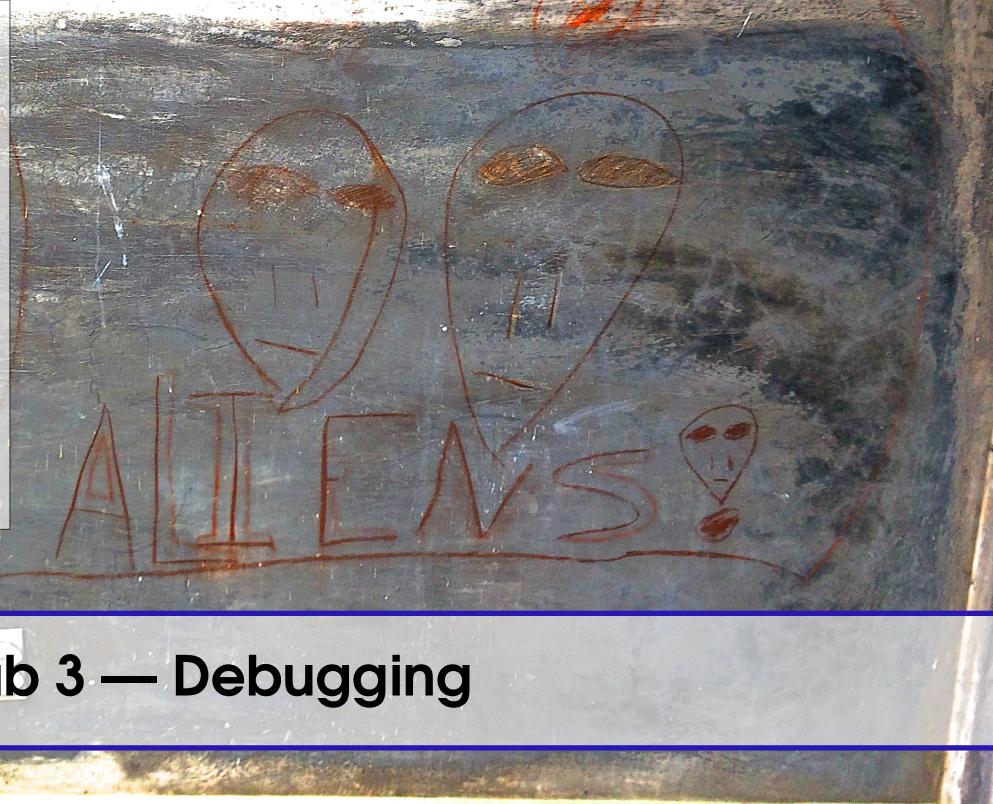


Of Moths, Wolves and Rubber Ducks  
 About Bugs  
 Ducks and Wolves: Two Basic Strategies for Debugging  
**Lab 3 Classroom Work**  
 Debugging Java with NetBeans Debugger  
 Debugging Web Pages using Chrome Developer Tools  
**References**



## 3 — Lab 3 — Debugging

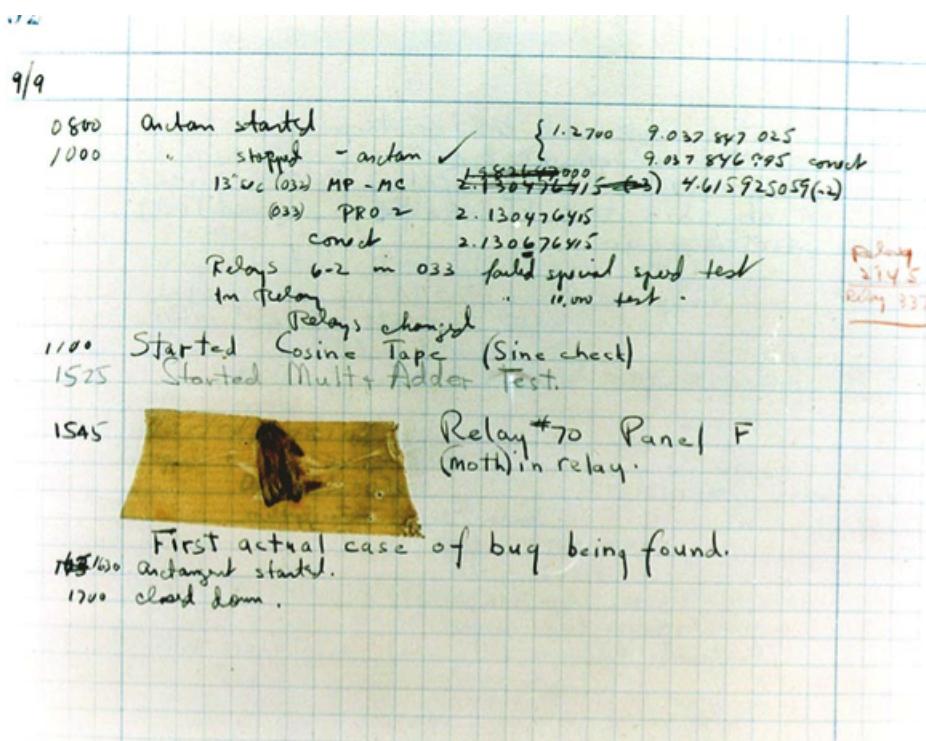


Figure 3.1: Admiral Hopper's famous bug.

### 3.1 Of Moths, Wolves and Rubber Ducks

Debugging is the term used in the software industries to describe the process of locating and correcting faults in software. It is a term that comes from outside of software engineering, and its use goes back to at least Thomas Edison's days. One of the first documented uses of the word to describe the process of debugging a computer goes back to Rear Admiral Grace Hopper's use to describe pulling a moth out of a computer circuit in 1947; she actually found a moth in the

computer hardware. Her famous log entry is depicted in Figure 3.1.

Bugs can be divided into two broad categories. Syntax bugs, by far the easiest to catch, are mistakes in the "spelling" of code; compilers are generally very good at finding these types of bugs. Semantic bugs, conceptual errors in the code, are much more difficult to find. Fortunately there are tools and techniques that we can familiarize ourselves with to aid in debugging these types of bugs.

This lab is designed to introduce you to some debugging techniques, and introduce you to two debugging tools: a code debugger, and a web page debugger.

## 3.2 About Bugs

Research has been conducted into software bugs, their causes and types, and effective means of removing and preventing them. Research indicates that knowing about the nature of bugs helps debugging. Researchers at Yale identified seven common causes of bugs committed by beginning programmers<sup>1</sup>. Some of the errors identified include: off-by-one errors (looping once too few or many times), plan-dependency problems (putting code in the wrong part of a complicated structure), and negation and whole part problems (forgetting to apply DeMorgan's theorem when needed).

Research has shown that learning about bugs can help new programmers debug better.

## 3.3 Ducks and Wolves: Two Basic Strategies for Debugging



Figure 3.2: Your bug is a howling wolf.

<sup>1</sup> McCauley, et. al., "Debugging: a review of the literature from an educational perspective." *Computer Science Education*, Vol. 18 No. 2, June 2008, 67-92

There are two basic strategies that can help you debug your software. The first is a debugging algorithm developed by Edward Gauss, not surprisingly, at the University of Alaska<sup>2</sup>. Applying this algorithm results in your bug becoming isolated, an important step in any debugging process (Figure 3.3). He states it like this:

Pretend your bug is a howling wolf and your code is the entire territory of Alaska.

Then:

1. Let A be the territory known to contain the wolf (initially all of Alaska).
2. Construct a fence across A, along any convenient natural line that divides A into B and C.
3. Listen for the howls; determine if the wolf is in B or C.
4. Go back to Step 1 until the wolf is contained in a tight little cage.

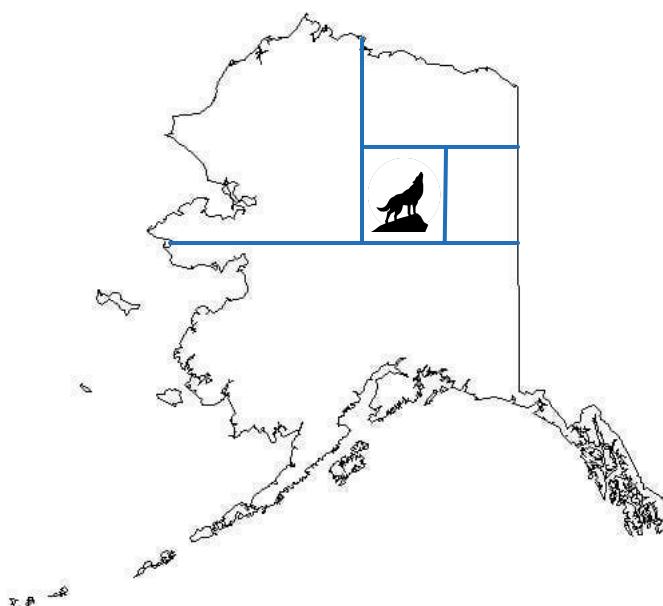


Figure 3.3: I can find that bug in 5 partitions!

[howling wolf icon: wolf: <http://www.clipartbest.com/clipart-bcyEGKeMi>]  
[alaska map: <http://geography.about.com/library/blank/blxusak.html>]

A strategy called “Rubber Ducking” (Figure 3.4) is based on the premise that you have a rubber duck with you at all times when programming. You should pause occasionally in your coding (which you should be doing every hour or so anyway), and explain clearly (rubber ducks need things to be kept simple) what it is you are doing and why you’re doing it. The concept is that this helps you keep your goal in mind as you proceed with your coding (why was I writing this code?). Losing track of the coding goal is a common reason for bugs.

## 3.4 Lab 3 Classroom Work

### 3.4.1 Debugging Java with NetBeans Debugger

Most beginning programmers learn the technique of printing out variable values from within their program to determine what the cause of a bug might be. While this is a tried and true

<sup>2</sup>Gauss, E., “The ‘Wolf Fence’ algorithm for debugging.” Communications of the ACM, Vol. 25 No. 11, Nov. 1982, 780—



Figure 3.4: Ducks eat bugs.

method, it is time consuming. Over the years, debugging tools have developed to help ease the debugging process.

For this example, you will use the NetBeans debugger. This debugger has most of the tools that any other debugging programming might contain, and is a good introductory debugger to work with.

1. Open NetBeans. Create a new Java Application project named DebugExample.
2. Browse to <http://csc412fsu.com/~csc412/JavaExample/DebugExample.java> and copy the code into the project you created. Make sure that the package and class names correctly match your project.

```
run:  
 1 2 3 4 5 6 7 8 9 10 **  
 2 3 4 5 6 7 8 9 10 **  
 3 4 5 6 7 8 9 10 **  
 4 5 6 7 8 9 10 **  
 5 6 7 8 9 10 **  
 6 7 8 9 10 **  
 7 8 9 10 **  
 8 9 10 **  
 9 10 **  
 10 **  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.5: Desired Output from DebugExample program.

3. The desired output of the program is depicted in Figure 3.5. Run the program. The output you get should be different, and in fact the run should generate a runtime error. We will now attempt to find the problem using the NetBeans debugger.
4. If you examine the output that was generated during your run, you can see that the program is trying to work, there are at least some numbers printing and some asterisks.

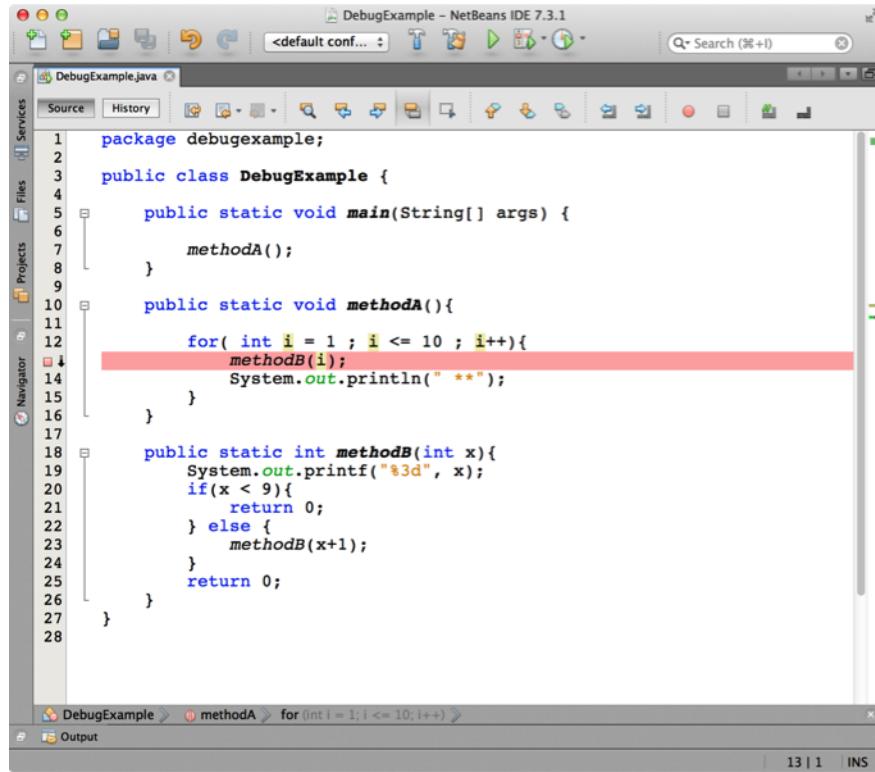


Figure 3.6: Breakpoint set on line 13.

5. Set a breakpoint at line 13 in the source code. When debugging a program, a breakpoint is where the program pauses and allows you to inspect what is going on in the program. To set a breakpoint, click on the 13 in the bar of numbers. It should then appear as depicted in Figure 3.6

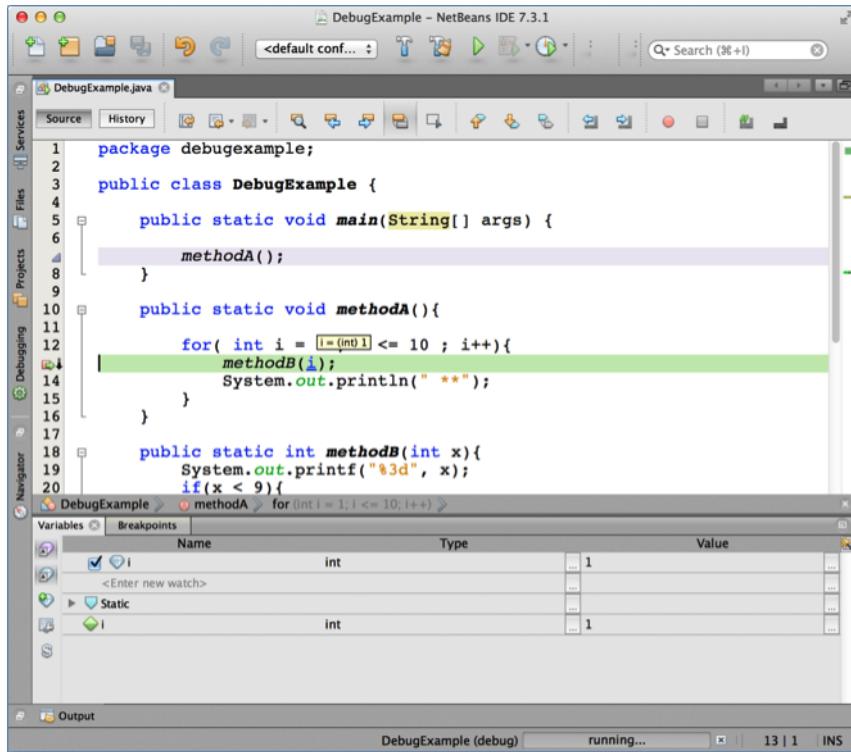


Figure 3.7: The value of i at the first break.

6. Run the program in debug mode. To do this from the menu select **Debug** **> Debug Main Project** (or either **⌘-F5** or **ctrl-F5** in Windows). Your program should now be paused at line 13. If you hover your cursor over the variable i in line 13, you should see a value pop up, indicating the current value of i. You can also see i's current value in the variables window on the bottom (Figure 3.7).
7. Now, examine what happens the next time around in the loop. To see this you can to "continue" program execution. To do this, from the menu select **Debug** **> Continue** (or **F5** on mac or in Windows). Notice how the value of i changes. Click Continue Execution a number of times until the value of i reaches 5.
8. Examine the output window. You should see 4 rows of output text.
9. A feature among many debuggers is the ability to change the value of a variable while the program is running (in debug mode). In the variables pane, click on the button with an ellipsis (three dots) to the right of i, and enter the value 3.

The Output window displays the following text:

```
debug:
1 **
2 **
3 **
4 **
```

Figure 3.8: Output after changing the loop control variable.

10. Continue execution of the program through two more loops (until i is 5), and examine the out window's contents. It should appear if the loop backed up two steps (Figure 3.8). This is, of course, exactly what happened since you modified the value of the loop control

variable in step 9.

11. Another feature that most debuggers offer is the ability to allow the program to run one line of code a time, pausing after each line. This is called stepping, of which there are three types:

**Step Over** (`Debug > Step Over` or `F8`) - Executes the highlighted line of code.

If the line contains a method call;, it runs the method without descending into the method, pausing at each statement in the method.

**Step Into** (`Debug > Step Into` or `F7`) - Executes the highlighted line of code. If the line contains a method call, the debugger descends into the method, and pauses on the first line in the method.

**Step Out** (`Debug > Step Out` or `⌘-F7` / `ctrl-F7`) - Executes the highlighted line.

If the line is in a method, the debugger continues executing the remaining lines in the method, returns from the method, then pauses in the location that the method was called.

12. Keep stepping (10 or 20 times) through the program using step over and step into (remember to step over when you get to the `println` statement). Observe what is happening in the variables window, and observe what is getting executed each time you seem to "loop". Remember, stepping into means that you descend into the method. What type of method is `methodB` (hint: `methodB` calls itself)?
13. Using the debugger, set the value of `x` to 0 before you enter `methodB` again, keep stepping into the method, observing what happens. When you have an idea what is happening, you can terminate the program in the debugger (`Debugger > Finish Debugger Session`).
14. Can you figure out what may be wrong with the program? If you still can't figure it out using the debugger, scroll down to the answer in the references section at the end of the lab.
15. Correct the error, then run the program normally, and observe the output. It should appear as in Figure 3.5.

### 3.4.2 Debugging Web Pages using Chrome Developer Tools

During Lab 2, many people had difficulties with getting the CSS to cooperate. the tools demonstrated here can help you debug web pages much more easily than just adjusting values in your CSS and reloading the page ( the web equivalent to printing and running your program code).

Most web browsers now provide tools to allow you to understand how the browser interprets and renders a web page and why. Google's Chrome browser provides Google's Dev Tools for this purpose. In this part of the lab you will modify your class page's CSS using Dev Tools, but this is just the beginning. Dev Tools provide many powerful tools that significantly aid in web page development. You are encouraged to experiment with and do further reading about Dev Tools.

1. Open Chrome. If you do not have Chrome installed on your computer, go to <http://google.com/chrome>, download and install it.
2. Go to your about webpage (e.g. <http://csc412sfsu.com/~csc412/about.html>)

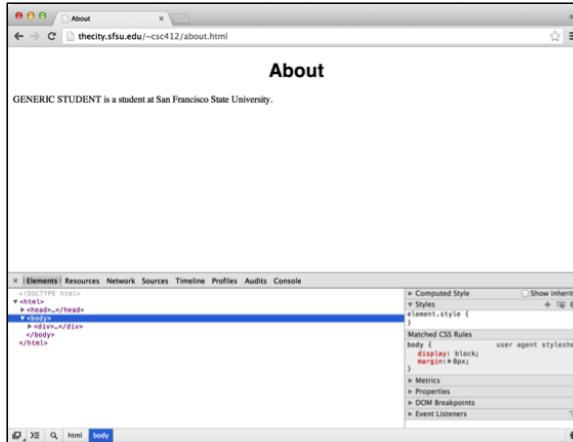


Figure 3.9: Chrome with Developer Tools enabled

3. Open the developer tools (**View** **>** **Developer** **>** **Developer Tools** or **Option - ⌘-i** on Mac, or **Tools** **>** **Developer Tools** or **ctrl-↑-i** in Windows). Panes should appear at the bottom of the page and should appear as in Figure 3.9.
4. Move your cursor up and down (no clicks) over the elements in the lower-left pane. You should observe different sections of your web page being highlighted. What you are seeing is a correlation between what HTML element corresponds to what part of your page.
5. The right hand pane contains information about the current state of your page, including the current styles in use in your page. Look through the different information that is available to you.

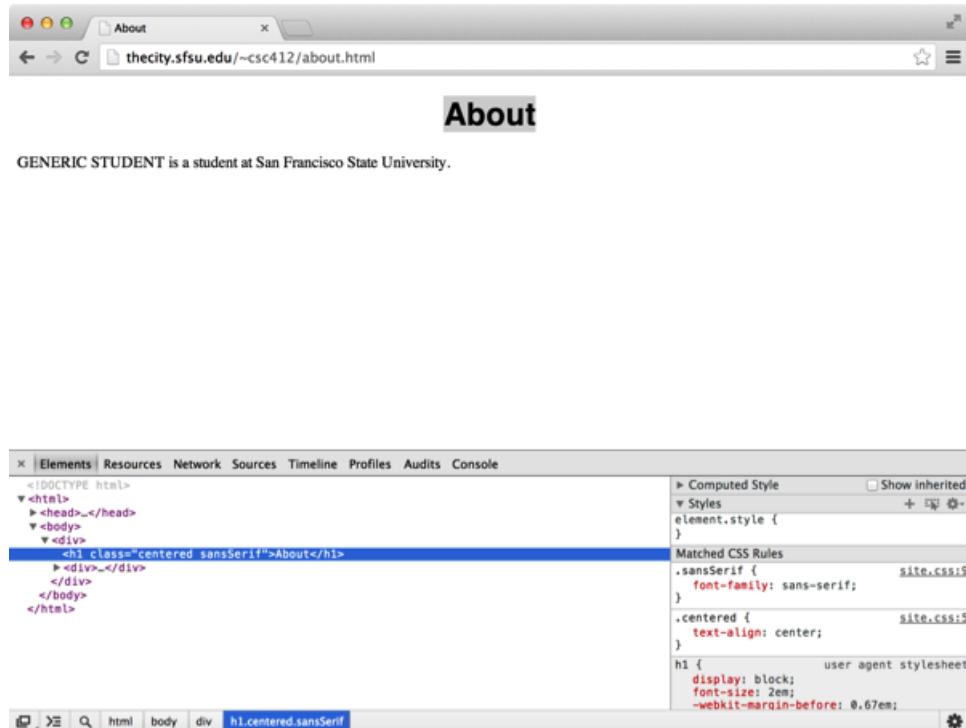


Figure 3.10: Inspecting an element of a webpage in Chrome

6. In the upper pane, right click on the word About, and select Inspect Element. You should see the About title's html element selected on the lower-left pane, and the matching CSS

rules in the lower-right pane Figure 3.10.

- If you mouse over the line containing fontfamily under Matched CSS Rules, a checkbox will appear. Click on the checkbox so the box is cleared, and notice what happens to your title. Click on it a few times, leaving it in the checked state.

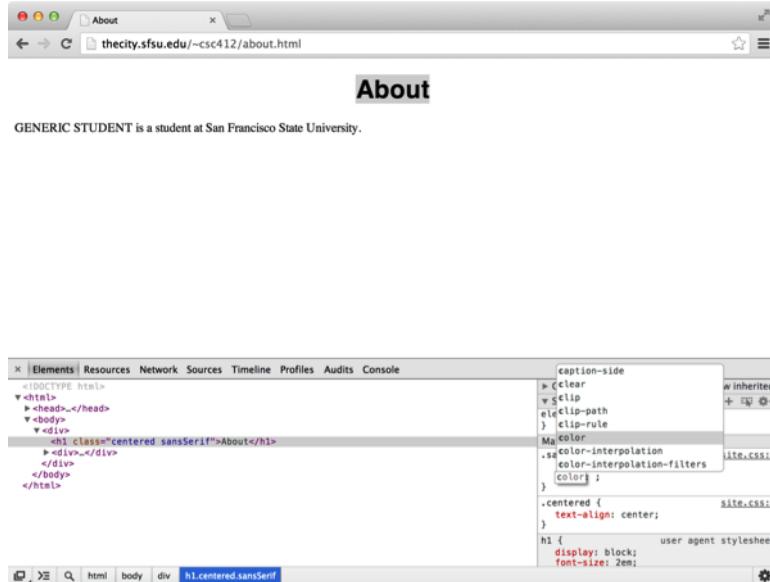


Figure 3.11: Selector for CSS properties

- Click on the close curly brace below the font-family line. A blank line should appear. Type c, and wait a moment. Much like an IDE, a list of CSS properties beginning with the letter ‘c’ should appear. Use the cursor to select color, then press . Next pressing (tab) should move your cursor to the right of a colon next to color. Type r, and wait for the possible colors available. Select red, and press . Notice what happens to your title.
- In the Matched CSS portion of the pane, you will see a link in the upper-right corner, indicating something like site.css:9. Click this link. The Sources tab should open in the lower-left pane, showing site.css.
- In the site.css tab in the sources pane, you can actually see the CSS file that was loaded. Below the color: red; property line, type:
- The modifications that you have made to the site.css cannot simply be saved back to your server with the current configurations. However, you can save the file to your local file system. Right click in the site.css tab pane, and observe that you can save the file to your local system. You can then upload your site.css file to your local file system if you choose.
- Before moving on to the next step, show your changes to the TA and/or Instructor and get their signature.
- Press the refresh button. Your page should revert to its original state.

### 3.5 References

Good starting points for debugging, paying special attention to the references at the end:

- [http://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](http://en.wikipedia.org/wiki/Rubber_duck_debugging)
- <http://en.wikipedia.org/wiki/Debugging>
- <http://upload.wikimedia.org/wikipedia/commons/8/8a/H96566k.jpg>

#### Answer to #14

What you were observing here was a recursive method with no base case. `methodB` kept calling itself again and again, with a larger and larger value for `x`, eventually causing the stack to overflow (remember stack frames?). By checking instead if `x > 9`, `x = 10` becomes the base case, and causes the recursion to terminate.