Chris Piech
CS 106A

Handout #9
January 24, 2018

Debugging

Thanks to Eric Roberts, Nick Parlante and Mehran Sahami for portions of this handout.

Much of your time as a computer programmer will likely be spent debugging. This phenomenon is best described by a quotation from one of the first computer pioneers, Maurice Wilkes:

> *As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*
>
> — Maurice Wilkes, 1949

In order to be better prepared to undertake the more complex future debugging that you will be doing, we aim to give you here both a sense of the philosophy of debugging as well as to teach you how to use some of the practical tips that make testing and debugging easier.

The Philosophy of Debugging

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. To a large extent, the problems that people face debugging programs are not so much technical as they are psychological. To become successful debuggers, you must learn to think in a different way. There is no cookbook approach to debugging, although Nick Parlante's 11 Truths of Debugging (given below) will probably help. What you need is insight, creativity, logic, and determination.

As computer scientists, it is important to remember that the programming process leads you through a series of tasks and roles:

Task	—	***Role***
Design	—	Architect
Coding	—	Engineer
Testing	—	Vandal
Debugging	—	Detective

These roles require you to adopt distinct strategies and goals, and it is often difficult to shift your perspective from one to another. Although debugging can often be very difficult, it can be done. It will at times take all of the skill and creativity at your disposal, but you can succeed if you are methodical and do not give up on the task.

Debugging is an important skill that you will use every day if you continue in Computer Science or any related field. Even though it is the final task of those listed above, it is certainly not the least important. You should always plan ahead and allow sufficient time for testing and debugging, as it is required if you expect to produce quality software. In addition, you should make a concentrated effort to develop these skills now, as they will be even more important as programs become more complicated later in the quarter.

The 11 Truths of Debugging

1. Intuition and hunches are great—you just have to test them out. When a hunch and a fact collide, the fact wins. That's life in the city.
2. Don’t look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable producing extremely simple and obvious errors from time to time. Look at code critically—don’t just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.
3. The clue to what is wrong in your code is in the values of your variables and the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.
4. Be systematic and persistent. Don’t panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.
5. If you code was working a minute ago, but now it doesn’t—what was the last thing you changed? This incredibly reliable rule of thumb is the reason your section leader told you to test your code as you go rather than all at once.
6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable in an experiment at a time. It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs.
7. If you find some wrong code that does not seem to be related to the bug you were tracking, fix the wrong code anyway. Many times the wrong code was related to or obscured the bug in a way you had not imagined.
8. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions that led you to find a bug. Alternately, if you have a bug but can’t pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your functions cannot contain the bug. One of these arguments will contain a flaw since one of your functions does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.
9. Be critical of your beliefs about your code. It’s almost impossible to see a bug in a function when your instinct is that the function is innocent. Only when the facts have proven without question that the function is not the source of the problem should you assume it to be correct.
10. Although you need to be systematic, there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the functions you suspect the most first. Good instincts will come with experience.
11. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. *Debugging* code is more mentally demanding than *writing* code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. You cannot debug when you are not seeing things clearly. Many times a programmer can spend hours late at night hunting for a bug only to finally give up at 4:00 A.M. The next day, they find the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the “go do something else for a while, come back, and find the bug immediately” scenario happens too often to be an accident.

— Nick Parlante, Stanford University

Using a Debugger

Because debugging is a difficult but nonetheless critical task, it is important to learn the tricks of the trade. The most important of these tricks is to get the computer to show you what it is doing, which is the key to debugging. The computer, after all, is there in front of you. You can watch it work. You cannot ask the computer why it is not working, but you can have it show you its work as it goes. Modern programming environments like Eclipse come equipped with a **debugger,** which is a special facility for monitoring a program as it runs. Using the debugger helps you build up a good sense of what your program is doing, and often points the way to the mistake.

To illustrate the operation of the Eclipse debugger in as concrete a way as possible, let us look at how we might us the debugger to find bugs in the `GuessYourNumber.java` program shown in Figure 2 below. As the bug icon indicates, the program is buggy. As a programmer, it is your job to figure out why. The remainder of this handout describes the techniques we might use to look for bugs with the help of the Eclipse debugger.

Tip: follow along! The Eclipse project containing the `GuessYourNumber.java` program is on the course website, alongside this handout in the "Handouts" tab.

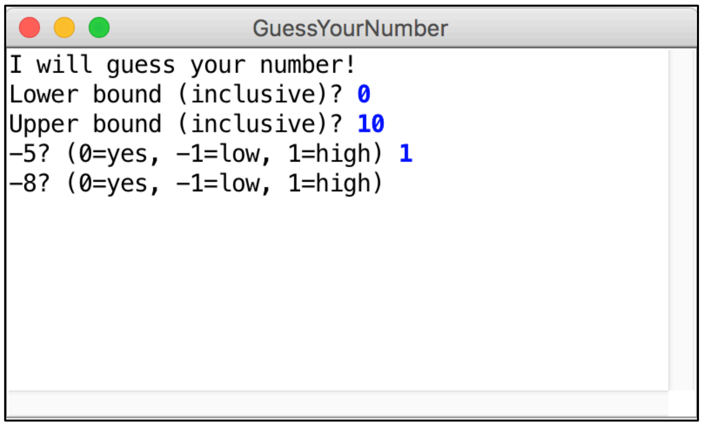
Figure 2. Buggy program intended to guess the user's number

```
/*
 * File: GuessYourNumber.java
 * ---------------------
 * This program attempts to guess what number you are thinking of.
 * It asks for the range of your number, and then proceeds to make
 * guesses until it narrows in on your number!
 */

import acm.program.*;

public class GuessYourNumber extends ConsoleProgram {
   public void run() {
      println("I will guess your number!");
      int lowest = readInt("Lower bound (inclusive)? ");
      int highest = readInt("Upper bound (inclusive)? ");
      int answer = findNumber(highest, lowest);
      println("Ha!  I knew your number was " + answer + "!");
   }

   /*
    * Returns the final computer guess after narrowing down the
    * range of possible numbers the user is thinking of.  Takes
    * as parameters the initial bounds of the user's number.
    */
   private int findNumber(int lowerBound, int upperBound) {
      while (lowerBound != upperBound) {
         int guess = (upperBound - lowerBound) / 2;

         // Update our bounds depending on the result
         int check = readInt(guess + "? (0=yes, -1=low, 1=high)");
         if (check == -1) {
            lowerBound = guess + 1;
         } else if (check == 1) {
            upperBound = guess - 1;
         } else {
            lowerBound = guess;
            upperBound = guess;
         }
      }

      return lowerBound;
   }
}
```

The general idea for this program is to ask the user for the range containing their secret number, and then keep narrowing in the range until we end up with one number that must be the user's guess. The program will always guess the *midpoint* of the current range of possible numbers.

Assessing the symptoms

Before we start using the debugger, it is always valuable to run the program to get a sense of what the problems might be. If we run it with a secret number of 3, we might see:



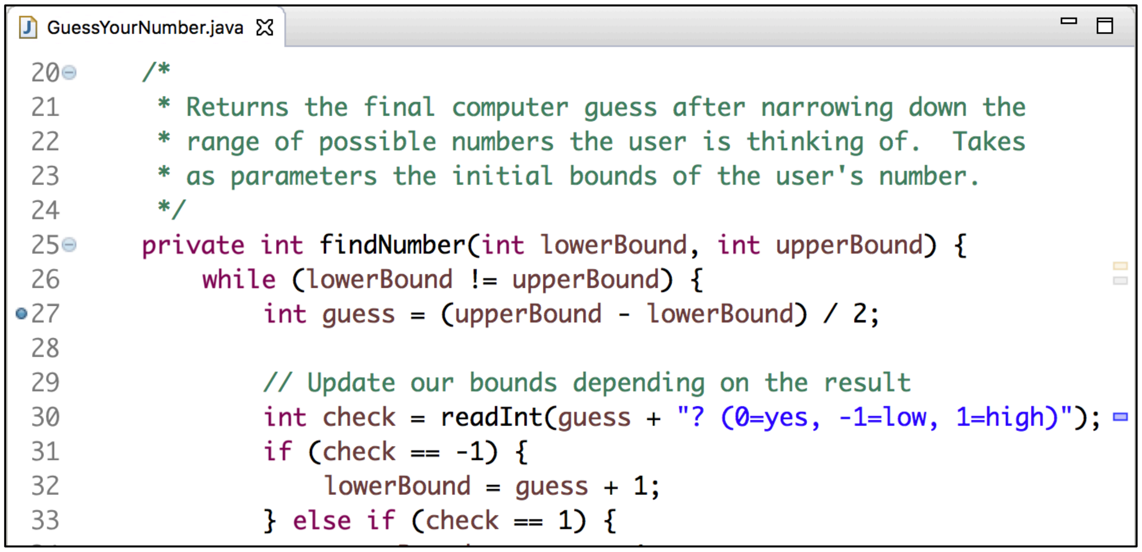
```
I will guess your number!
Lower bound (inclusive)? 0
Upper bound (inclusive)? 10
-5? (0=yes, -1=low, 1=high) 1
-8? (0=yes, -1=low, 1=high)
```

Even though the program is "running," the results do not look so good. The problem is that we told the program that our secret number is between 0 and 10, but the program immediately guesses -5. Something is definitely wrong.

The Eclipse Debugger

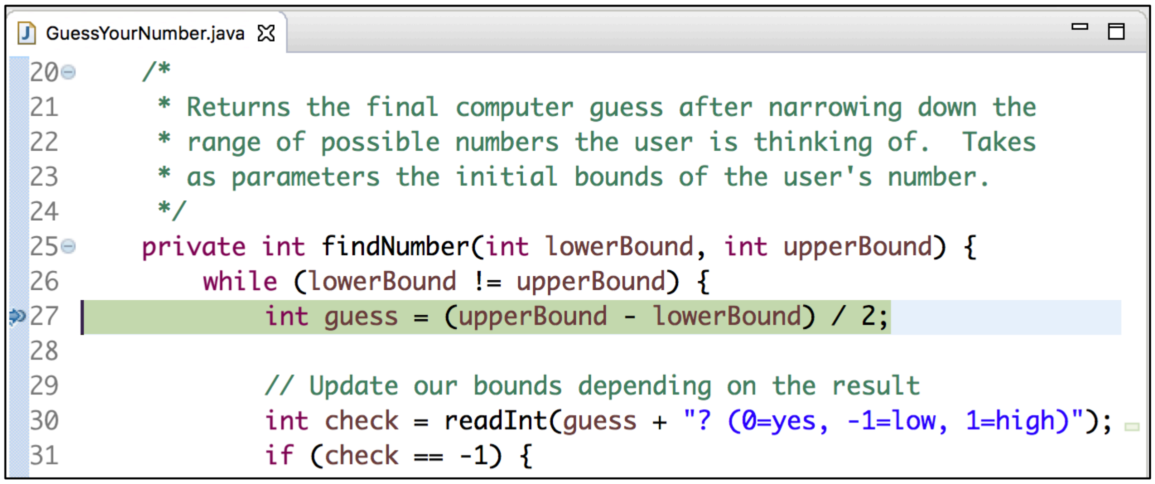
One of the easiest ways to figure out what the program is doing is to use the Eclipse debugger. When you run a project under Eclipse, you can use the debugger to set breakpoints in your code, which tells the debugger to pause on that line and await further instructions. This enables you to step through the program one step at a time, examine variables, and do other useful things.

Debugging, however, is a creative enterprise, and there is no magic technique that tells you what to do. You need to use your intuition, guided by the data that you have. In the **GuessYourNumber** program, intuition is likely to suggest that the problem has something to do with coming up with a guess. Thus, we might set a **breakpoint** on line 21, inside the **while** loop, so that the program will stop there. To do so, double-click in the just barely gray margin at that line, at which point a small circle appears indicating that there is now a breakpoint on that line, as shown:



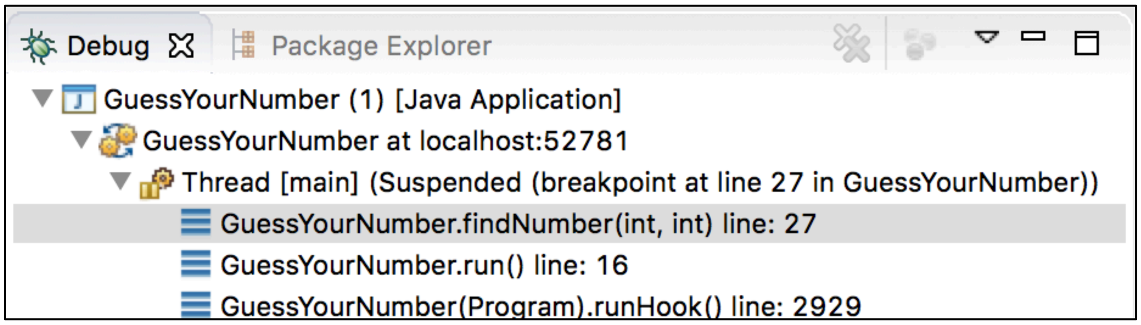
```
20    /*
21     * Returns the final computer guess after narrowing down the
22     * range of possible numbers the user is thinking of.  Takes
23     * as parameters the initial bounds of the user's number.
24     */
25    private int findNumber(int lowerBound, int upperBound) {
26        while (lowerBound != upperBound) {
27            int guess = (upperBound - lowerBound) / 2;
28
29            // Update our bounds depending on the result
30            int check = readInt(guess + "? (0=yes, -1=low, 1=high)");
31            if (check == -1) {
32                lowerBound = guess + 1;
33            } else if (check == 1) {
```

Once we have set the breakpoint, our next step is to run the program as normal, with the running person icons. The program will start and ask us to enter our secret number's bounds. Suppose that we again enter **0** and **10**. At that point, the program enters the **`while`** loop, where it stops at the breakpoint as shown below. (Note that you may first see a dialog box which says "**This kind of launch is configured to open the Stanford Debugger perspective when it suspends. Do you want to open this perspective now?**" If you do see this dialog, go ahead and click "Yes".) The arrow and the highlight mark the line of code the debugger is *about to execute*.


GuessYourNumber.java

```
20     /*
21      * Returns the final computer guess after narrowing down the
22      * range of possible numbers the user is thinking of.  Takes
23      * as parameters the initial bounds of the user's number.
24      */
25     private int findNumber(int lowerBound, int upperBound) {
26         while (lowerBound != upperBound) {
27             int guess = (upperBound - lowerBound) / 2;
28 
29             // Update our bounds depending on the result
30             int check = readInt(guess + "? (0=yes, -1=low, 1=high)");
31             if (check == -1) {
```

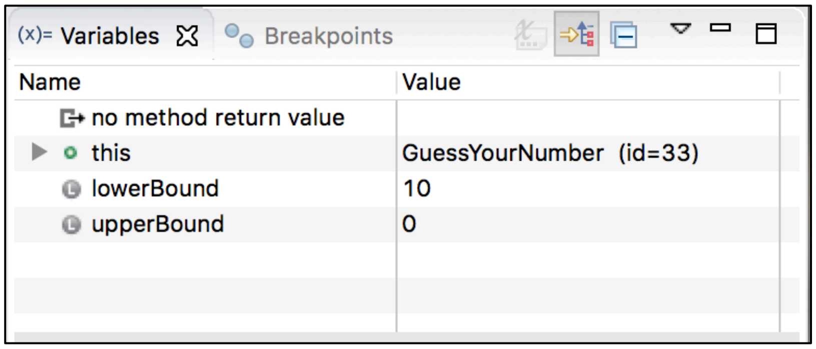
There is a lot more information displayed in Eclipse's debugging perspective as well. The "Debug" pane at the upper left, for instance, shows the execution history of the program.



```
GuessYourNumber (1) [Java Application]
  GuessYourNumber at localhost:52781
    Thread [main] (Suspended (breakpoint at line 27 in GuessYourNumber))
      GuessYourNumber.findNumber(int, int) line: 27
      GuessYourNumber.run() line: 16
      GuessYourNumber(Program).runHook() line: 2929
```

These lines tell us where we are in the execution. We are currently at line 27 of **`findNumber`**, which was called from line 16 of **`run`**. Each of these constitutes a **stack frame**, as described in the text.

The pane on the upper right is the "Variables" pane, which allows us to see the current values of variables in the current stack frame:

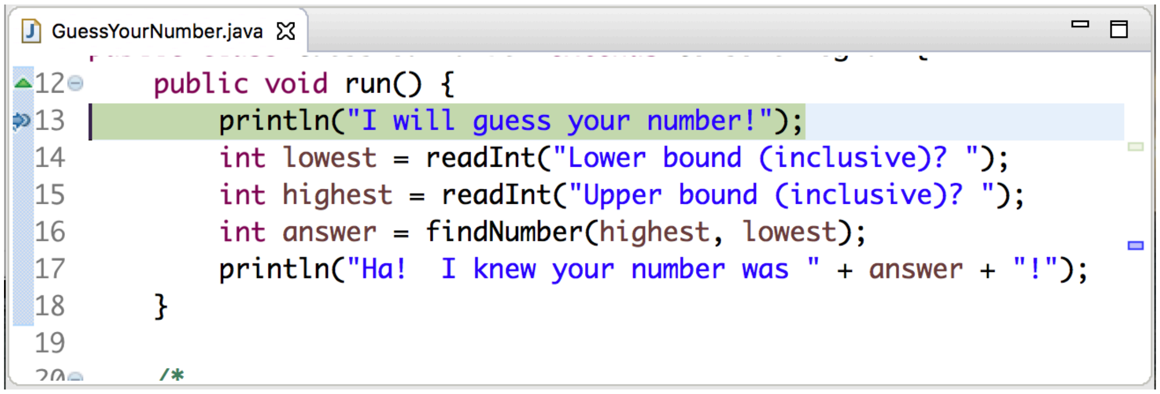


Name	Value
no method return value	
this	GuessYourNumber (id=33)
lowerBound	10
upperBound	0

In this frame, the local variables are simply the parameters to **`findNumber`**. We entered the bounds 0 and 10, so those are stored here. But – wait a minute – the lower and upper bounds seem to be flipped. Why does this frame show the lower bound as 10 and the upper bound as 0? Clearly the computer is doing something wrong.

In point of fact, that diagnosis—tempting as it sometimes is for all of us—is almost certainly incorrect. The computer is almost certainly doing exactly what we told it to do. The problem is that the programmer has done something wrong. The task now is to find out what that is. The problem is obviously earlier in the program than intuition might suggest – perhaps we are storing the bounds incorrectly – so it is necessary to go back and insert an earlier breakpoint.

We can now go back and stop the **`GuessYourNumber`** program, either by clicking in its close box or by clicking on the red square in the Debug window. Double click in the margin on line 13, the first line of **`run`**, to put a breakpoint there. If we debug the program again, it will soon stop in the following state:



```
12     public void run() {
13         println("I will guess your number!");
14         int lowest = readInt("Lower bound (inclusive)? ");
15         int highest = readInt("Upper bound (inclusive)? ");
16         int answer = findNumber(highest, lowest);
17         println("Ha!  I knew your number was " + answer + "!");
18     }
19
20     /*
```

Finding the critical clues

As always in debugging, our primary goal is to figure out what the program is doing rather than why is it not doing what we wanted. To do so, we need to gather as much information as possible, and Eclipse offers great tools for doing so. The most useful tools at this point are the various controls that appear along the top of the Debug window, of which the following are the most important:

Resume. Continues the program from where it last stopped, either because of hitting a breakpoint or because the user clicked **Suspend**.

Suspend. Stops the program immediately as if it had hit a breakpoint.

Terminate. Exits from the program entirely.

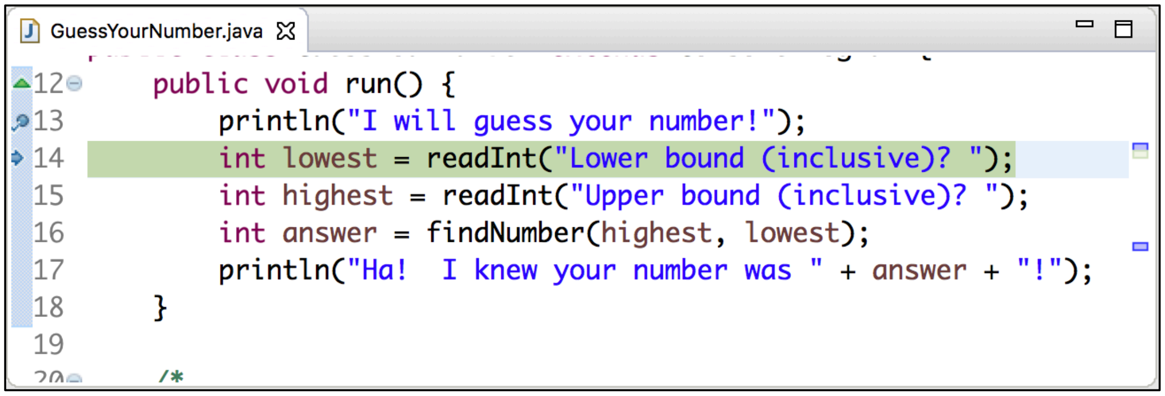
Step Into. Executes one statement of the program and then stops again. If that statement includes a method call, the program will stop at the first line of that method. As noted below, this option is not as useful as **Step Over**.

Step Over. Executes one statement of the program at this level and then stops again. Any method calls in the statement are executed through to completion unless they contain explicit breakpoints.

Step Return. Continues to execute this method until it returns, after which the debugger stops in the caller (the method that called the current method).

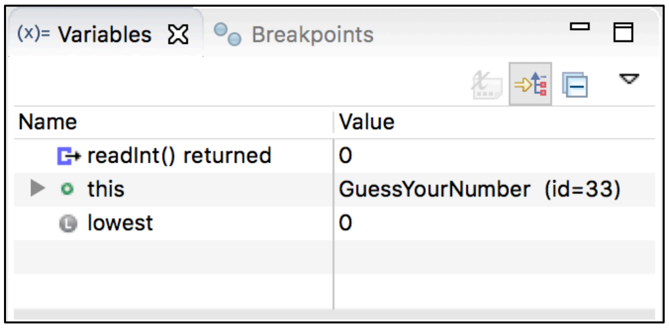
The three stepping options are extremely useful, but you need to take some care in choosing which one to use. In most cases, **Step Over** is the right option to use. Intuitively, it has the effect of continuing to the next step at this method level, allowing you to stay at the same conceptual level of abstraction. If the current statement calls one of your own methods, **Step Into** may be exactly what you want, because that will allow you to debug that subsidiary method if necessary. The danger with **Step Into** arises when the current statement contains calls to library methods such as **`println`**. In such cases, **Step Into** will try to step through the code for those methods, which can be extremely confusing.

We would like to step through each line of the **`run`** method to make sure we are handling the bounds correctly. Therefore, **Step Over** is the best choice. If we select that, **`println`** will execute, followed by the debugger stopping at the next line:


GuessYourNumber.java

```
12     public void run() {
13         println("I will guess your number!");
14         int lowest = readInt("Lower bound (inclusive)? ");
15         int highest = readInt("Upper bound (inclusive)? ");
16         int answer = findNumber(highest, lowest);
17         println("Ha!  I knew your number was " + answer + "!");
18     }
19
20     /*
```

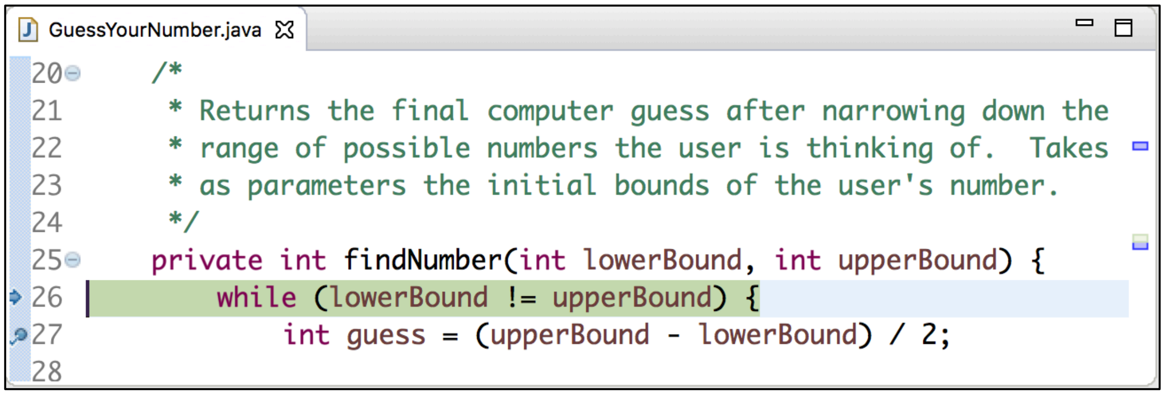
Click **Step Over** once more to execute the line asking the user for the lower bound, and notice that the debugger does not pause on the next line. Why is this? **`readInt`** has not finished executing! Back in the program window, the program is now prompting us for a number. Remember that **`readInt`** executes until we enter a number and press ENTER. If we enter the lower bound **0** followed by ENTER, the debugger will now pause on line 15. If we take a look at the Variables pane, things look ok so far; the variable lowest is correctly storing the value **0** that we just entered.



Name	Value
readInt() returned	0
this	GuessYourNumber (id=33)
lowest	0

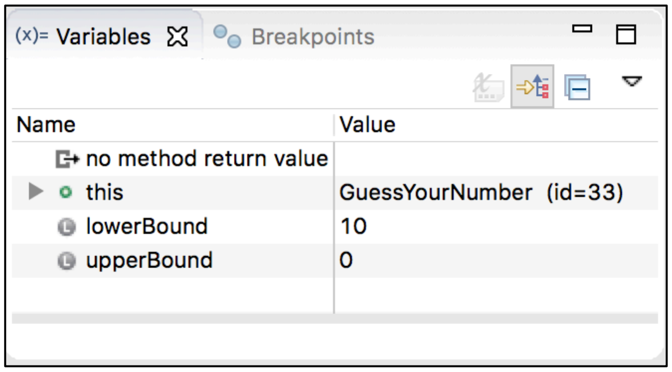
Press **Step Over** once more, enter the upper bound **10**, and the debugger should now pause on line 16. If we take a look at the Variables pane again, the variable **`highest`** correctly has value 10.

This all looks correct; so what gives? It might be that the issue is in calling **`findNumber`**. We would like to now step the debugger *into* the method **`findNumber`** to execute each line there one at a time. To do this, with the debugger stopped at the line where **`findNumber`** is called, click **Step Into** to go into the **`findNumber`** method. The debugger should now be paused on the first line of **`findNumber`**, line 26. (if we had instead clicked **Step Over**, the debugger would have executed the entire **`findNumber`** method, and paused again only once it reaches line 17 – the next line in the *same method* - which is not what we want).



```
20    /*
21     * Returns the final computer guess after narrowing down the
22     * range of possible numbers the user is thinking of.  Takes
23     * as parameters the initial bounds of the user's number.
24     */
25    private int findNumber(int lowerBound, int upperBound) {
26        while (lowerBound != upperBound) {
27            int guess = (upperBound - lowerBound) / 2;
28
```

Here, we find something interesting; if we look at the Variables pane, we notice that the lower and upper bound values have swapped!

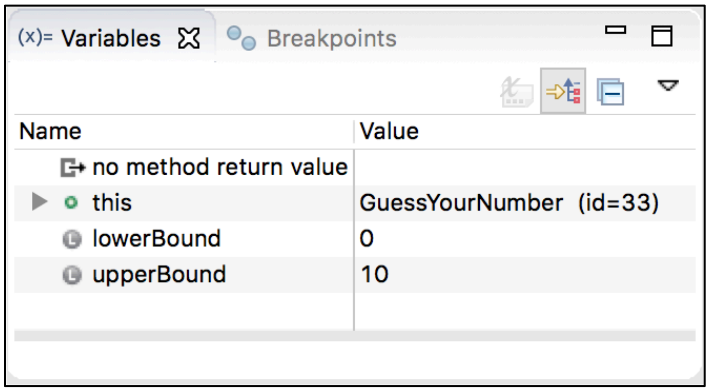


Name	Value
no method return value	
this	GuessYourNumber (id=33)
lowerBound	10
upperBound	0

That is strange; they were just correct a moment ago in the **`run`** method…but very soon, the problem will jump out. The values in run were stored correctly, but they were passed as parameters in the wrong order to **`findNumber`**! **`findNumber`** expects the first parameter value to be the *lower* bound, and the second parameter to be the *upper* bound. It is easy to realize this because we have a descriptive comment and variable names for **`findNumber`**. So we need to rewrite line 16 in **`run`** as follows:

```
int answer = findNumber(lowest, highest);
```

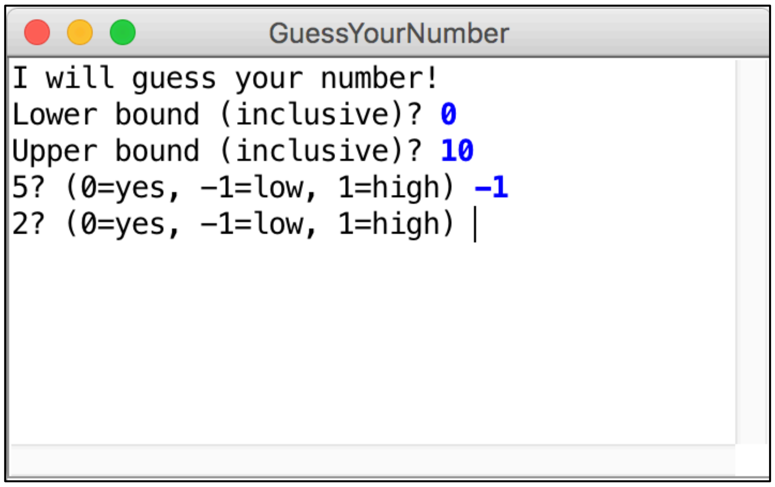
Now, after making this change, an important next step is to *immediately confirm* that this change fixes the issue we saw. An easy way to do this is to remove all existing breakpoints (double-click on a breakpoint dot to remove it) and add a new one at line 26, which is the start of **`findNumber`**. This way, we can instantly verify that **`findNumber`** has the correct bounds values. Run the program with bounds values **0** and **10** once more, and….



Name	Value
no method return value	
this	GuessYourNumber (id=33)
lowerBound	0
upperBound	10

Yes! The bounds values are correct. Good job, bug-hunter!

Now, let us try running the program without any breakpoints, with the same bounds of **0** and **10**. Initially, it looks good! The program's first guess is **5**, which is exactly what we expect given that the program always guesses the midpoint of the current bounds. Continue, assuming our secret value is **8**. Enter **-1** to indicate that 5 is too low. However, the next guess is *lower*, which does not make sense since we just told the program that 5 was already too low.

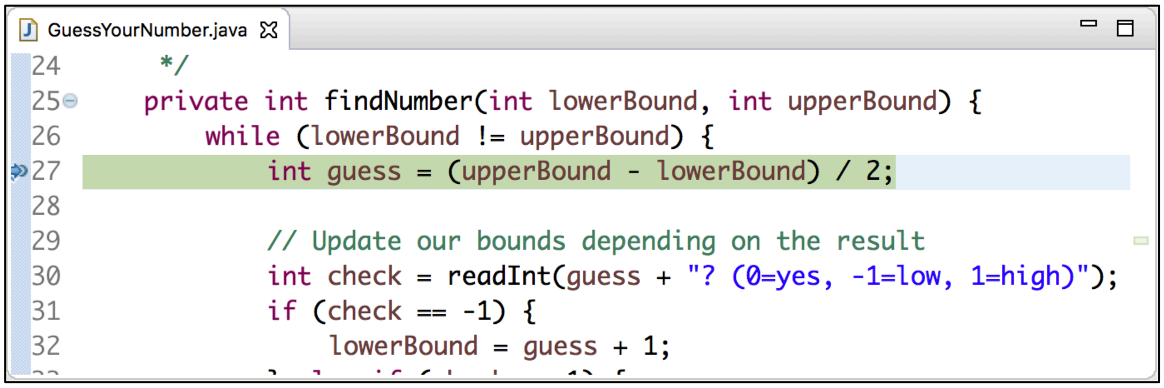


```
I will guess your number!
Lower bound (inclusive)? 0
Upper bound (inclusive)? 10
5? (0=yes, -1=low, 1=high) -1
2? (0=yes, -1=low, 1=high)
```

Alright, time to put on our debugger hats once more.

From the previous bug we fixed, we know that **`findNumber`** is receiving the correct bounds values. This helps us narrow down the problem to be somewhere in **`findNumber`**. This is an important tactic; do your best to *narrow down* the possible place(s) in your program where the bug could be coming from. If you can eliminate entire blocks of code or even entire methods by confirming they work correctly, it will go a long way in helping you track down the issue.

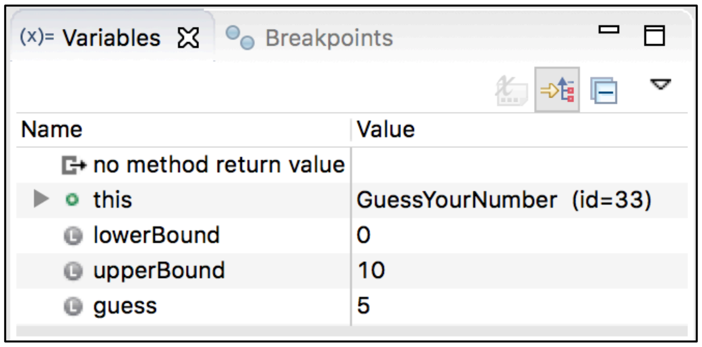
With this in mind, a reasonable guess for the location of this bug might be line 27, where the program calculates its next guess. Put a breakpoint there and run the program with the same bounds of **0** and **10**. The debugger should stop on the line like so:



GuessYourNumber.java

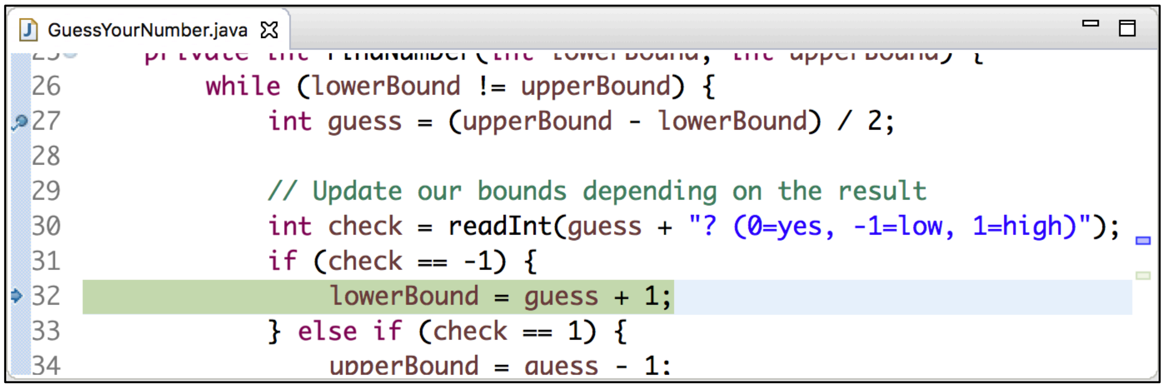
```
24        */
25     private int findNumber(int lowerBound, int upperBound) {
26         while (lowerBound != upperBound) {
27             int guess = (upperBound - lowerBound) / 2;
28
29             // Update our bounds depending on the result
30             int check = readInt(guess + "? (0=yes, -1=low, 1=high)");
31             if (check == -1) {
32                 lowerBound = guess + 1;
```

Now, we can step through each line of **`findNumber`** to find out what is going on. If we press **Step Over**, the debugger jumps to line *30*, since it steps to the next line of code (it skips comments and blank space). In the Variables pane, we can see that **`guess`** has now been created with value **5**, which we previously said was correct.



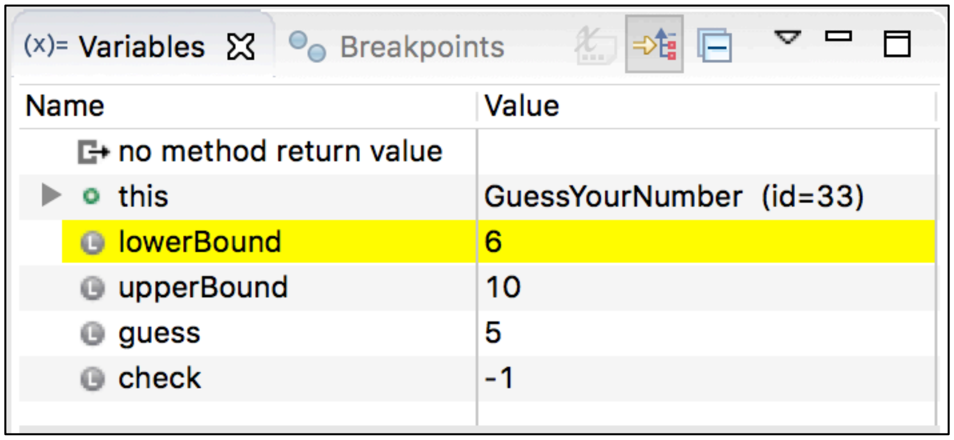
Name	Value
no method return value	
this	GuessYourNumber (id=33)
lowerBound	0
upperBound	10
guess	5

If we click **Step Over** once more, the debugger will wait for us to enter a response in the program window. For a secret number of 8, we enter **-1** to indicate that the guess 5 is too low. The debugger will now pause at the **`if`** statement on line 31. From the Variables pane, we see that **`check`** has value **-1**, which is correct. So if we press **Step Over** now, the debugger will enter the body of this **`if`** statement because the condition **`check == -1`** evaluates to **`true`**. This is a neat benefit of the debugger; we can see *exactly* the flow of our program through our loops and conditionals.


GuessYourNumber.java

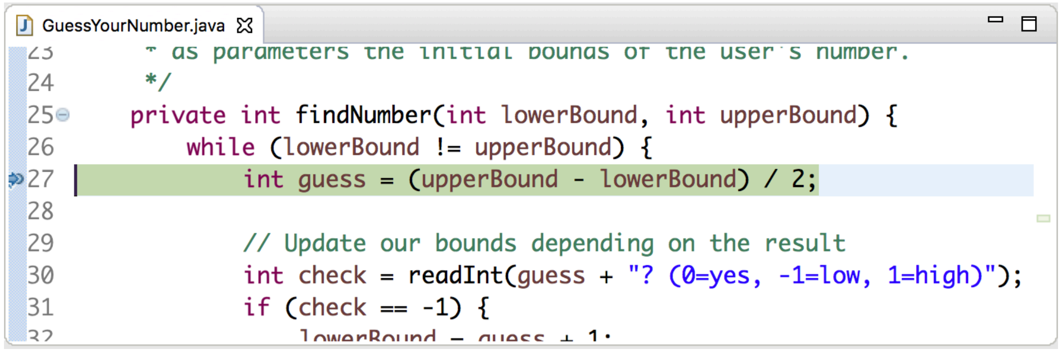
```
25      private int findNumber(int lowerBound, int upperBound) {
26          while (lowerBound != upperBound) {
27              int guess = (upperBound - lowerBound) / 2;
28
29              // Update our bounds depending on the result
30              int check = readInt(guess + "? (0=yes, -1=low, 1=high)");
31              if (check == -1) {
32                  lowerBound = guess + 1;
33              } else if (check == 1) {
34                  upperBound = guess - 1;
```

This line that we are about to execute makes sense; if the program's guess was too low, then the *smallest* that the user's secret number can be is **`guess + 1`**. Press **Step Over** to confirm that **`lowerBound`**'s value is indeed updated to 6; the debugger even highlights the change in yellow in the Variables pane!



Name	Value
no method return value	
this	GuessYourNumber (id=33)
lowerBound	6
upperBound	10
guess	5
check	-1

At this point, the debugger is paused on line 33 with the **`else if`** statement. However, if we press **Step Over** once more, the debugger will jump back up to the **`while`** loop statement. The reason for this is, since the **`if`** statement condition evaluated to **`true`**, we evaluate the **`if`** statement body and then *skip* the **`else if`** and the **`else`**. Press **Step Over** again to once again stop on line 27, where the next guess is calculated.


GuessYourNumber.java

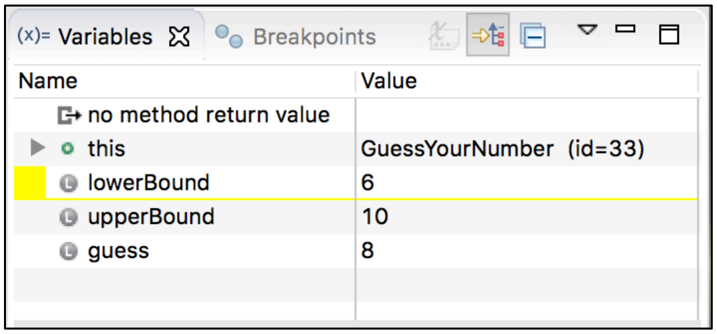
```
23       * as parameters the initial bounds of the user's number.
24       */
25      private int findNumber(int lowerBound, int upperBound) {
26          while (lowerBound != upperBound) {
27              int guess = (upperBound - lowerBound) / 2;
28
29              // Update our bounds depending on the result
30              int check = readInt(guess + "? (0=yes, -1=low, 1=high)");
31              if (check == -1) {
32                  lowerBound = guess + 1;
```

From the Variables pane, we can see that our bounds values are correct: **6** and **10**. But if we press **Step Over** and check the value of the newly-created variable **`guess`** (remember **`guess`** goes away at the end of each time around the **`while`** loop, and is re-created again later because of its scope), we see that it has value **2**! In this case, the way in which the difficulty of seeing the bug manifests itself has to do with thinking too literally about the expressions as we have written them, seeing them as what we *want* them to mean as opposed to what they in fact *do* mean. We want the next guess to be the midpoint of the lower and upper bounds. But the equation on line 27 finds half the *difference* between the two bounds, which only calculates the correct value when the lower bound is 0. This is why the first guess seemed ok! But once we say that 5 is too low, the lower bound changes to **6** and this equation no longer calculates reasonable values. If we instead take the *average* of the two bounds, we should always get the correct value. Thus, line 27 should be:

```
int guess = (upperBound + lowerBound) / 2;
```

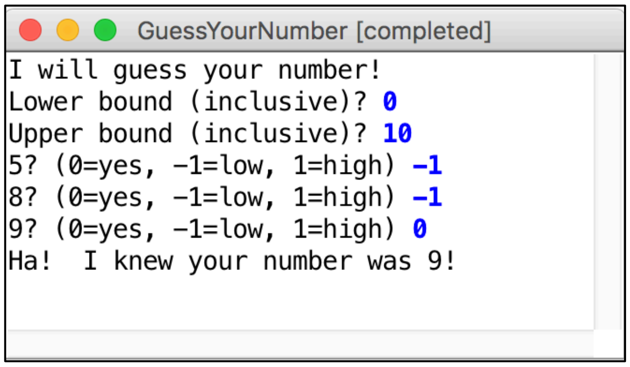
We just touched on another important debugging tactic. Make sure that you can *explain why the proposed change will fix the problem* before you make the change. Do not simply make changes to see if they fix the problem; you should know beforehand what the outcome will be. Otherwise, you risk making unnecessary modifications and introducing additional bugs.

Go ahead and terminate the program, leaving the breakpoint on line 27 in, and run it once more to confirm that our change fixed the issue. Enter **0** and **10** as the bounds, after which the debugger should pause on line 27. This time, we have already stepped through the next lines, so click the green **Resume** button; this tells the debugger to keep executing until it hits our breakpoint again. The program should keep running; pretend our secret number is 9, and enter a **-1** to indicate 5 is too high. The debugger again hits our breakpoint and pauses; this time, press **Step Over** to see what the value of guess is.



Name	Value
no method return value	
this	GuessYourNumber (id=33)
lowerBound	6
upperBound	10
guess	8

8! Hooray! This is exactly what we expect, since this is the midpoint of the new bounds **6** and **10**. And sure enough, if we remove the breakpoint, press **Resume**, and continue playing with 9 as our mystery number, the program should continue making reasonable guesses until it narrows in on 9.



```
I will guess your number!
Lower bound (inclusive)? 0
Upper bound (inclusive)? 10
5? (0=yes, -1=low, 1=high) -1
8? (0=yes, -1=low, 1=high) -1
9? (0=yes, -1=low, 1=high) 0
Ha!  I knew your number was 9!
```

Note: one step command this tutorial did not touch on was **Step Return**. **Step Return** tells the debugger to continue executing the method it is currently in, and pause at the line that *called* this method. As a concrete example, if we have a breakpoint somewhere in **`findNumber`** and we press **Step Return**, the debugger would continue executing the rest of **`findNumber`**, and pause again on line **16** (which has already been executed), which is where this method was called from. Essentially, as opposed to **Step Into**, which tells the debugger to go *down* a level in your code, **Step Return** tells the debugger to go *up* a level.

Wrapping Up

One of the most common failures in the debugging process is inadequate testing. After making the corrections described in the preceding sections, you could run this program for some time, not see that anything is amiss, and assume that it is functioning correctly. However, you should *always* test each part of your code rigorously to ensure that this is the case as much as possible. That being said, there is no strategy that can guarantee that your program is ever bug free. Testing helps, but it is important to keep in mind the caution from Edsger Dijkstra that "testing can reveal the presence of errors, but never their absence." By being as careful as you can when you design, write, test, and debug your programs, you will reduce the number of bugs, but you will be unlikely to eliminate them entirely.

Additional Eclipse Tips

Once you have completed using the debugger and want to return to using Eclipse in the regular "Editor" perspective (the one that you have been using most of the time), you can simply go to the "**Stanford Menu**" and pick the "**Editor**" selection. This also works in the other direction; if you would like to view the debug perspective, simply pick the "**Debugger**" selection instead.

Additionally, if you close out of some of the Eclipse panes and cannot get them back, go to the "**Stanford Menu**" and click "**Reset UI**". This will restore Eclipse's user interface to the default view, showing all original panes.