Mehran Sahami
CS 106A

Handout #33
November 5, 2007

Debugging

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

Much of your time as a computer programmer will likely be spent debugging. This phenomena 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. I've heard that Diet Coke™ helps too, but this may be anecdotal…

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 don't 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 an online 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's doing, which is the key to debugging. The computer, after all, is there in front of you. You can watch it work. You can't ask the computer why it isn't 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. By using the Eclipse debugger, for example, you can step through the operation of your program and watch it work. 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's look at how we might us the debugger to find bugs in the `Roulette.java` program shown in Figure 2 below. As the bug icons indicate, the program is buggy. As a programmer, it is your job to figure out why. The remainder of this handout describes the techniques you might use to look for bugs with the help of the Eclipse debugger.

Figure 2. Buggy program intended to play a simplified form of roulette

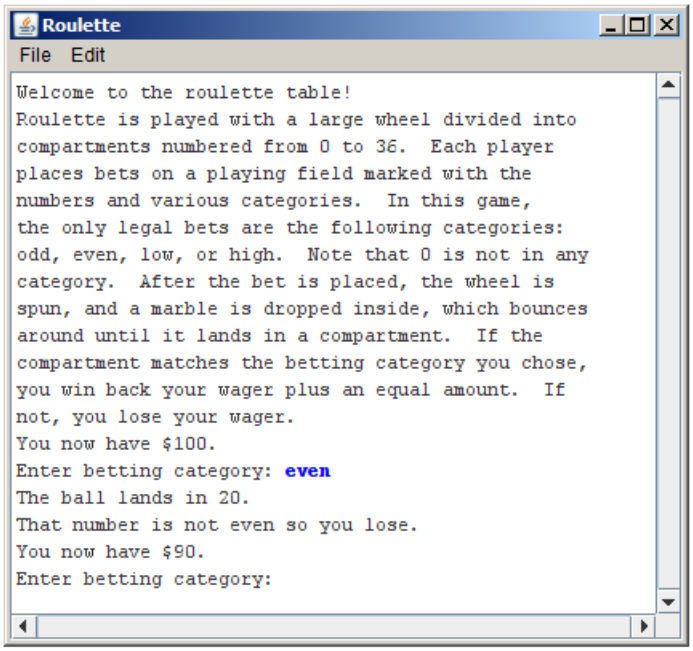
```
/*
 * File: Roulette.java
 * -------------------
 * This program simulates a small part of the casino game of
 * roulette.
 */

import acm.program.*;
import acm.util.*;

public class Roulette extends ConsoleProgram {

/** Amount of cash with which the player starts */
   private static final int STARTING_MONEY = 100;

/** Amount wagered in each game */
   private static final int WAGER_AMOUNT = 10;

/** Runs the program */
   public void run() {
      giveInstructions();
      playRoulette();
   }

/**
 * Plays roulette until the user runs out of money.
 */
   private void playRoulette() {
      int money = STARTING_MONEY;
      while (money > 0) {
         println("You now have $" + money + ".");
         String bet = readLine("Enter betting category: ");
         int outcome = spinRouletteWheel();
         if (isWinningCategory(outcome, bet)) {
            println("That number is " + bet + " so you win.");
            money += WAGER_AMOUNT;
         } else {
            println("That number is not " + bet + " so you lose.");
            money -= WAGER_AMOUNT;
         }
      }
      println("You ran out of money.");
   }
```

Figure 2. Buggy program intended to play a simplified form of roulette (continued)

```
/**
 * Simulates the spinning of the roulette wheel.  The method
 * returns the number of the slot into which the ball fell.
 */
   private int spinRouletteWheel() {
      println("The ball lands in " + rgen.nextInt(0, 36) + ".");
      return rgen.nextInt(0, 36);
   }

/*
 * Returns true if the outcome matches the category specified
 * by bet.  If the player chooses an illegal betting
 * category, this function always returns false.
 */
   private boolean isWinningCategory(int outcome, String bet) {
      if (bet == "odd") {
         return outcome % 2 == 1;
      } else if (bet == "even") {
         return (outcome % 2 == 0);
      } else if (bet == "low") {
         return (1 <= outcome && outcome <= 18);
      } else if (bet == "high") {
         return (19 <= outcome && outcome <= 36);
      } else {
         return (false);
      }
   }

/**
 * Welcomes the player to the game and gives instructions on
 * the rules of roulette.
 */
   private void giveInstructions() {
      println("Welcome to the roulette table!");
      println("Roulette is played with a large wheel divided into");
      println("compartments numbered from 0 to 36.  Each player");
      println("places bets on a playing field marked with the");
      println("numbers and various categories.  In this game,");
      println("the only legal bets are the following categories:");
      println("odd, even, low, or high.  Note that 0 is not in any");
      println("category.  After the bet is placed, the wheel is");
      println("spun, and a marble is dropped inside, which bounces");
      println("around until it lands in a compartment.  If the");
      println("compartment matches the betting category you chose,");
      println("you win back your wager plus an equal amount.  If");
      println("not, you lose your wager.");
   }

/* Private instance variables */

   private RandomGenerator rgen = new RandomGenerator();
}
```

Assessing the symptoms

Before you start using the debugger, it is always valuable to run the program to get a sense of what the problems might be. If you run the **Roulette** program, you might see the following sample run:

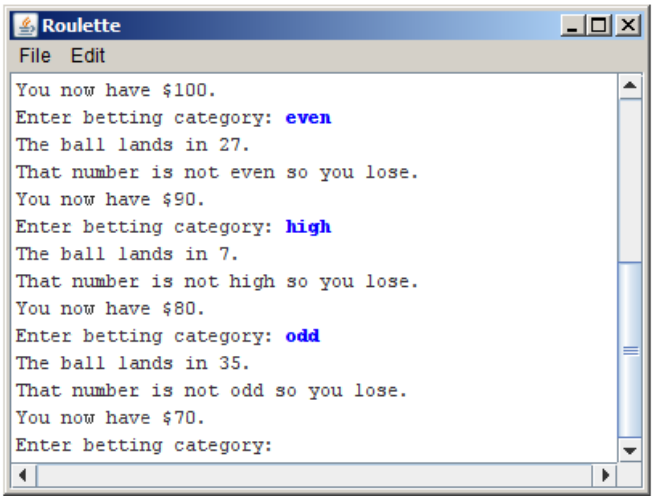


Even though the program is "running," the results don't look so good. The problem is that 20 is even, but the program tells you that it isn't and has happily taken your wager. Something is definitely wrong.

For programs that use random numbers, it is important to debug the program in a deterministic environment. For this reason, it is useful to set the random number generator to a particular seed before proceeding. You can do that by adding the following line to the **run** method:

```
rgen.setSeed(1);
```

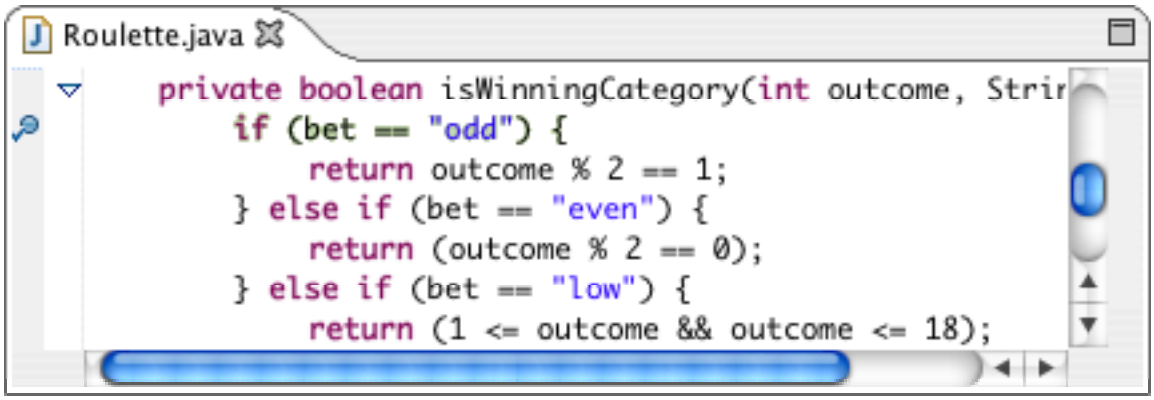
That way your program will work the same way each time, so that you can always get back to the same situation. With this new statement, the output (leaving out the instructions this time) looks like this:



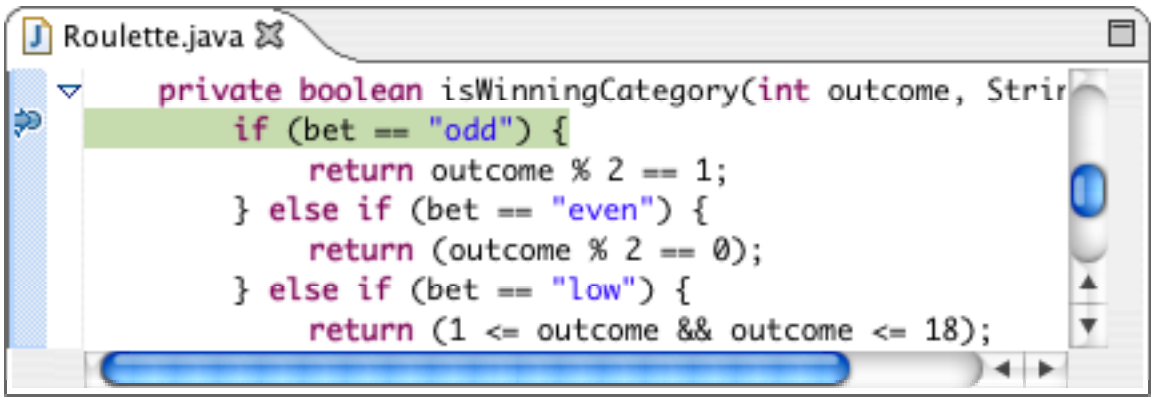
The Eclipse debugger

One of the easiest ways to figure out what the program is doing is to use the Eclipse debugger to help with that process. When you run a project under Eclipse, you can use the debugger to set breakpoints in your code, which will enable you to stop it at interesting places, 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 `Roulette` program, your intuition is likely to suggest that the problem is in the method `isWinningCategory`, given that the sample runs suggest—on the basis of what is still a pretty small sample—that the computer is never allowing the user to win. Thus, you might scroll the editor window down to `isWinningCategory` and set a **breakpoint** on the first line, so that the program will stop there. To do so, double-click in the just barely gray margin (which will surely show up as white on the photocopies), at which point a small circle appears indicating that there is now a breakpoint on that line, as shown:

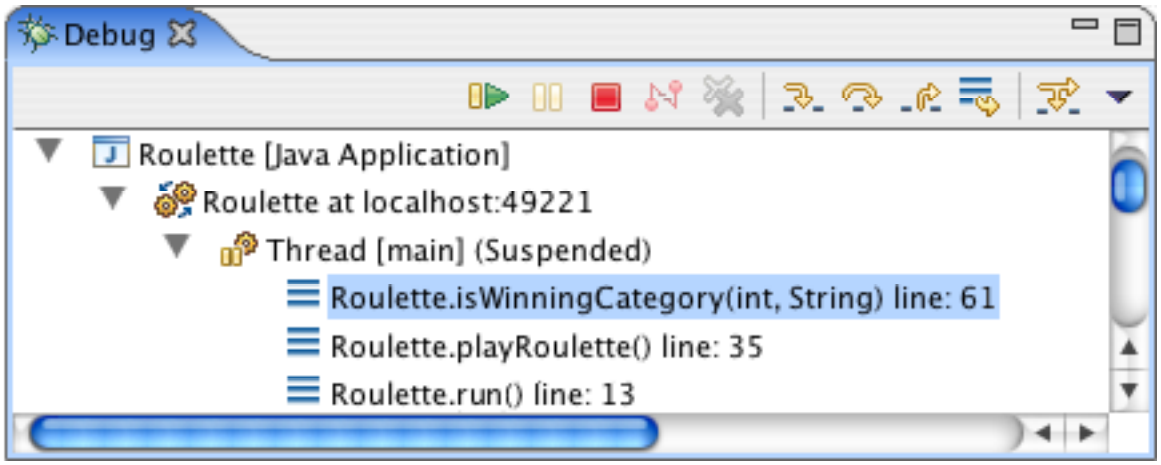


Once you've set the breakpoint, your next step is to run the program with the debugger connected, which you can do by clicking on the bug icon (🐞) that appears to the left of the green triangle on the Eclipse tool bar. Your program will then start running, display all of the instructions, and then again ask you to select a category. Suppose that you again enter `"even"` and watch as the program again tells you that the wheel has landed in number 27 (given that it is now behaving deterministically). At that point, the program calls `isWinningCategory`, 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".)



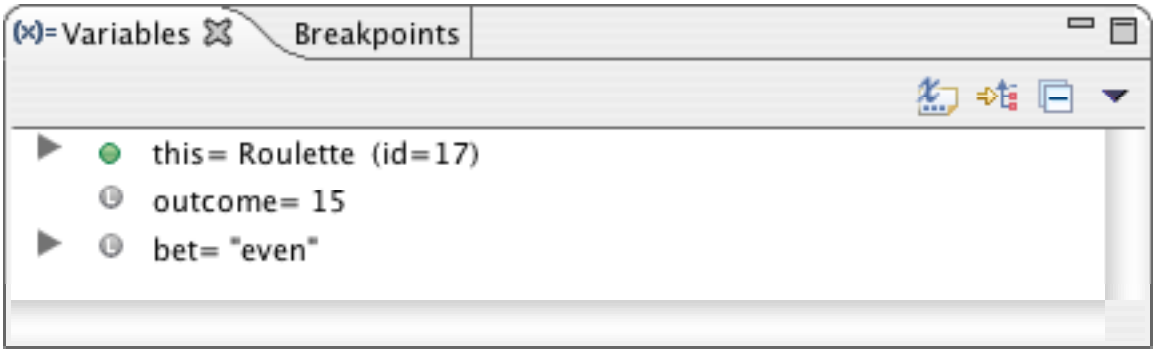
The arrow and the background highlight mark the line of code you are about to execute.

As it happens, however, there is a lot more information displayed in Eclipse's debugging perspective. The "Debug" window at the upper left shows the execution history of the program. If you scroll this window down a couple of lines, you see the following:



The bottom lines tell you where you are in the execution. You are currently at line 61 of **`isWinningCategory`**, which was called from line 35 of **`playRoulette`**, which was in turn called from line 13 of **`run`**. Each of these constitutes a **stack frame,** as described in the text.

The window on the upper right is the "Variables" window, which allows you to see the current values of variables in the current stack frame, which looks like this:

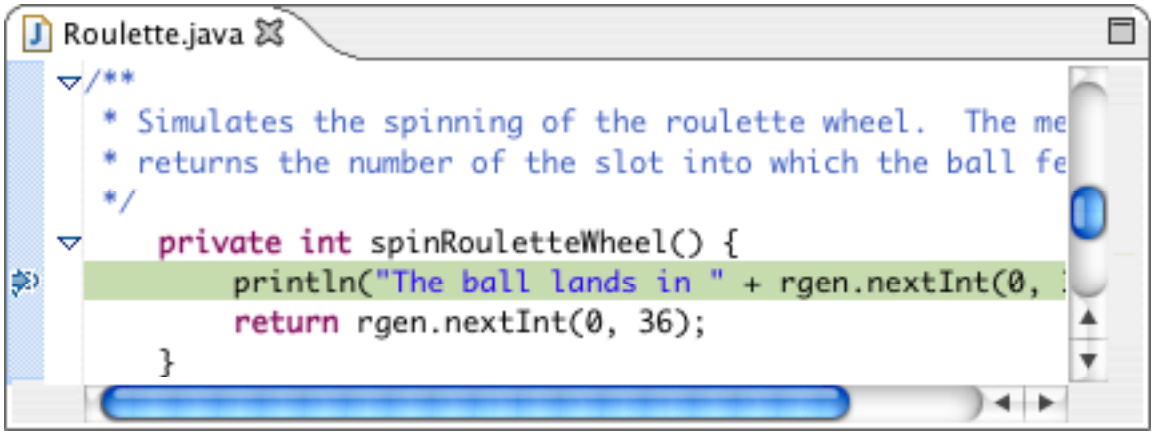


In this frame, the local variables are simply the parameters to **`isWinningCategory`**. We chose the category **`"even"`**, which appears as the value of the variable **`bet`**, and the variable **`outcome`** has the value 15.

But wait a minute, the outcome as shown on the console indicates that the ball landed in 27. Why does this frame show it as 15? 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 you 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, so it is necessary to go back and insert an earlier breakpoint.

You can now go back and stop the **Roulette** program, either by clicking in its close box or by clicking on the red square in the Debug window. In the editor window, you can double-click on the first line of **spinRouletteWheel** to see what's happening there. If you debug the program again, it will soon stop in the following state:



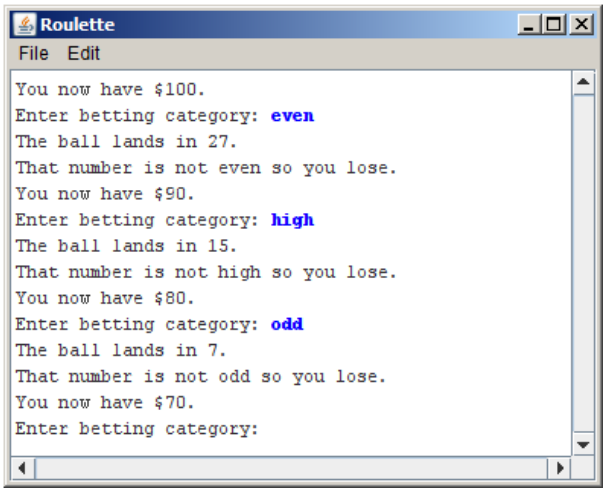
Think about what you know. The **spinRouletteWheel** method prints out one outcome, but somewhere later on that value seems to be different. Armed with that understanding of the problem, look at the code for **spinRouletteWheel**:

```
private int spinRouletteWheel() {
   println("The ball lands in " + rgen.nextInt(0, 36) + ".");
   return rgen.nextInt(0, 36);
}
```

Very soon, the problem will jump out at you. The **println** method generates a random number in the range 0 to 36, but the **return** statement goes on and generates a new one. There is nothing to guarantee that the value **spinRouletteWheel** prints and the value it returns will be the same. You need to rewrite the method as follows:

```
private int spinRouletteWheel() {
   int spin = rgen.nextInt(0, 36);
   println("The ball lands in " + spin + ".");
   return spin;
}
```

Surely, that's the bug. Given your confidence, you try running it without the debugger. Unfortunately, the output looks exactly as it did before. You still lose every time:



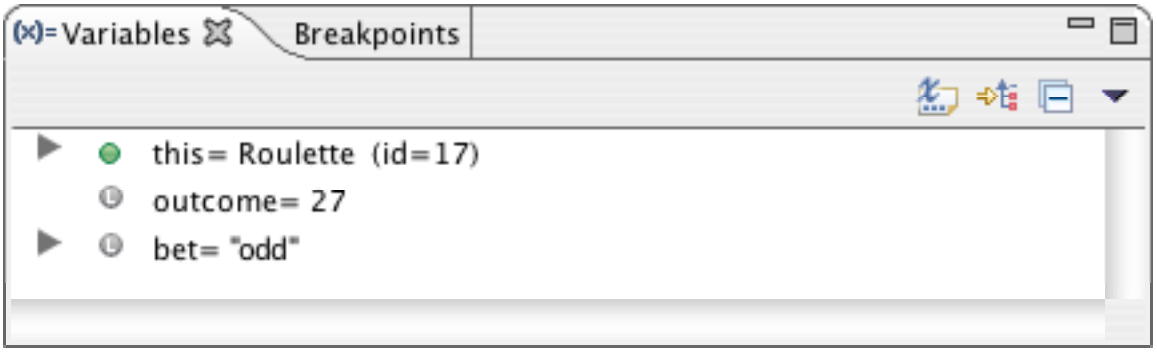
Back to the drawing board.

Finding the critical clues

As always in debugging, your primary goal is to figure out what your program is doing rather than why is it not doing what you wanted. To do so, you need to gather as much information as you can, and Eclipse offers great tools for doing so. It's time to go back and look for more clues.

You still have the breakpoints in place, so try debugging the program and letting it run to the breakpoint in `isWinningCategory`. To save time, however, it is probably worth changing your wager given that you know that the first spin is going to be 27. You're trying to see why the program doesn't pay off, so it might make sense to choose the category `"odd"` rather than `"even"`, since that will allow you to follow through what happens when you have picked the correct category.

When you arrive at the breakpoint, things look a little better. The variables window now looks like this:



The value of `outcome` shows correctly as 27, and we are betting on `"odd"`. Things should therefore be good for the player, but something is still going wrong.

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 as if it had hit a breakpoint.

Terminate. Exits from the program entirely.

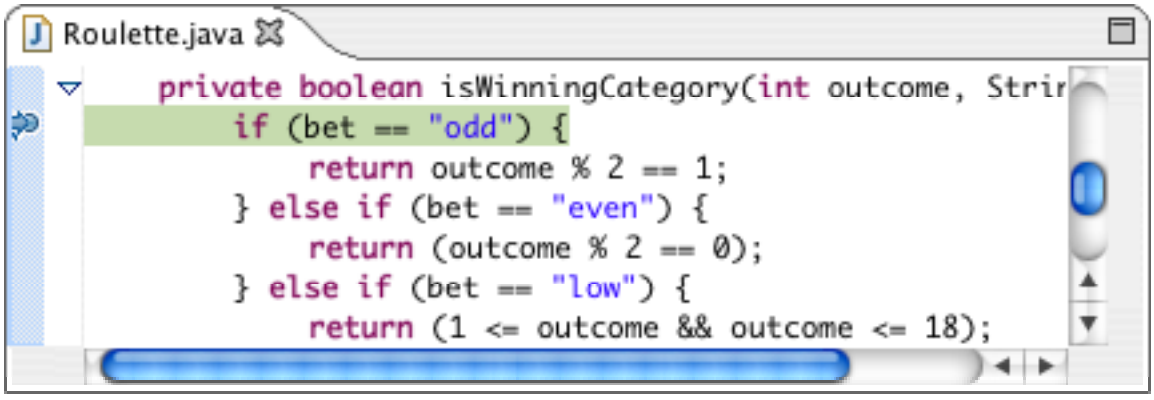
Step Into. Executes one statement of the program and then stops again. If that statement includes a method call, the program will stop at the first line of that method. As noted below, this option is not as useful as **Step Over**.

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.

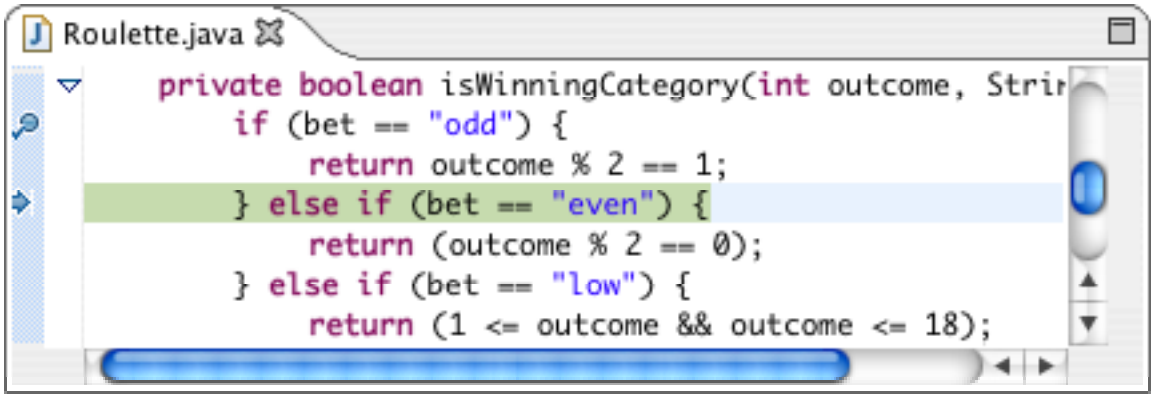
The three stepping options are extremely useful, but you need to take some care in choosing which one to use. In most cases, **Step Over** is the right option to use. Intuitively, it has the effect of continuing to the next step at this level, allowing you to stay at the same conceptual level of abstraction. If the current statements 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.

The source window in the debugger now shows the following situation:



Here, the program is stopped at a line that includes no method calls, so it doesn't matter which of **Step Into** or **Step Over** you choose. To get into the habit, however, it's probably best to go with **Step Over** unless there is a reason not to.

What you would like to have happen at this point is for the program to determine that **`bet`** is in fact equal to **`"odd"`** and continue on to the **`return`** statement on the next line. But it doesn't. When you click on **Step Over**, you end up in the following position:



Avoiding value rigidity

If you go look at the code for **isWinningCategory**, many of you will see the problem immediately. On the other hand, others might stare right at the bug and not see anything. 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 you've written them, seeing them as what you *want* them to mean as opposed to what they in fact *do* mean. Here, the problem lies in the **if** tests, starting with the first:

```
if (bet == "odd")
```

It is so easy to read this conditional test as "if **bet** is equal to the string **"odd"**." What else could it mean? The answer is that Java interprets the **==** operator for objects in a much less intuitive way. The correct English translation of this statement is "if **bet** is the same object as the constant string **"odd"**." Since **bet** was read in from the user, this condition will never be true.

The fix in this case is to replace the instances of the **==** operator with calls to **equals** or, better yet, **equalsIgnoreCase**. Now the **isWinningCategory** method looks like this:

```
private boolean isWinningCategory(int outcome, String bet) {
   if (bet.equalsIgnoreCase("odd")) {
      return outcome % 2 == 1;
   } else if (bet.equalsIgnoreCase("even")) {
      return (outcome % 2 == 0);
   } else if (bet.equalsIgnoreCase("low")) {
      return (1 <= outcome && outcome <= 18);
   } else if (bet.equalsIgnoreCase("high")) {
      return (19 <= outcome && outcome <= 36);
   } else {
      return (false);
   }
}
```

Seeing the process through

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 before you discovered anything amiss. There is, however, one additional problem. The instructions generated by the program go out of their way to remind you that the number 0 is not considered part of any category. The current implementation of **isWinningCategory** correctly determines that 0 is neither high or low, but does allow it to count as even. This error—if it were allowed to stand—would give the player a better-than-50/50 chance of winning by betting on **"even"**, which would undoubtedly be of some concern to the owners of the casino. To avoid the problem, the implementation of **isWinningCategory** must specifically disallow 0 when it tests for even numbers. Thus, the code should look as shown on the next page.

```
private boolean isWinningCategory(int outcome, String bet) {
   if (bet.equalsIgnoreCase("odd")) {
      return outcome % 2 == 1;
   } else if (bet.equalsIgnoreCase("even")) {
      return (outcome % 2 == 0 && outcome != 0);
   } else if (bet.equalsIgnoreCase("low")) {
      return (1 <= outcome && outcome <= 18);
   } else if (bet.equalsIgnoreCase("high")) {
      return (19 <= outcome && outcome <= 36);
   } else {
      return (false);
   }
}
```

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.

Getting back to the regular Editor perspective

Once you have completed using the debugger and want to return to using Eclipse in the regular "Editor" perspective (the one that you've been using most of the time), you can simply go to the "**Stanford**" menu and pick the "**Switch to Editor**" selection.