

## Debugging

Finding and correcting problems, or "bugs", in computer programs can be time consuming and frustrating. It would be great if in this short section a step-by-step process could be articulated to direct you in debugging Java programs. Unfortunately, this is not possible. Debugging is tricky business, and it is extremely difficult to teach. Although there are custom debugging tools available for languages like Java, they are complex and have a steep learning curve. If you are a professional programmer, then the time invested is worth it. But if you program less than, say, two hours a day, you probably lack the time or expertise to operate a debugger.

In this section, we present some tips on debugging. But be forewarned. You will have — and must have — your own style to problem solving. When crunch time arrives, it is *your* approach you'll adopt, regardless of what you read here.

### ***Learn by Experience***

If there is one point worth emphasizing, it is this: You cannot learn computer programming by reading a book. You've got to roll-up your sleeves and dig-in. Similarly, you cannot learn debugging by reading about it. You must write programs to learn programming, and you must make and correct mistakes along the way to hone your debugging skills. Every time you fix a compile error or a logic error, you get a little smarter. Take the initiative to write programs to exercise new operators, classes, or methods. Learn by experience. Write programs, and write lots of them.

### ***Stepwise Refinement***

Write little programs to do small things to confirm your understanding. If they work, you got it right. If they don't, you missed something. Obviously, the less code introduced at once, the better. The worst possible scenario in debugging is confronting too many problems at once. So, write programs incrementally — one step at a time. This is called *stepwise refinement*. Obtain closure on small sections of code, and test them before proceeding. For example, if you need a custom method, start with the shell and fill in the details later. Let's say you need a method called `complexJob()`. You might organize your program initially as follows:

```
public static void main(String[] args)
{
    ...
    double x = complexJob(a, b);
    System.out.println(x);
    ...
}

public static double complexJob(double arg1, double arg2)
{
    return 0;
}
```

The idea is to get the method's definition and call sequence up and running "structurally", before adding the details. If you have problems in the structure *and* in the details, debugging is harder because too many problems are confronted at once. When you add the details, the same rule applies: Add a few lines of code, get closure on them, and compile and test them before proceeding. Take it one step at a time.

## **Compile Errors**

A compile error is a *syntax* error. You have broken a rule in the Java language, and it must be fixed before proceeding. It is common to get a lot of compile errors from a small typing mistake, like misplacing a semicolon. The most important compile error is the first one generated, so focus your attention on it first. You may be surprised to discover that, having eliminated the first compile error, others vanish as well.

The flip side of the coin is this: Even though you have only one compile error, don't be fooled into thinking your program is almost working. Some errors are so drastic that the compiler "gives up" early on and outputs just one error message. Having fixed this, you may discover dozens of new compile errors. Welcome to debugging!

## **Run-time Errors**

A run-time error is a *semantic* error. The code checked-out fine by the compiler, but when the program executed something ran amuck. There is an error in the meaning, or semantics, of the program's behaviour or in the logical flow of the program. Such errors (e.g., accessing a non-existing array element) are not caught by the compiler because it cannot anticipate changes in variables, and the effect of such, as a program executes.

The symptoms for run-time errors are often strange and seemingly unrelated to the underlying problem in the code. Because of this, run-time errors are generally more difficult to correct than compile errors. This hits home at a point made earlier: Take it one step at a time. If you write a lot of code before testing, you are asking for trouble.

There are two broad categories of run-time errors: those that cause the program to crash, and those that cause the program to generate incorrect results or to behave improperly. Both are semantic errors: the former are errors in how you used Java, the latter are errors in how you approached the problem.

## **Program Traces**

One of the simplest debugging techniques is to add a *program trace* to a program. A program trace is simply a print statement that generates intermediate results. By outputting intermediate results, you can observe a program's behaviour *as it executes*. Let's revisit two programs seen earlier and retrofit them with program traces. The program `Palindrome3` is the same as `Palindrome2`, except program traces are added to output intermediate results (see Figure 1).

```

1  import java.io.*;
2  import java.util.*;
3
4  public class Palindrome3
5  {
6      public static void main(String[] args) throws IOException
7      {
8          // open keyboard for input (call it 'stdin')
9          BufferedReader stdin =
10             new BufferedReader(new InputStreamReader(System.in), 1);
11
12             // prepare to extract words from lines
13             String line;
14             StringTokenizer words;
15             String word;
16
17             // main loop, repeat until no more input
18             while ((line = stdin.readLine()) != null)
19             {
20                 // tokenize the line
21                 words = new StringTokenizer(line);
22
23                 // process each word in the line
24                 while (words.hasMoreTokens())
25                 {
26                     word = words.nextToken();
27                     if (word.length() > 2 && isPalindrome(word))
28                         System.out.println(word);
29                 }
30             }
31     }
32
33     public static boolean isPalindrome(String w)
34     {
35         int i = 0;
36         int j = w.length();
37         while (i < j)
38         {
39             System.out.println(w.substring(i, i + 1)
40                 + " <---> " + w.substring(j - 1, j));
41             if (!w.substring(i, i + 1).equals(w.substring(j - 1, j)))
42             {
43                 System.out.println("NOT A PALINDROME");
44                 return false;
45             }
46             i++;
47             j--;
48         }
49         System.out.println("Yes, it's a palindrome");
50         return true;
51     }
52 }

```

Figure 1. Palindrome3.java

An example dialogue with this program follows:

```
PROMPT>java Palindrome3
kayak
k <----> k
a <----> a
y <----> y
Yes, it's a palindrome
kayak
momentum
m <----> m
o <----> u
NOT A PALINDROME
^z
```

Two words were inputted: "kayak", which is a palindrome, and "momentum", which is not. Program traces were added in lines 39-40, 43, and 49. The output above is interesting but it is not particularly relevant to the present discussion — because the program works. However, if the program had a run-time error, or generated wrong answers, the output above would shed some light on the problem. For example, the character comparisons in line 41 are shown explicitly in the output, for example "k <----> k". If the comparison used the wrong indices, this would surface in the trace. Or, if the program crashed, the absence of a trace suggests approximately where the crash occurred.

The `StringBackwards3` program shown earlier used a recursive algorithm to reverse the characters in a string. `StringBackwards4` is the same program, with program traces added (see Figure 2).

```

1  import java.io.*;
2
3  public class StringBackwards4
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter a message string: ");
11         String s = stdin.readLine();
12         s = backwards(s);
13         System.out.print(s);
14     }
15
16     public static String backwards(String s)
17     {
18         if (s.length() == 1)
19         {
20             System.out.println("Length = 1, Return: " + s);
21             return s;
22         }
23         else
24         {
25             System.out.println("Length = " + s.length()
26                 + ", Call argument: "
27                 + s.substring(1, s.length()));
28             s = backwards(s.substring(1, s.length())) + s.substring(0, 1);
29             System.out.println("s = " + s);
30             return s;
31         }
32     }
33 }

```

Figure 2. StringBackwards4.java

A sample dialogue with this program follows:

```

PROMPT>java StringBackwards4
Enter a message string: hyppopotamus
Length = 12, Call argument: yppopotamus
Length = 11, Call argument: ppopotamus
Length = 10, Call argument: popotamus
Length = 9, Call argument: opotamus
Length = 8, Call argument: potamus
Length = 7, Call argument: otamus
Length = 6, Call argument: tamus
Length = 5, Call argument: amus
Length = 4, Call argument: mus
Length = 3, Call argument: us
Length = 2, Call argument: s
Length = 1, Return: s
s = su
s = sum
s = suma
s = sumat
s = sumato
s = sumatop
s = sumatopo
s = sumatopop
s = sumatopopp
s = sumatopoppy
s = sumatopoppyh
sumatopoppyh

```

Program traces are added in lines 20, 25-27, and 29. Not only do program traces help in debugging, they can reveal the inner behaviour of complex algorithms, in this case a recursive algorithm. The output above clearly shows the deepening recursion as method calls are nested inside previous calls, as well as the unwinding of the recursion.

### **Test Cases**

It is a pleasant relief to write a program and get it working properly. You have invested many hours crafting the code — fixing bugs along the way — and when done, it's a great feeling. But, how do you know your code works properly? That's easy, you tested the program and it performed the intended task and generated the correct results. End of story. Hopefully it is, but sometimes an odd thing happens. Many days or weeks later you are working on a new program that uses methods and classes from your earlier work. All of a sudden you notice that results aren't quite as expected. Upon further inspection, you discover that the problem lies in a method or class from the earlier program.

What went wrong in the preceding scenario? It all lies in the testing. We are in the habit of testing our programs with "nominal" data or input conditions, and we are sometimes too quick to pat ourselves on the back when a program works. As it turns out, finding a comprehensive set of test cases for computer programs is quite a challenge. In fact, for very large software projects combining numerous modules written by teams of programmers, it is an absolutely horrendous task. If the software is destined for safety-critical applications (e.g., nuclear power plants or medical equipment), the stakes are high and validation is a massive and complex undertaking.

We needn't worry here, because we just want to write and debug some simple Java programs. However, a little extra effort can go along way in creating robust code. When testing portions of code (i.e., methods), devise test cases that cover the range of possible conditions that may arise. Nominal cases are fine, but test for the extremes as well. For any method that receives an

argument, consider, and test for, extreme cases. If a string argument is passed, make sure the method smoothly handles an empty string. If an integer or double argument is passed, make sure the method can cope with negative values, or special values like zero. These are known as *preconditions*, and they should be tested for at the beginning of a method.

If a method performs numeric calculations, try to anticipate the possibility of results that are out of range. If an integer result is greater than  $2^{31}$ , it cannot be represented by an `int`. A `long` variable should be used. If the result is greater than  $2^{63}$ , it cannot be represented by a `long`. At this point you may have a task for which Java is ill-suited, but, at the very least, attempt to anticipate and accommodate these possibilities.<sup>1</sup>

Let's take this discussion a little further by re-examining a method presented earlier in two programs. The program `Factorial` included a non-recursive method named `factorial()`. The program `Factorial2` included a recursive version of the same method. We noted earlier that the two programs "behaved" the same, and this is true. The following is a sample dialogue for either program, showing an attempt to compute the factorial of a negative number:

```
PROMPT>java Factorial
Enter a non-negative integer: -3
Oops!
```

Both programs reasonably anticipate the possibility of the user inputting a negative number. But, there is a problem lurking if one attempts to paste the code for either `factorial()` method into another program. The check for a negative value occurred "outside" the method. That is, if the value is negative the method is not called in the first place. So, we know how the *program* responds to a negative input, but we don't know how the *method* responds. And, as it turns out, the two methods are quite different in the way they respond to a negative argument. The `factorial()` method in `Factorial` returns the original argument if it is negative (see lines 25-28 in `Factorial`). The `factorial()` method in `Factorial2` embarks on an arduous and incorrect set of recursions if it receives an negative argument (see lines 23-26 in `Factorial2`).<sup>2</sup> This fault was not caught during testing, because the method was not called if the inputted value was negative. A modified version of the recursive `factorial()` method appears below:

```
public static int factorial(int n)
{
    if (n < 0)
        return n;
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Now, the method returns immediately with the original argument if it is negative, as with the non-recursive version. This is a precondition and it is checked for at the beginning of the method. The behaviour is consistent and predictable. Yes, but it is still wrong. We cannot calculate the

---

<sup>1</sup> All is not lost, however, as there is a class called `BigInteger` in the `java.math` package that supports integer arithmetic with arbitrary precision.

<sup>2</sup> If the check for negative input is removed in `Factorial2`, and a negative value is entered, the program will crash with a `StackOverflowError`.

factorial of a negative number, just as we cannot calculate the square root of a negative number. By returning the original (negative!) argument we are opening the door to future problems, since we do not know, and cannot anticipate, how a calling program might use the value returned by the `factorial()` method. A more appropriate way to cope with a negative argument is to throw an exception and thereby shift the responsibility (of passing an the invalid argument) back to the calling program. We will learn how to do this later.

While we are discussing the factorial programs, here's something else to consider. What is the factorial of 15? Compare the results from the Factorial program with that obtained using a pocket calculator. We'll leave it to you to think about the different results and to explore these through the debugging techniques just discussed.