Exercises: Getting Started

Since these exercises are of the "test your IDE and deployment process" variety, you probably want to do all of them, in order. In later exercises you can pick and choose depending on your experience. These exercises assume you have Eclipse set up with Tomcat 7 in development mode. For details, see http://www.coreservlets.com/Apache-Tomcat-Tutorial/, then click on link to Tomcat 7.

1. Make sure you can deploy and test the "test-app" project. R-click on Tomcat at the bottom, choose "Add and Remove Projects", choose "test-app", and Add. Then start Tomcat by R-clicking and doing "Start". Then enter http://localhost/test-app/ in a browser. Click on the links to the HTML and JSP pages to run them. Then click on the links to test the servlets. Note that you do not have to change any code anywhere: this exercise is just to test that you have Eclipse and Tomcat configured correctly, and that you know how to deploy apps.

2. Create a new Eclipse application called "test". (File, New, Project, Web, Dynamic Web Project. Or, if "Dynamic Web Project" is shown as an option after choosing "New", select it. Be sure to choose Tomcat 7 as the Target runtime.)

Create a package called "newPackage". R-click on "Java resources: src" and choose "New" and then "Package". Go to the test-app project, R-click on TestServlet (under src/testPackage), and choose "Copy". Go back to test/src/newPackage, R-click, and choose "Paste".

Drop hello.html and hello.jsp into WebContent.

3. Deploy the test project on Tomcat. (R-click on Tomcat 6 at the bottom, choose "Add and Remove Projects", choose the test project, and click "Add".) Then:

- Start the server. R-click on Tomcat and choose Start.
- Start a Web browser and enter http://localhost/test/.
- Run both hello.html and hello.jsp.
- Run TestServlet (http://localhost/test/test1)

4. Edit hello.html in some small way, save it, and re-run it to see the changes. Note that you do *not* have to redeploy the project or restart the server.

5. Edit TestServlet.java in some small way, save it, and re-run it to see the changes. You can guess on how to change the output without really knowing servlet syntax yet. If you change the body of the doGet method, you should be able to re-run without restarting the server. But, if you change the @WebServlet annotation, you will need to restart the server after doing so. Restart the server by R-clicking on Tomcat and choosing Restart.

Exercises: Servlet Basics

Make a new Eclipse project called exercises-basics (or some such). Note that my solutions are in the project called basics-exercises. For this and all other exercises in the course, we will review the solutions before starting the next lecture. But you can always peek at my solutions ahead of time if you want.

1. Make a package called servletBasics (or some other name of your choosing) and copy TestServlet from your previous project. Note that if you copy from the testServlet package (which is what I happened to name the package in the "test-app" project) and paste into some other package, Eclipse will automatically fix the package statement for you. Modify it so that it says "Hello *Your-Name*". Deploy the application and run the new servlet. (http://localhost/*your-project-name*/*your-address-from-the-WebServlet-annotation*)

- You do not need to use my ServletUtilities class, but if you want to, either copy ServletUtilities to the servletBasics directory and change "package coreservlets" to "package *yourPackage*", or (better!) leave it in the coreservlets directory and add "import coreservlets.*" to whichever of your servlets uses ServletUtilities. I think it is better at the beginning to skip ServletUtilities and make your servlets self-contained.

2. Create a servlet that makes a bulleted list of four random numbers. I find it easier and better to copy and rename an existing servlet like TestServlet than to do "New Servlet" in Eclipse, but you can try it both ways and see which you like better.

- Reminder 1: you use Math.random() to output a random number in Java.
- Reminder 2: you make a bulleted list in HTML 4 as follows

```
<UL>
   <LI>List item 1
   <LI>List item 2
   ...
</UL>
```

(In xhtml, you should use lower case and include the </li> tags.)

3. Create a servlet that uses a loop to output an HTML table with 25 rows and 10 columns. For instance, each row could contain "Row*X*, Col1", "Row*X* Col2", and "Row*X* Col3", where X is the current row number. If you don't know the HTML syntax for tables, please ask me.

Exercises: Form Data

1. Open the “forms” project. Run the ThreeParams servlet directly via http://localhost/forms/three-params. You should see null for all three parameters. Run it again by starting with three-params-form.html and submitting the form. You should now see the entries from the textfields.

2. Make a new Eclipse project called exercises-forms (or some such). Use this project for the rest of the exercises.

3. Copy the ThreeParams servlet from my “forms” project and put it in a new package/directory in your project. Similarly, make a copy of three-params-form.html. Test the servlet and the form.

4. Copy/rename the ThreeParams servlet and form and have the new version use POST instead of GET. Note that you will have to make a small change to *both* the servlet and the form. You will probably need to restart Tomcat after making the changes (R-click Tomcat and choose Restart).

5. Use three-params-form.html to send data to the ThreeParams servlet that contains HTML-specific characters. Verify that this can cause malformed results. Make a variation of the servlet that filters the strings before displaying them. Be sure to copy ServletUtilities from the “forms” or “test-app” project first, so that you can simply do this:

   ```
   someString = ServletUtilities.filter(someString);
   ```

6. Make a “registration” form that collects a first name, last name, and email address. Send the data to a servlet that displays it. Feel free to modify the ThreeParams HTML form and servlet to accomplish this. Next, modify the servlet to use a default value (or give an error message) if the user omits any of the three required parameters.

7. **[Hard; for the truly inspired.]** Make a variation of the previous servlet and accompanying form, but, in this case, if the user fails to supply a value for any of the fields in the form, you should redisplay the form but with an error message saying that the value is missing. Don’t make the user reenter values that they’ve entered already. Hint: you have two main choices given the tools available to us so far: have one big servlet that generates both the HTML form and the results page (the form submits the data to itself), or two separate servlets (one that generates the HTML form and the other that generates the results page). There is an example of the first approach in the book. If you want to try the second approach, you might want to know about response.sendRedirect, which lets you redirect the browser to a specified page. For example, here is a doGet method that sends the user to www.whitehouse.gov:

   ```
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
       throws ServletException, IOException {
     response.sendRedirect("http://www.whitehouse.gov/");
   }
   ```

Exercises: Request Headers

Do whichever exercises fit your background and interests. They are ordered in approximate order of difficulty. As always, feel free to experiment on your own with other related exercises.

Make a new Eclipse project called exercises-request-headers (or some such).

1. Go to the request-headers project and copy the servlet that shows all request headers (ShowRequestHeaders). Paste it in a package of your new project. Access it from both Firefox and Internet Explorer. Use the URL http://localhost/*project-name*/show-request-headers.

2. Make a tiny Web page that contains a hypertext link to the ShowRequestHeaders servlet. To make the HTML page, you can copy hello.html from my test-app project, or copy one of the forms you had earlier. When you click to get to the ShowRequestHeaders servlet, what new header shows up that wasn't there previously?

- Note: if you are bit rusty with HTML, hypertext links look like this:
  <a href="some-address">some text to display</a>
  The browser shows the text, but takes you to the address when you click on it.

3. Some informational sites want users to access pages in a certain order. They want to prohibit users from jumping directly to a bookmarked page later in the sequence because the data on the pages may have changed since the user bookmarked the page.

Create two pages. The first page should be a normal HTML page with a link to the second page (which should be a servlet). If a user accesses the first page and then follows the link to the second page, it works normally. But, if the user directly types in the address of the second page (or follows a link from a page with a different name than the first page), they should get sent back to the first page automatically. Hints:

- Use response.sendRedirect to send users back to page1. See the description of send-Redirect on the bottom of the previous set of exercises.
- It is not necessary to make page1 be a servlet, but page2 must be.

4. Write a servlet that just says "Hello." Use a red background and a yellow foreground for Internet Explorer users; use a yellow background and a red foreground for Firefox and other non-IE users. If you are a bit rusty with HTML, you set colors as follows:
`<body bgcolor="colorName" text="colorName">` or
`<body bgcolor="#rrggbb" text="#rrggbb">`
(where r, g, and b are hex values for the red, green and blue components. I.e., #ff00ff is magenta -- 255 for red, 0 for green, and 255 for blue).

Exercises: HTTP Status Codes

Make a new Eclipse project called exercises-status-codes (or some such).

1. Write a servlet that sends about half the users to http://www.google.com and about half to http://www.bing.com. Choose at random (compare the output of Math.random() to 0.5). As a starting point, copy/paste a servlet from one of your previous projects, but remember that when you do response.sendRedirect, you should *not* do response.setContentType or out.println.

Question to ponder: Suppose you entered the address of the servlet and ended up at google.com. Now you hit the Reload button on your browser. What is the probability that you now go to bing.com?

2. Write a servlet that returns a "page not found" error page (404) unless the user supplies a favoriteLanguage request (form) parameter with a value of "Java."

3. Write a servlet that sends the first 9 requests to washingtonpost.com (an organization that uses servlets and JSP extensively, by the way), the next request to nytimes.com, and then repeats. That is, every tenth request should get sent to nytimes.com and the rest should get sent to washingtonpost.com. Don't just use probabilities (i.e., don't compare Math.random to 0.9); actually count the overall requests. So, for example, if the server received between 1000 and 1009 overall requests since last being booted, exactly 100 of them should have sent users to nytimes.com, and the rest should have sent users to washingtonpost.com.

Hint: use an instance variable (aka field, data member).

Exercises: HTTP Response Headers

1. Redo the first problem from the previous set of exercises (the one that redirects users at random). Instead of using response.sendRedirect, set the appropriate status code and response header explicitly. Do you get the same result?

2. Write a servlet that instructs the browser to reconnect every five seconds. Display the time (print `new java.util.Date()`) on each connection.

3. Write a servlet that generates an Excel spreadsheet with 10 rows and 5 columns. Display a random number in each cell. Try it from both Firefox and Internet Explorer. What happens when you hit the “Reload” button?

4. Write a servlet that returns a page to Internet Explorer users saying they will be sent to http://www.microsoft.com after 10 seconds. Send them there after that timeout. Send Firefox users to http://www.mozilla.org after the same delay. You’ll have to read about the Refresh header in the book (page 203) to see the option of supplying a specific URL to connect to.

Exercises: Cookies

1. In a previous problem, you had two pages. You used the Referer header to force users to get to page 2 by following a link from page 1. Now, suppose that it is not necessary to *always* go to page 1 before page 2; you just want to make sure that users have been to page 1 *at least once* before they are allowed to visit page 2. For example, page 1 might be a page that introduces the site and gives some legal disclaimers, and users are required to visit that page at least once before they can see the second page.

Create two servlets. If a user visits page 2 before having ever visited page 1, they should get redirected to page 1. **Note:** some people find this problem easier if they manually delete cookies between tests. To do this in IE 5/6, start at the Tools menu, then select Internet Options, General, and Delete Cookies. With IE 7, start at the Tools menu, then select Internet Options, General, and click “Delete” under “Browsing history”. With Firefox, click on Tools, then Options, then Privacy, then Cookies—you’ll have various options from there.

2. Write a servlet that displays the values of the firstName, lastName, and emailAddress request parameters. But, remember what users told you in the past, and use the old values if the current values are missing. So, if a parameter is missing and the client is a first-time visitor, have the servlet list “Unknown” for the missing values. If a parameter is missing and the client is a repeat visitor, have the servlet use previously-entered values for the missing values.

3. Make a small page that displays some simple information of your choosing. Make another page that lets users choose the foreground and background color that they want the first page to use. For example, if users never visit the the color choice page, the main informational page should use default colors of your choosing. But if a user visits the color choice page and chooses a yellow background and red foreground, all subsequent visits to the main page by that user should result in red on yellow. There is no need to vet the colors; you can accept whatever color values the user gives you.

If you are a bit rusty with HTML, you set colors as follows:

```
  <BODY BGCOLOR="colorName" TEXT="colorName">
```
or
```
  <BODY BGCOLOR="#RRGGBB" TEXT="#RRGGBB">
```
(where R, G, and B are hex values for the red, green and blue components. I.e., #FF00FF is magenta -- 255 for red, 0 for green, and 255 for blue).

4. Repeat exercise number one, but with session cookies, not persistent cookies. Visit it twice in IE, then click on the IE icon *on the desktop* to start a new browser. Is this considered the same session or a new one? Repeat the process with Firefox. Notice that they work differently.

5. Check out your cookie files in IE and Firefox. On IE, start at the Tools menu, then do Internet Options, General, Settings, View Files. It is easier to find the cookie files if you do “Delete Files” first. With Firefox, do a search (“Find”) for a file called cookies.txt in a Mozilla folder. See if you notice a cookie from doubleclick.net in both cases. Take a look at the structure of these files to see if you can figure out how they represent the various cookie parameters.

Exercises: Session Tracking

1. Use session tracking to make a servlet that says "Welcome Aboard" to first-time visitors (within a browsing session) and "Welcome Back" to repeat visitors. If you did this same task earlier with the Cookie API, was this servlet harder or easier than the equivalent version using cookies explicitly?

2. Write a servlet that displays the values of the firstName, lastName, and emailAddress request parameters. But, remember what users told you in the past, and use the old values if the current values are missing. So, if a parameter is missing and the client is a first-time visitor, have the servlet list "Unknown" for the missing values. If a parameter is missing and the client is a repeat visitor, have the servlet use previously-entered values for the missing values. This should definitely be easier than a version that uses cookies explicitly.

3. Make a servlet that prints a list of the URLs of the pages that the current user has used to link to it (within the current browsing session). That is, if one user has followed hypertext links from three different pages to the servlet, the servlet should show the user the URLs of those three pages. Test out your servlet by making a couple of different static Web pages that link to it. If you feel inspired, modify the basic approach from the notes so that you do not store repeated entries; if the same user follows a link from page1 to the servlet twice, the servlet should list the URL of page1 only once in the list. If you are unfamiliar with the list-related data structures in Java, see note at bottom regarding the ArrayList class.

4. **[Medium hard, for the moderately inspired.]** The previous problem tracked the referring page and ignored repeated entries. In this problem, you should track a request parameter and count the repeats. Make a servlet that keeps track of the number of each item being ordered. For example, if the user orders yacht, car, book, yacht, the servlet should show something like this:
   - yacht (2)
   - car (1)
   - book (1)

   To simplify your code, you can just use the item name as sent from the HTML form; there is no need to check each name against a table of legal names as in the much-more-complicated Shopping Cart example in the book.

Note: Using the ArrayList Class

The java.util.ArrayList class is useful for keeping lists of items when you don't know how many items you will have. It has several key methods. Here is a summary that assumes that you are storing entries of type String. Change <String> to <YourType> if you store something other than Strings.

- Constructor: no required arguments. E.g.
  ```
  List<String> items = new ArrayList<String>();
  ```
- Adding items to end of list: call "add". E.g.
  ```
  items.add("Item One");
  ```
- Looping down the list
  ```
  for(String item: items) {
    doSomethingWith(item);
  }
  ```
- Number of items in list: call "size". E.g.
  ```
  int numItems = items.size();
  ```
- Get an item out of a specific location: "get". E.g.
  ```
  String item = items.get(0);
  ```
- Testing if an item is already in the list: "contains". E.g.
  ```
  if (items.contains("Item One")) ...
  ```

Exercises: JSP Intro

1. Make a new Eclipse project, grab OrderConfirmation.jsp from the “jsp-intro” project, and copy it to the WebContent folder of your new project. Also copy the JSP-Styles.css file that OrderConfirmation.jsp uses. Compare the time it takes to access it the first time to the time it takes on subsequent requests.
2. Look at the source code for OrderConfirmation.jsp. See if you can figure out how to make it output “Thanks for ordering a yacht.” instead of “Thanks for ordering null.”. Hint: you do not need to make *any* changes to the JSP page.
3. If you wanted a servlet to build a Web page that displayed an image, where would you put the image and how would you refer to it from the servlet? How does this differ from JSP? (This question is trickier than it sounds at first.)
4. Make a very simple static HTML page. If you want, download one from the Web or use one from a previous exercise. Rename it to have a .jsp extension. Have an HTML form with nothing but a SUBMIT button that directs the user to the static page. Now, change the form to use POST instead of GET. Why does it work both times? (Answer: the auto-generated code can’t just be in the doGet method. Where *does* the auto-generated code go, then?)

Exercises: JSP Scripting Elements

1. Make a JSP page that randomly selects a background color for each request. Just choose at random among a small set of predefined colors. Be sure you do not use the JSP-Styles.css style sheet, since it overrides the colors given by
<BODY BGCOLOR="...">.

Note: in Eclipse, *all* Java code goes in src/*subdirectory-matching-packageName*, regardless of whether the Java code is a servlet or some other kind of Java class. Then, if you want a class called ColorUtils with a static method called randomColor and have it be in a package called utils, you might write something like this:

```
package utils;

public class ColorUtils {
  public static String randomColor() {
    if (Math.random() < 0.5) {
      return("BLUE");
    } else {
      return("RED");
    }
  }
}
```

In JSP, you can call the method with
`<%= utils.ColorUtils.randomColor() %>`
Also, if you want to test some Java code separately from a JSP page, note that in Eclipse, if your class has a “main” method, you can right-click anywhere in the file, choose Run As --> Java Application, and Eclipse will run main and put the output in the console at the bottom.

2. Make a JSP page that lets the user supply a request parameter indicating the background color. If no parameter is supplied, a background color should be selected at random.

3. Make a JSP page that lets the user supply a request parameter indicating the background color. If no parameter is supplied, the most recently used background color (from a previous request by *any* user) should be used.

Exercises: The page Directive

1. Use JSP to make an Excel spreadsheet where the first row says "Year," "Apples," and "Oranges." It should then have two rows of data (2008 and 2009), where each entry is a random number between 0 and 10. I.e. the result should look something like this:

Year	Apples	Oranges
2008	9.23456	3.98765
2009	4.45678	2.223344

If you choose to use the tab-separated data approach (which is probably best here), be careful because Eclipse might change your tabs to spaces automatically. If so, you can edit the file outside of Eclipse using a normal text editor, or you can do Window --> Preferences --> General --> Editors --> Text Editors --> Insert spaces for tabs.

2. Make an Excel spreadsheet with a random number of rows.

3. The java.math package has a class called BigInteger that lets you create whole numbers with an arbitrary number of digits. Create a JSP page that makes a large BigInteger from a String you supply as a request parameter, squares it, and prints out the result. Use the online API at http://java.sun.com/javase/6/docs/api/ to see the syntax for the BigInteger constructor and squaring operations.

4. Make a JSP page that sleeps for 20 seconds before returning the page. (Call TimeUnit.SECONDS.sleep(20) inside a try/catch block that catches InterruptedException). Access it "simultaneously" from Firefox and Internet Explorer. Repeat the experiment with the JSP page not allowing concurrent access. Verify the slower result.

5. **[Very hard; for the truly inspired.]** The above example worked because our version of Tomcat implements isThreadSafe="false" by queueing up the requests and passing them one at a time to the servlet instance. But, that is not the only legal implementation. Create a test case that will definitively show which of the following three approaches a server uses for isThreadSafe="false":

- Keeps a single servlet instance and queues up the requests to it
- Makes a pool of instances but lets each instance only handle one request at a time
- Ignores isThreadSafe altogether

Note that *all three* approaches have been represented by Tomcat versions in the past.

6. **[Just for fun.]** Grab the ComputeSpeed and SpeedError pages from the page-directive project. Access the ComputeSpeed page with numeric values for the "furlongs" and "fortnights" form parameters attached to the end of the URL. (See page 366 if you want more detail). Now, supply a non-numeric value for one of the parameters. Next, supply a legal number for furlongs, but 0 for fortnights. Can you explain the unexpected result you get?

Exercises: Including Files and Applets

Problem #1 is quite easy, and is the most important one of the group: it is how you most commonly use jsp:include. Problems 2 and 3 look at more advanced but much less commonly used features of jsp:include.

1. Make an HTML “signature” block with your name and email address. Include it in two JSP pages.

2. The value of the page attribute of jsp:include is allowed to be a JSP expression. That is, you are permitted to do this:

```
<jsp:include page="<%= someVariable %>"/>
```

Use this idea to make a JSP page that includes a “good news” message or a “bad news” message at random.

3. Suppose that you have two different JSP pages that do two different things. However, for both pages you want to let the user supply a bgColor attribute to set the background color of the page. Implement this, but use an include mechanism to avoid repeating code. For example:
White background: http://host/path/page1.jsp
White background: http://host/path/page2.jsp
Red background: http://host/path/page1.jsp?bgColor=RED
Yellow background: http://host/path/page2.jsp?bgColor=YELLOW

For testing, I do not care if you write an HTML form to collect the bgColor parameter or if you simply attach it onto the end of the URL “by hand.”

4. Make two separate JSP pages that have bulleted lists containing random integers in a certain range. Avoid repeating code unnecessarily by including a page that defines a randomInt method.

5. If you are familiar with applets, make a trivial one that does nothing but set the background color to blue and print a string derived from the MESSAGE parameter embedded in the HTML by means of a PARAM element. Convert it to a version that uses the Java Plug-In. Note that, if this runs at all, it proves that you are correctly accessing the Plug-In. You don’t need to use Swing or Java2D to verify that you are using the Plug-In, since the tag generated by jsp:plugin is incompatible with the standard virtual machine used by Firefox and IE. Try both Firefox and Internet Explorer to see which (if any) of them has the Plug-In installed. Reminder: applets run on the client, not on the server. So your applet’s .class files can’t go in the server’s WEB-INF/classes directory. These .class files work the same way as for regular applets: they go in the same directory as the JSP/HTML file that uses the applet tag. This is nothing specific to JSP, but is just the normal way applets work.

Exercises: Beans

For these exercises, avoid *any* Java code in the JSP pages.

1. Define a class called ColorBean that stores strings representing a foreground color and a background color. Compile and test it in a standalone manner (i.e., without using a servlet or JSP page). Remember that in Eclipse, if your class has a “main” method, you can right-click anywhere in the file, choose Run As --> Java Application, and Eclipse will run main and put the output in the console at the bottom.

Also, note that Eclipse has a very nice shortcut for creating getter and setter methods in beans. Start by writing a class like this:

```
package myPackage;

public class ColorBean {
  private  String foregroundColor = "BLACK";
  private  String backgroundColor = "WHITE";
}
```

Then, R-click in the class, go to the Source sub-menu, and choose “Generate Getters and Setters”. You can then have Eclipse automatically create getForegroundColor, setForegroundColor, getBackgroundColor, and setBackgroundColor.

2. Make a “color preference” form that collects the user’s preferred foreground and background colors. Send the data to a JSP page that displays some message using those colors. This JSP page should use a default value for any form value that the user fails to supply (but don’t worry about empty strings). So, for example, if the user goes directly to the JSP page (bypassing the form), the JSP page should still work fine. For now, don’t worry about the user sending you whitespace; just handle totally missing values.

3. Redo the color preference example, but if the user fails to supply either of the colors, use whatever value they gave last time. If you have no previous value, use a default. (Hint: this problem is almost exactly the same difficulty as the previous one.)

4. Redo the color preference example, but if the user fails to supply any of the parameters, use whatever color the most recent user gave last time. Why could this give bad results?

5. Fix problem #2 so that it also handles whitespace. Do so without adding any explicit Java code to the JSP page.