Appendix B

Debugging[†]

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

- Compile-time errors are produced by the compiler and usually indicate that there is something wrong with the syntax of the program. Example: omitting the semi-colon at the end of a statement.
- Run-time errors are produced by the run-time system if something goes wrong while the program is running. Most run-time errors are Exceptions. Example: an infinite recursion eventually causes a StackOverflowException.
- Semantic errors are problems with a program that compiles and runs, but doesn't do the right thing. Example: an expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, there are some techniques that are applicable in more than one situation.

B.1 Compile-time errors

The compiler is spewing error messages.

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it fails, and it reports spurious errors.

In general, only the first error message is reliable. I suggest that you only fix one error at a time, and then recompile the program. You may find that

[†]This appendix appears in Allen B. Downey, *How to Think Like a Computer Scientist: Java Version*, 2002.

one semi-colon "fixes" 100 errors. Of course, if you see several legitimate error messages, you might as well fix more than one bug per compilation attempt.

I'm getting a weird compiler message and it won't go away.

First of all, read the error message carefully. It is written in terse jargon, but often there is a kernel of information there that is carefully hidden.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally the error will be prior to the location of the error message, but there are cases where it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Now is a good time to go through the whole program and make sure it is indented properly. I won't say that good indentation makes it easy to find syntax errors, but bad indentation sure makes it harder.

Now, start examining the code for the common syntax errors.

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.

2. Remember that upper case letters are not the same as lower case letters.

3. Check for semi-colons at the end of statements (and no semi-colons after squiggly-braces).

4. Make sure that any strings in the code have matching quotation marks (and that you use double-quotes, not single).

5. For each assignment statement, make sure that the type on the left is the same as the type on the right.

6. For each method invocation, make sure that the arguments you provide are in the right order, and have right type, and that the object you are invoking the method on is the right type.

7. If you are invoking a fruitful method, make sure you are doing something with the result. If you are invoking a void method, make sure you are not *trying* to do something with the result.

8. If you are invoking an object method, make sure you are invoking it on an object with the right type. If you are invoking a class method from outside the class where it is defined, make sure you specify the class name.

9. Inside an object method you can refer to the instance variables without specifying an object. If you try that in a class method, you will get a confusing message like, "Static reference to non-static variable."

If nothing works, move on to the next section...

I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your development environment to make sure the program you are editing is the program the compiler is compiling. If you are not sure, try putting an obvious and deliberate syntax error right at the beginning of the program. Now compile again. If the compiler doesn't find the new error, there is probably something wrong with the way you set up the project.

Otherwise, if you have examined the code thoroughly, it is time for desperate measures. You should start over with a program that you can compile and then gradually add your code back.

- Make a copy of the file you are working on. If you are working on `Fred.java`, make a copy called `Fred.java.old`.

- Delete about half the code from `Fred.java`. Try compiling again.

  - If the program compiles now, then you know the error is in the other half. Bring back about half of the code you deleted and repeat.
  - If the program still doesn't compile, the error must be in this half. Delete about half of the code and repeat.

- Once you have found and fixed the error, start bringing back the code you deleted, a little bit at a time.

This process is called "debugging by bisection." As an alternative, you can comment out chunks of code instead of deleting them. For really sticky syntax problems, though, I think deleting is more reliable—you don't have to worry about the syntax of the comments, and by making the program smaller you make it more readable.

B.2 Run-time errors

My program hangs.

If a program stops and seems to be doing nothing, we say it is "hanging." Often that means that it is caught in an infinite loop or an infinite recursion.

- If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the section titled "Infinite loop."

- Most of the time an infinite recursion will cause the program to run for a while and then produce a StackOverflowException. If that happens, go to the section titled "Infinite recursion."

  If you are not getting a StackOverflowException, but you suspect there is a problem with a recursive method, you can still use the techniques in the infinite recursion section.

- If neither of those things works, start testing other loops and other recursive methods.

- If none of those things works, then it is possible that you don't understand the flow of execution in your program. Go to the section titled "Flow of execution."

Infinite loop

If you think you have an infinite loop and think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition, and the value of the condition.

For example,

```
while (x > 0 && y < 0) {
    // do something to x
    // do something to y

    System.out.println ("x: " + x);
    System.out.println ("y: " + y);
    System.out.println ("condition: " + (x > 0 && y < 0));
}
```

Now when you run the program you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loops keeps going, you will be able to see the values of `x` and `y` and you might figure out why they are not being updated correctly.

Infinite recursion

Most of the time an infinite recursion will cause the program to run for a while and then produce a StackOverflowException.

If you suspect that method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case, but the program doesn't seem to be reaching it, add a print statement at the beginning of the method that prints the parameters. Now when you run the program you will see a few lines of output every time the method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each method with a message like "entering method foo," where `foo` is the name of the method.

Now when you run the program it will print a trace of each method as it is invoked.

It is often useful to print the parameters each method receives when it is invoked. When you run the program, check whether the parameters are reasonable, and check for one of the classic errors—providing parameters in the wrong order.

When I run the program I get an Exception.

If something goes wrong during run time, the Java run-time system prints a message that includes the name of the exception, the line of the program where the problem occurred, and a stack trace.

The stack trace includes the method that is currently running, and then the method that invoked it, and then the method that invoked *that*, and so on. In other words, it traces the path of method invocations that got you to where you are.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened.

NullPointerException: You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out what variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an object type, it is initially `null`, until you assign a value to it. For example, this code causes a NullPointerException:

```
Point blank;
System.out.println (blank.x);
```

ArrayIndexOutOfBoundsException: The index you are using to access an array is either negative or greater than `array.length-1`. If you can find the site where the problem is, add a print statement immediately before it to print the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backwards through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing.

If either one is a parameter, go to the place where the method is invoked and see where the values are coming from.

StackOverFlowException: See "Infinite recursion."

I added so many print statements I get inundated with output.

One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: either simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the error.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the error is in a deeply-nested part of the program, try rewriting that part with simpler structure. If you suspect a large method, try splitting it into smaller methods and test them separately.

Often the process of finding the minimal test case leads you to the bug. For example, if you find that a program works when the array has an even number of elements, but not when it has an odd number, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

B.3 Semantic errors

My program doesn't work.

In some ways semantic errors are the hardest, because the compiler and the run-time system provide no information about what is wrong. Only you know what the program was supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes this hard is that computers run so fast. You will often wish that you could slow the program down to human speed, but there is no straightforward way to do that, and even if there were, it is not really a good way to debug.

Here are some questions to ask yourself:

- Is there something the program was supposed to do, but doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should. Add a print statement to the beginning of the suspect methods.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to built-in Java methods. Read the documentation for the methods you invoke. Try out the methods by invoking the methods directly with simple test cases, and check the results.

In order to program, you need to have a mental model of how programs work. If your program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the classes and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

Here are some common semantic errors that you might want to check for:

- If you use the assignment operator, `=`, instead of the equality operator, `==`, in the condition of an `if`, `while` or `for` statement, you might get an expression that is sytactically legal, but it doesn't do what you expect.
- When you apply the equality operator, `==`, to an object, it checks shallow equality. If you meant to check deep equality, you should use the `equals` method (or define one, for user-defined objects).

- Some Java libraries expect user-defined objects to define methods like `equals`. If you don't define them yourself, you will inherit the default behavior from the parent class, which may not be what you want.

- In general, inheritance can cause subtle semantic errors, because you may be executing inherited code without realizing it. Again, make sure you understand the flow of execution in your program.

I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
bobby.setColor (bobby.getColor().darker());
```

Can be rewritten as

```
Color bobbyColor = bobby.getColor();
Color darker = bobbyColor.darker ();
bobby.setColor (darker);
```

The explicit version is easier to read, because the variable names provide additional documentation, and easier to debug, because we can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Java, you might write

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit.

```
double y = x / (2 * Math.PI);
```

Any time you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intend); it will also be more readable for other people who haven't memorized the rules of precedence.

I've got a method that doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of

```
public Picture fourPics (Picture p1, Picture p2, Picture p3, Picture p4){
    return above (beside (p1, p2), beside (p3, p4));
}
```

You could write

```
public Picture fourPics (Picture p1, Picture p2, Picture p3, Picture p4){
    Picture row1 = beside (p1, p2);
    Picture row2 = beside (p3, p4);
    Picture grid = above (row1, row2);
    return grid;
}
```

Now you have the opportunity to print (or display) any of `row1`, `row2` or `grid` before returning.

My print statement isn't doing anything

If your use the `println` method, the output gets displayed immediately, but if you use `print` (at least in some environments) the output gets stored without being displayed until the next newline character gets output. If the program terminates without producing a newline, you may never see the stored output.

If you suspect that this is happening to you, try changing all the `print` statements to `println`.

I'm really, really stuck and I need help

First of all, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and/or rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backwards").
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and I let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. You program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need.

- What kind of bug is it? Compile-time, run-time, or semantic?
- If the bug occurs at compile-time or run-time, what is the error message, and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, in this class the goal is not to make the program work. The goal is to learn how to make the program work.