COMP/EECE 7012/8012

Exam 1

Spring 2013

Solutions

Name: ______________________________

Rules:

- No potty breaks.
- Turn off cell phones/devices.
- Closed book, closed note, closed neighbor.

Reminders:

- Verify that you have 9 pages of questions and 5 of figures.
- Don't forget to write your name.
- Read each question carefully.
- Don't forget to answer every question.

Additional Items:

- For questions that involve writing code:
  - You may omit `import` statements.
  - You may omit exception-handling code.

1. [10pts] Write HTML would create web page depicted in Figure 1. Your solution must include the following types of HTML elements (and no other types): **!DOCTYPE**, **a** (with **href** attribute), **body**, **h1**, **head**, **html**, **img** (with **src** attribute), **li**, **p**, **title**, **ul**. The link should go to http://acm.org/. Assume the image is in **WebContent/images/kitty.jpg**.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>My Web Page</title>
</head>
<body>

<h1>My Heading (Level 1)</h1>

<p>My paragraph.</p>

<p>Paragraph with <a href="http://acm.org">link to ACM</a>.</p>

<img src="resources/kitty.jpg">

<ul>
<li>Item 1
<li>Item 2
</ul>

</body>
</html>
```

2. [10pts] Consider the following scenario involving Subversion (SVN). Alice and Bob are both working on a shared project **MyProj** that is stored in an SVN repository. Bob does a **checkout** on the project. What does SVN do when Bob issues the **checkout** command?

When Bob issues the **checkout** command, svn creates a working copy of the **MyProj** files on Bob's machine.

Next, Bob edits the **MyProj** file **Foo.java**. Then, he does a **commit**. What does SVN do when Bob issues the **commit** command?

When Bob issues the **commit** command, svn creates a new revision of the **MyProj** files with Bob's edits in the repository.

Next, Alice does a **checkout** on **MyProj**. Then, Alice and Bob both edit **Foo.java** in parallel. Foo.java has over 100 lines of code. Alice edits a couple lines at the top of the file, and Bob edits a couple lines at the bottom of the file. Then, Bob does a **commit**. Finally, Alice does a **commit**. What does SVN do when Alice issues the **commit** command?

When Alice issues the **commit** command, svn responds with an out-of-date error and does nothing further.

What SVN command should Alice issue next and what would the result of the command be?

Alice should issue the **update** command. The result of the command will be that svn inserts Bob's edits into her working copy of **Foo.java**.

After the previous command, what command should Alice issue?

Alice should issue the **commit** command.

Imagine that the University of Memphis provides students with snacks (chips and a drink) during class and has a web app that students can use to order their snacks. Students go to a web page that presents them with the form depicted in Figure 2. By filling out and submitting the form, students can choose what flavor of drink and type of chips they will receive. After a student submits the form, he/she is presented with a page like that depicted in Figure 5, or in the event of an error, the error page in Figure 6.

3. [5pts] Below is a list of the components that make up this web app. For each one, tell what part of an MVC architecture it belongs to. Put the full name of the part (i.e., do not just put "V").

_____View_______ An HTML page with the form in Figure 2

_____View_______ A JSP for the order-summary page (see Figure 5)

_____View_______ A JSP for the order-error page (see Figure 6)

_____Model______ A plain old Java class that represents a snack order (see Figure 3)

_____Model______ A plain old Java class for storing/retrieving snack order records (see Figure 4)

__Controller___ A servlet class that receives requests from the Figure 2 form, uses the plain old Java classes (Figure 3 and Figure 4) to service the request, and responds using the JSPs (Figure 5 and Figure 6).

4. [15pts] On the next page, reverse engineer the servlet class for ordering a snack. That is, write Java code that implements the servlet.

   - The order-summary JSP assumes that the request contains an attribute with the name "order" that maps to a **SnackOrder** object that contains the order info.

   - Note the JSP paths in Figure 5 and Figure 6.

   - Note the Java API excerpts in Figure 7.

Write your answer to question 4 here.

```
@WebServlet("/orderSnack.do")
public class OrderSnackServlet extends HttpServlet {
     private static final long serialVersionUID = 1L;

     protected  void  doPost(HttpServletRequest  request,  HttpS-
ervletResponse response) throws ServletException, IOException {
          String drink = "";
          String chips = "";
          drink = request.getParameter("drink");
          chips = request.getParameter("chips");
          SnackOrder order = new SnackOrder();
          order.setDrink(drink);
          order.setChips(chips);
          request.setAttribute("order", order);
          SnackOrderDao dao = new SnackOrderDao();
          if (dao.insertOrder(order) == -1) {
               request.getRequestDispatcher("WEB-
INF/orderError.jsp").forward(request, response);
          } else {
               request.getRequestDispatcher("WEB-
INF/orderSummary.jsp").forward(request, response);
          }
     }

}
```

5. [8pts] Reverse engineer order-summary JSP (Figure 5). As in the previous question, assume that the request contains an attribute with the name “order” that maps to a **SnackOrder** object that contains the order info.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="com.snack.*" %>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Order Summary</title>
</head>
<body>

<h1>Your Order</h1>

<ul>
<li>Drink: <%= ((SnackOrder) request.getAttribute("order")).getDrink() %>
<li>Chips: <%= ((SnackOrder) request.getAttribute("order")).getChips() %>
</ul>

</body>
</html>
```

E-commerce sites, like Amazon.com, commonly allow users to add items to a “shopping cart” while they browse. Then, when they’ve selected all the items they want, they can “check out” and purchase the items in their cart.

Consider the **Cart** class excerpt in Figure 9 that implements shopping-cart functionality and the servlet **doGet** method in Figure 10 that removes the most expensive item from the cart.

6. [6pts] Based on the above code, describe a scenario in which a call to **doGet** removes an item from the cart that is not the most expensive item. (Hint: concurrency.) Make sure that your answer is thorough and concise.

Assume there are two threads T1 and T2, each of which is about to execute the **doGet** method. Also assume that there are three items in the cart. The item at index 0 is most expensive; the item at index 1 is least expensive; and the item at index 2 has a price that falls in between the other two items.

T1 executes first. It completes the call to **findMostExpensive**, which returns the index 0. But then a context switch occurs.

T2 executes next. It completes the entire call to **doGet**. Since the item at index 0 is still the most expensive, T2 removes that item from the cart. As a result, the other items slide forward, so now the item at index 0 is the least expensive and the item at index 1 is most expensive.

Another context switch occurs, and T1 starts executing again. Since it previously got the index 0 from the **findMostExpensive** call, it proceeds with removing the item at index 0. Of course, due to T2’s actions, the item at index 0 is no longer the most expensive.

7. [6pts] Rewrite the **doGet** method from Figure 10 to correct the error. I've started the method for you:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ... {
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ... {
      HttpSession session = request.getSession();
      synchronized (session) {
             ...
             Cart c = (Cart) session.getAttribute("cart");
             int x = c.findMostExpensive();
             if (x != -1) { c.removeItem(x); }
      }
      ...
}
```

8. [10pts] Given the TVGuide database in Figure 8, what would the result of the following query be?

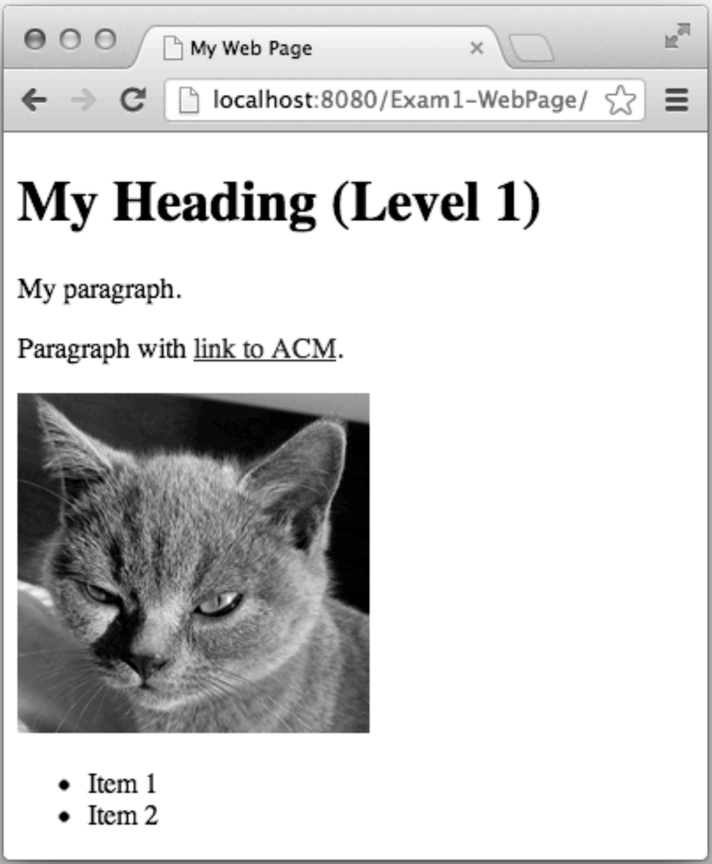
```
SELECT
    `LastName`, `NetworkName`
FROM
    (`TVNetwork` INNER JOIN `TVShowNetwork`
     ON `TVNetwork`.`NetworkID` = `TVShowNetwork`.`NetworkID`)
INNER JOIN
    (`Talent` INNER JOIN `TVShowCreator`
     ON `Talent`.`TalentID` = `TVShowCreator`.`TalentID`)
ON `TVShowNetwork`.`ShowID` = `TVShowCreator`.`ShowID`
ORDER BY `NetworkName`, `LastName`;
```

Fill the table below with your answer. Cross out any cells in the table that you do not need. Don't forget to label the columns.

LastName	NetworkName		
Groening	Comedy Central		
Groening	Fox		
Whedon	Fox		

Name: ____

Figures


My Web Page

localhost:8080/Exam1-WebPage/

My Heading (Level 1)

My paragraph.

Paragraph with link to ACM.

- Item 1
- Item 2

Figure 1. Example web page.

```
<form method="post" action="orderSnack.do">
ORDER SNACK HERE
<br>
Drink flavor: <input type="text" name="drink">
<br>
Type of chips: <input type="text" name="chips">
<br>
<input type="submit" value="Submit">
</form>
```

Figure 2. Snack-order form.

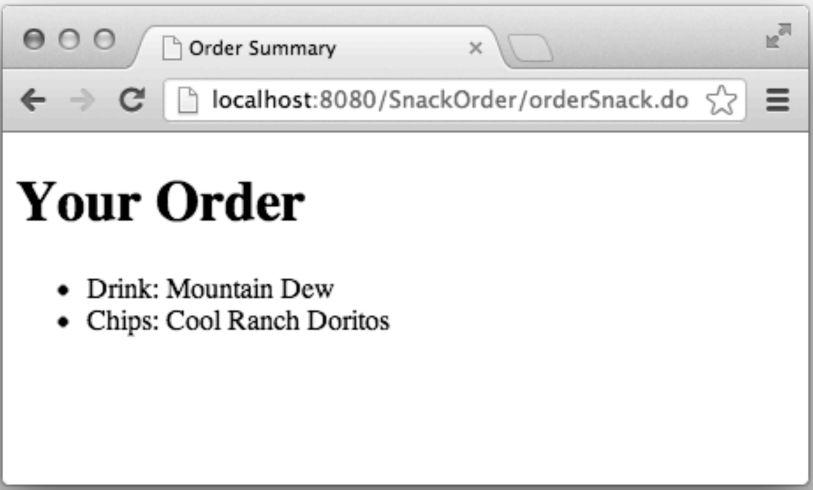
```
public class SnackOrder {
      private String drink;
      private String chips;
      public void setDrink(String s) { drink = s; }
      public void setChips(String s) { chips = s; }
      public String getDrink() { return drink; }
      public String getChips() { return chips; }
}
```

Figure 3. Java class that represents a snack order.

```
public class SnackOrderDao {
      /**
       * Returns newly created order number or -1 on error.
       */
      public int insertOrder(SnackOrder order) {
             ...
      }

      ...
}
```

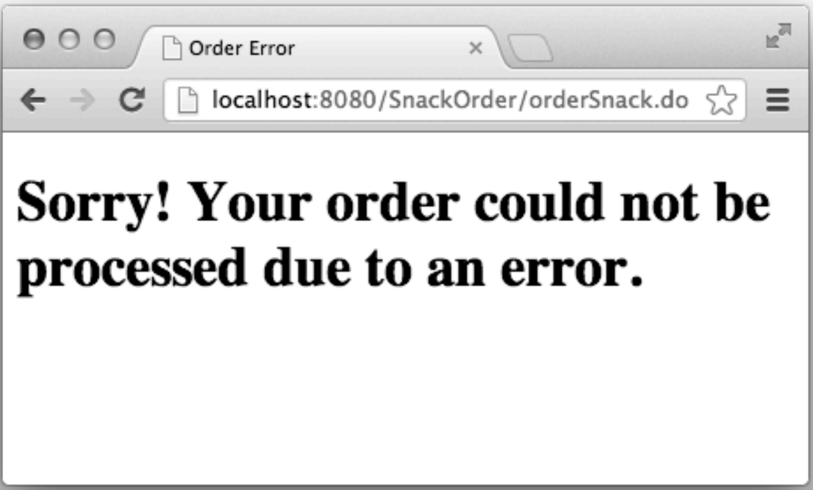
Figure 4. Java class for storing/retrieving snack-order records.



Your Order

- Drink: Mountain Dew
- Chips: Cool Ranch Doritos

Figure 5. WEB-INF/orderSummary.jsp



Sorry! Your order could not be processed due to an error.

Figure 6. WEB-INF/orderError.jsp

- Class **HttpServlet**
  - Annotation to declare URL pattern: **@WebServlet**
  - protected void **doGet**(HttpServletRequest req, HttpServletResponse resp)
  - protected void **doPost**(HttpServletRequest req, HttpServletResponse resp)
  - protected void **service**(HttpServletRequest req, HttpServletResponse resp)
- Interface **HttpServletRequest**
  - Object **getAttribute**(String name)
  - String **getParameter**(String name)
  - RequestDispatcher **getRequestDispatcher**(String path)
  - HttpSession **getSession**()
  - void **setAttribute**(String name, Object o)
- Interface **HttpServletResponse**
  - PrintWriter **getWriter**()
- Interface **RequestDispatcher**
  - void **forward**(ServletRequest request, ServletResponse response)
- Class **PrintWriter**
  - void **print**(String s)
  - void **println**(String x)
- Interface **HttpSession**
  - Object **getAttribute**(String name)
  - boolean **isNew**()
  - void **setAttribute**(String name, Object value)

Figure 7. Java API excerpts.

TVNetwork

NetworkID	NetworkName
111	CBS
666	Comedy Central
999	Fox

TVShow

ShowID	ShowName	Seasons
2222	The Simpsons	24
5555	Futurama	7
8888	Firefly	1

TVShowNetwork

ShowID	NetworkID
2222	999
5555	666
8888	999

Talent

TalentID	LastName	FirstName
333333	Groening	Matt
777777	Whedon	Joss

TVShowCreator

ShowID	TalentID
2222	333333
5555	333333
8888	777777

Figure 8. TVGuide database.

Class **Cart**

- private Item[] **items**
    - Array of items in the cart. Not sorted in any particular way.
- public int **findMostExpensive**()
    - Returns the index of the most expensive item in the cart, or -1 if the cart is empty.
- public void **removeItem**(int i)
    - Removes the item at index i from the cart.

Figure 9. Cart class excerpt.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ... {
      HttpSession session = request.getSession();
      ...
      Cart c = (Cart) session.getAttribute("cart");
      int x = c.findMostExpensive();
      if (x != -1) { c.removeItem(x); }
      ...
}
```

Figure 10. Servlet doGet method that removes the most expensive item from the cart.