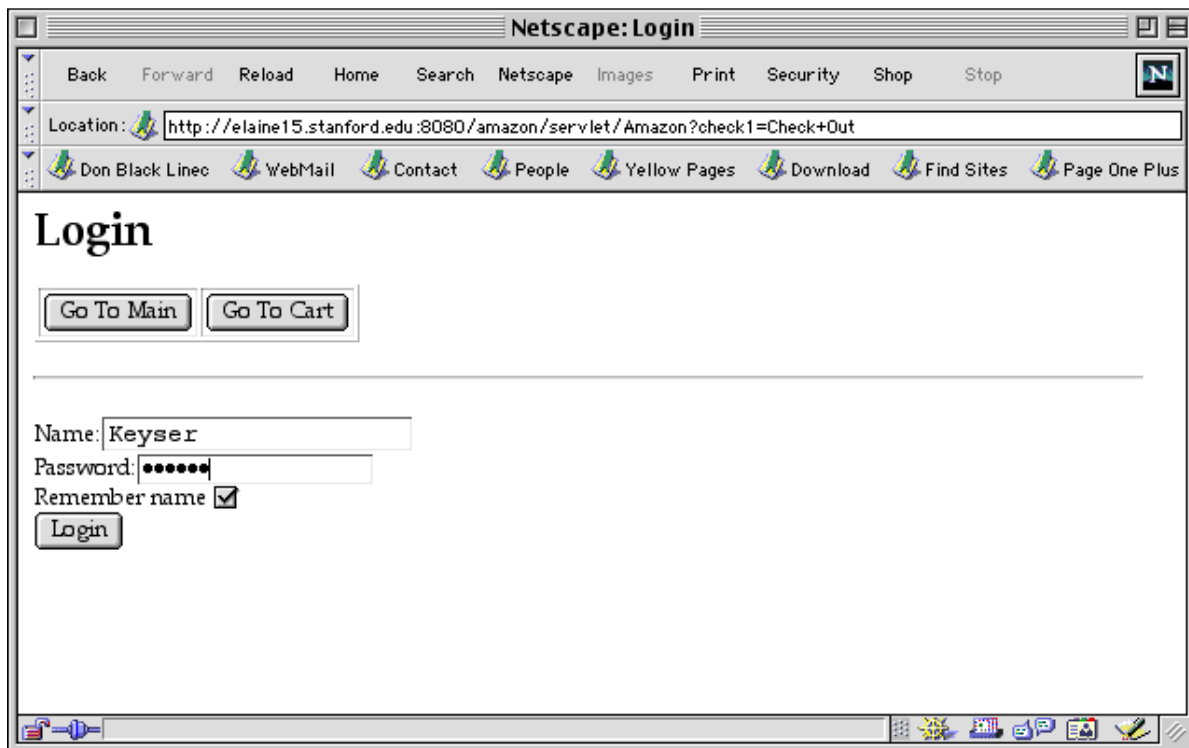
Remember Name

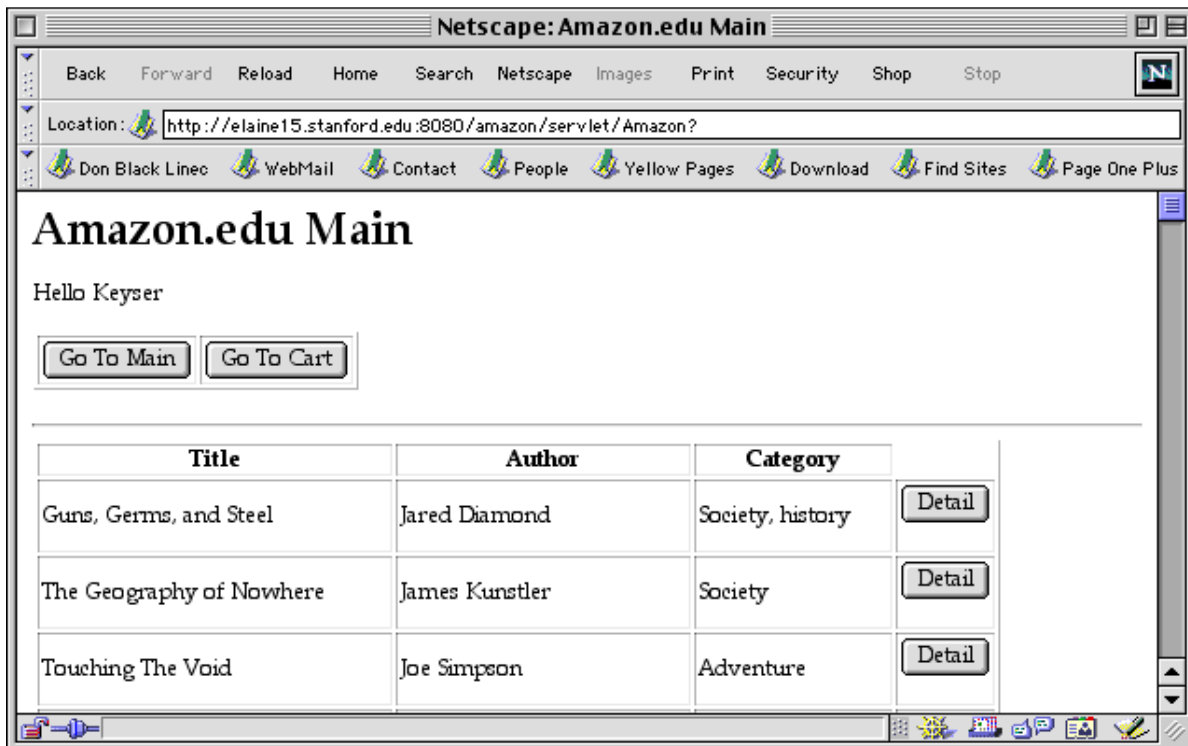
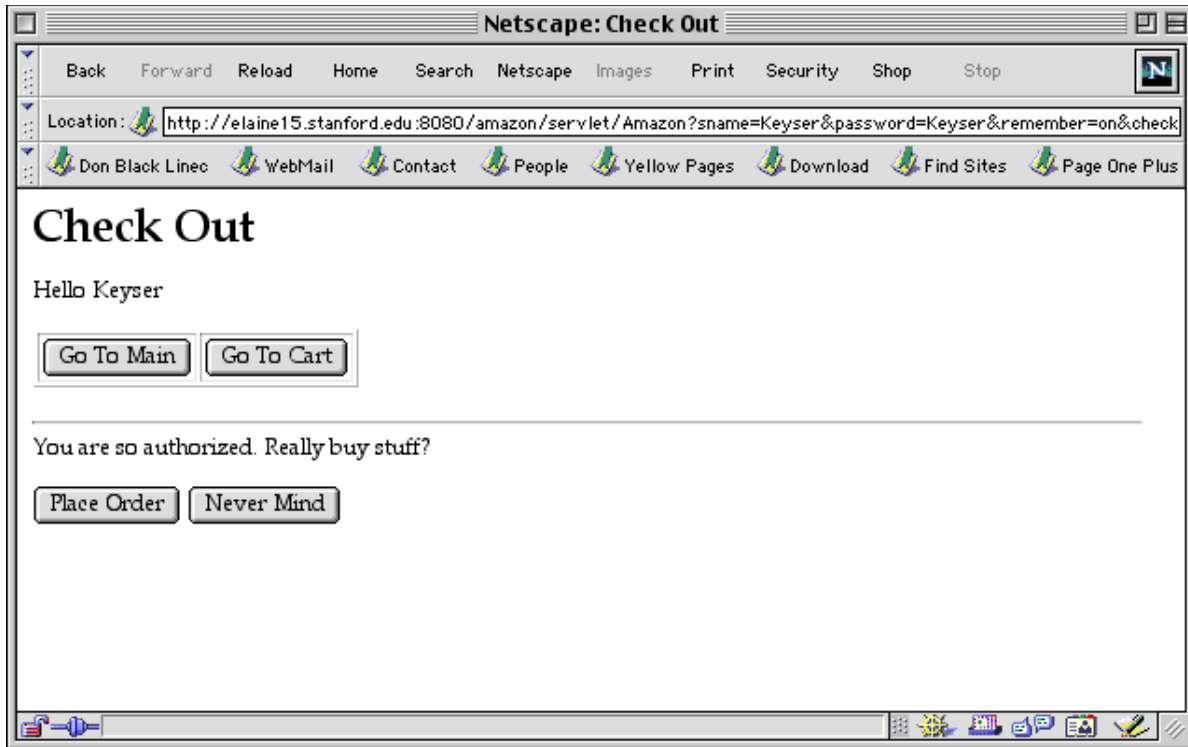
If the user checks the "Remember Name" checkbox, when submitting c2 with the Login button, then a persistent cookie should be set so that...

1. The top of every page includes a "Hello <name>" greeting below the buttons
2. The next time the user comes to the c1 page, the name field should already be filled in.

These effects should continue to work, even if the user restarts their browser and re-connects to Amazon.edu. If the "Remember name" check box is not checked when the login button is used, then the cookie should be cleared, so there is no greeting and the name is not pre-filled in the form.

Here's an example sequence where Keyser Soze logs in, but then decides to keep shopping -- notice the greeting at the top of the page...





XML Passwords

The last thing to fix is the passwords. We'll store the username/password data in a simple users.xml file...

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user name="foo" password="bar" />
  <user name="keyser" password="binky" />
  ...
</users>
```

The servlet should use the SAX parser to read the name/password pairs into one or two instance variables in the servlet. We'll assume that the username/password database never changes. A simple but acceptable strategy is to read the usernames in to one Vector and the passwords in to a second Vector. The username and password string matches can be case sensitive (i.e. just use `String.equals`).

Deliverables

Put your readme in your servlet directory, and submit the whole thing.