Lab 4: Debugging
12:00 PM, Feb 12, 2020

Contents

1 The Mystery Continues... 1
2 Debugging 2
   2.1 Debuggers ................................................................. 2
   2.2 Getting Started in the IntelliJ Debugger .......................... 3
   2.3 IntelliJ Time! ............................................................... 4
   2.4 Helpful Alternative Debugging Strategies ........................ 6

Objectives

By the end of this lecture, you will be able to:

   • utilize state to achieve cool things
   • utilize the IntelliJ debugger

Setup

Before beginning today’s lab, copy the support code to your lab04/src directory inside your javaproject directory:

cp /course/cs0180/src/lab04/src/*
~/course/cs0180/workspace/javaproject/src/lab04/src

1 The Mystery Continues...

After returning from getting a sandwich at the Blue Room, you find a huge mob of angry CS 18 students at your door, demanding to see you, the resident CIT detective. They’re all shouting, but you manage to glean that over the past couple of hours, the SignMeUp queue keeps getting shuffled and the TAs don’t know what’s going on. No even seems to notice that you’ve arrived, so you run to the door, stand on a chair, and calm the crowd down. You listen to each complaint and jot down the times that the queue was shuffled. After you send the students away, you sit at your computer to think. While absentmindedly scrolling through CampusWire to check if your question has been answered, you come across a strange pattern: all the times Isabel answered a post match exactly to the times that the hours queue was shuffled!
You find Isabel in the third floor atrium to ask her some questions. She must be behind the queue shuffling and maybe even the other crimes — why else would someone answer so many CampusWire questions? After an intense interrogation, however, it’s clear that Isabel just has an addiction to CampusWire and doesn’t know anything about the SignMeUp shuffling. Baffled, you turn to the source; that is, the source code for SignMeUp. Looking at the huge amounts of code is daunting though, and Isabel can see you’re intimidated by the sheer volume. She suggests it might be useful to learn some tools to step through and debug code to make your job easier.

2 Debugging

As you’ve learned by now, debugging is an immensely important part of programming. Even when coding up an algorithm seems easy, fixing its bugs can be extremely difficult. This section of today’s lab is meant to teach you debugging strategies that should come in handy in your future CS endeavors, whether it be in CS 18 or otherwise.

Bugs are, put simply, when something that you don’t think will happen does in fact happen, or when something you think will happen does not happen.

As your programs grow in size, you’ll often find that an error in one part of your program is due to something in a completely different part of your program. This means you can’t just inspect your program at the place where the error manifests itself. Good debugging strategies provide you with context information that helps locate the origin of a bug.

First, we’ll show you a really cool tool common to many languages and IDEs, that IntelliJ provides!

2.1 Debuggers

A **debugger** is a program you can use to help you search for bugs. Critically, a debugger allows you to set **breakpoints**. If you set a breakpoint on a line of your code, then when you run your program, your program will pause on that line before executing it. This allows you to inspect the state of your program before resuming execution.

Luckily, IntelliJ comes with a built in debugger! Let’s go deeper into what debuggers allow you to do. Following are the most helpful tools you can use:

**Breakpoints** - setting a **breakpoint** on a line of code, then running your program, will pause your program each time it reaches that line. It will pause before executing the line. At that time, you can investigate the state of your program by looking at the menu of variables in the Variables pane. Then, you can continue your program in one of several ways.

**Step Into** - Once you’re stopped at a breakpoint, there are a few options on how you want to continue. Suppose you have a breakpoint on a line that has a call to a **method**. For example, suppose you are stopped on the line:

```java
getCatAge(myAge);
```

If you **step into** this method, then you will enter the `getCatAge` method you wrote, and pause execution on the **first line inside that method**, before that line is executed.

If you are stopped at a breakpoint, but there is no method call in the line you’re stopped at, you should avoid using step into.
Note: Since you are debugging the code you wrote, you should never step into a method that you did not write. You’ll reach all kinds of weird places. For example, if you step into this line:

```java
System.out.println("Hello World!");
```

You’ll find yourself in the source code for the `println` method! There’s no way that could be the cause of a bug in your program, so you should not step into system calls.

**Step Over** - Similar to step into, step over is a way to proceed after being paused at a breakpoint. However, whereas step into would pause on the first line inside the method call, step over will execute the method, then break on the following line of code. For example, suppose we again have a breakpoint on the call to `getCatAge`:

```java
getcAtAge(myAge);
getcHumanAge(myCatsAge);
```

Now, let’s say you *step over*. Then, `getCatAge` will execute, and you will pause on the `getHumanAge` line. In this sense, you do not *enter* a method call, but rather, execute it and go to the next line.

In effect, step over will *execute the current line, and pause on the next line* of code, regardless of whether the current line was a method or not.

**Step Return** - What if you pressed step into, but you meant to use step over instead, and now you’re stuck who-knows-where in some source code from a system call? Not to fear! You can use step return to finish executing the function you’re inside, then pause. For example, suppose in the following code, you have a breakpoint on the first line, and you accidentally *step into* it:

```java
System.out.println(myAge);
getcAtAge(myAge);
```

Then, if you press step return, you’ll finish executing the `println` and pause on the `getCatAge` line.

**Resume** - this is yet another way to continue from a breakpoint. This will, in effect, resume execution normally until the next breakpoint or until the program terminates.

Note: What would happen if you *step into* a line of code containing *multiple* function calls, like this one:

```java
System.out.println(getcAtAge(10));
```

When you step into this, you’ll be in the `getCatAge` code. Once `getCatAge` returns, you’ll again be on the above line. If you step into this again, you’ll now be in the `println` source code.

This might sound a bit confusing - how will you know which function you’re stepping into? The answer is to go by order of evaluation. In the above example, `getCatAge` must be evaluated before it can be printed, so when you step into the line, you step into `getCatAge` first. However, don’t stress too much - if you step into the wrong function, you can always use `resume` to continue executing until the next breakpoint or `step return` to finish executing the function you’re in and then pause again.

### 2.2 Getting Started in the IntelliJ Debugger

Phew! That was a lot of new information. We’re going to use IntelliJ’s debugger in this lab, but first, we want to make sure that all made sense. In the following task, we’ll list some examples of
what you might want to do using the debugger, and ask you how to accomplish them using the
commands we’ve just described.

**Task:** For each of the following, write down (on paper or in a text editor) the sequence of steps
you would need to accomplish the goal. This sequence should consist of commands from our list
above. For example, if one of the tasks asked how to pause on the main method, you would write
“set breakpoint on main method.”

Open up the provided files, namely Restaurant, Manager, and Party. There’s no need to
understand the details of what they’re each doing - yet!

1. I want to pause in the `arrive` method in Restaurant.

2. I want to pause at the beginning of the Restaurant constructor, and pause again once it’s
done, to ensure all of the Restaurant fields are initialized properly. Then, once I’m sure of
that, I want to execute the program with no further interruptions. I only want to use a single
breakpoint!

3. I have a breakpoint on the openRestaurant method, and I’ve just reached it. I want to look
at what’s happening in the startManaging method call in openRestaurant without adding
more breakpoints.

4. I had a breakpoint on the main method (not a good idea), but accidentally stepped into the
call to the LinkedList constructor. Help! How do I return to the Restaurant program I
wrote?

5. I again had a breakpoint on the main method, but I want to skip past all the reservations,
add calls, and go into the Restaurant constructor method, without adding more breakpoints.

**Note:** Your answers shouldn’t contain “restart the program”! Only use the tools we’ve mentioned
above.

You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 2.3 IntelliJ Time!

Now that you understand conceptually how to use the debugger, it’s time to put your new knowledge
to use.

- To run the debugger, use the little bug icon next to the green run button in the top right
corner. As with the run button, clicking it will run the last program ran. Use the dropdown if
you want to run something new!

- To insert a **breakpoint**, move your mouse to the left of the line you want to break on. If you
have line numbers enabled, it should be to the right of the line numbers. Then, single-click,
and a red dot should appear. To remove it single-click on the breakpoint.

- To **step over**, use the leftmost blue arrow button.

- To **step into**, use the second to the left blue arrow that points downward at a horizontal line.
- To **force step into**, use the third to the left blue arrow (you shouldn’t need to use this in this course).

- To **step return**, use the fourth the left blue arrow that points upward.

- To **run to cursor**, use the rightmost button among the arrow buttons that points at a cursor (this will run your program until the line where your cursor is currently located).

- To **resume**, use the green, play button on the left side of your window.

**Task:** Test out some of your answers from the last exercise to get comfortable with the debug window setup. Feel free to close the debugger tab by clicking ”X”.

Now, it’s time for you to debug our project! First, we’ll give you an outline of how it works.

At a high level, the project simulates a restaurant with a fixed number of tables, where parties of diners arrive and depart. We assume any party can eat at any table, and the maximum number of parties that can simultaneously be seated is, therefore, equal to the number of tables in the restaurant. Further, only parties on the list of upcoming reservations can be seated.

When a party (who is on the list of reservations) **arrives**, they are either seated (if there are open tables) or put on a queue of parties who are waiting to be seated. When a party **departs**, the next party waiting in the queue is seated immediately.

The following classes and descriptions are the parts of our project you should know:

**Party** - a class representing a group of people that can be seated at a restaurant. A party has only a name and no other characteristics.

**Restaurant** - a class representing the state of the restaurant. It has:

- a name, a Manager, list of upcoming reservations, list of currently seated parties, queue of parties waiting to be seated, and number of tables.

- a constructor which takes the name of the restaurant, number of tables in the restaurant, and list of upcoming reservations for that night.

- **void** `arrive` - a method that takes a party which has just arrived. The method checks the reservation list (and returns if the party isn’t on it), then seats the party if there is available room; otherwise, it adds the party to the queue of parties waiting to be seated.

- **void** `depart` - a method that takes the party which is departing. It removes the party from the list of seated parties, then checks the queue of waiting parties. If there’s a waiting party, it seats that party.

- a **main** method that creates several parties and starts up a `Restaurant`.

**Manager** - a class that randomly decides when parties arrive at or depart from the restaurant. It has:

- **int** `makeDecision` - a method that chooses at random whether to let a new party arrive or make a seated party depart.

- **void** `partyArrives` - a method that tells the restaurant to let a new party arrive.
• **void** partyDeparts - a method that tells the restaurant to let a seated party depart.

• **Party** chooseParty(Collection<Party> parties) - a method that chooses a party at random from a collection of parties.

There are a few new things included in this restaurant simulation. You’ll notice two new Java interfaces: Collection and Queue. The Collection interface represents any sort of collection, whereas the Queue interface is used for any data structure that can be used as a queue, such as a LinkedList. Feel free to check out the Java documentation for Collection and Queue for more information.

Further, you’ll notice that Restaurant has an instance of a Manager as a field, and Manager also has an instance of a Restaurant as a field. Circular references like this are common in OOP, and allow the two objects to interact by calling methods on each other.

If the restaurant simulation were working properly, you would see interleaved arrival and departure messages, sometimes multiple arrivals/departures in a row, etc. Critically, you should:

• Never see print statements indicating there are more seated parties than there are tables at any point in the output.

• Never see one particular party arrive or depart more than once.

• See that once a party departs, the party waiting the longest should be seated immediately.

• See that all parties eventually leave, and that the restaurant closes for the night once they do.

**Task:** Use the IntelliJ debugger to find the two bugs, and fix them! Be sure to be able to explain to a TA what you did and why the code wasn’t working before.

**Hint:** The bugs are both found in methods we’ve outlined above, not in getters/setters or any other unmentioned methods. Further, the bugs are not found in either makeDecision or chooseParty inside the Manager class.

**Hint:** Run the project before trying to debug it, to see what’s wrong with it and form an educated guess about where the bugs are originating.

**Hint:** We know you may not have seen all of these types before, and might be unfamiliar with their methods. Hover your mouse above any method name or any type declaration to get more information!

You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 2.4 Helpful Alternative Debugging Strategies

This section discusses other good ways to debug your programs, which you’ll hopefully find useful both in CS 18 and your future programming endeavors! It is not necessary to complete this section in order to receive full credit on the lab. However, we strongly recommend you read through this information regardless.

**Code Reviews:** After you write code, it’s always good to double-check your logic. To review, or walk through, your code, simply read it over line by line, piecing together the logic at each step. Be sure to keep an eye out for common errors like null pointer or array out-of-bounds exceptions.
Code reviews are generally more effective when done in pairs or groups. Under the CS 18 collaboration policy, you are not permitted to perform collaborative code reviews on homeworks or exams. However, we strongly encourage you to utilize this tool in labs and projects!

**Assertions:** Generally, when you write code, you make assumptions like the following: “I know that condition X cannot be true here because of how I’ve written my code; therefore, I will not handle condition X.” However, it is always possible that your assumptions are not actually true. A good debugging strategy, then, would be to write an explicit “test” in your code that checks whether condition X holds, and if it does, to throw an unchecked exception reporting what went wrong. Such an exception is not meant to be handled, but rather is meant to alert you about a mistake in your assumptions.

This type of double-checking by verifying your assumed invariants and failing if the invariant does not hold is called an **assertion**.

Suppose we are writing a method that takes an age in human years and converts it to cat years. Let’s say we’ve assumed that no one would input an age less than 0, but we want to double-check that this is actually the case, using asserts:

```java
public double getCatAge(double humanAge) {
    assert(humanAge >= 0);
    return (humanAge * 7 - 1);
}
```

**Note:** This is not the actual formula to convert human years to cat years.

An assert takes as input a statement that should evaluate to a boolean. If the boolean evaluates to true, the assert will pass and your program will run as if the assert were not there. However, if the boolean evaluates to false, Java will throw an AssertionError and terminate your program immediately.

Checking assertions takes time, so by default, Java ignores assertions. To enable them from the command line, run Java with the `-ea` flag. To enable them in IntelliJ, go to Run → Edit Configurations. Then, in the “VM options” box, type `-ea`. Then apply your settings. Once all your assertions pass, if your program is running slowly, feel free to turn off assertions again.

**Printing:** Adding print statements to your code is a very useful debugging technique. It’s useful for two key reasons:

- The order in which your print statements appear reveals information about the flow of control of your program. For example, print statements could alert you if a while loop is being executed too many times.
- You can print the values of variables, thus enabling you to inspect your program’s state at particular points in its execution.

**Warning:** though print statements can be useful, there are good ways and bad ways to use them. The bad way is to flood your program with them. Too much information can actually cause you to spend more time sifting through output than seeing the big picture. The good way is to come up with educated guesses of what might be going wrong, then using print statements to confirm or deny those hypotheses. For example, you might say “If I’m getting output X, this could be caused by variable Y being in state Z,” then use a print statement to verify if variable Y really is in state Z.
Importantly, be sure to delete print statements once you’ve finished using them. You shouldn’t hand in code containing debugging print statements in CS 18!

**Just for Fun:** Now that you’ve read all about our top debugging strategies, brainstorm some pros and cons to each strategy with your partner.

*The Mystery Continues to Continue...* After learning all these debugging techniques, you go back to the SignMeUp source code with renewed confidence. Using a copious amount of breakpoints, you manage to come across a mysterious piece of code. After further inspection, you’re sure that this is what caused the shuffling problem. Scrolling up a little, you see a commented out block of text that sends chills down your spine:

```c
/*
 * TO THE RESIDENT CIT DETECTIVE
 * THIS TWISTY CANDY WAS USED TO MAKE THE SHOE CHARLIE CHAPLIN ATE IN
 * "THE GOLD RUSH"
 * LGESJCULTJQGKWP
 */
```

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)