CSC 300 / CSC 402
Supplemental Material
Professor Eric Fredericks
DePaul University

Tracing Recursion and Call Stacks

In Java (and indeed, most languages), **recursion** is implemented by using a **stack** data structure, called the **call stack**. The stack is used to keep track of where the program is as it executes. If you think about a program in Java, the entry point is declared with the signature:

```
public static void main(String[] args)
```

Without this static function, you cannot run your code. This entry point is tracked as the first element of the call stack.

Elements (or "entries") in the call stack are called **stack frames**. A stack frame contains the memory allocated by the Java Virtual Machine (JVM) that is scoped to a particular invocation of a function. The line number at which a function is executing can also be thought of as being part of an invocation of a function, and so we will model line numbers as being stored in a stack frame.

When we have a **recursive implementation of the fibonacci function** (which, as we see in our programming homework on recursion can be pretty **terrible**!), we can visualze an invocation of the function using the following drawings.

For the following drawings, refer to the code and line numbers I provided to you in `MyRecursionTrace.java`. **We are calling `fib(4)`**. These diagrams illustrate how the call stack changes as the code executes. Note that each invocation of a function (recursive or not) adds a stack frame to the call stack, which is removed from the stack as the function returns.

In these drawings, we only draw the state of execution at the point where we are invoking a function (**pushing** onto the call stack) or returning from a function call to the caller (**popping** from the call stack). Intermediate lines of code that execute (and that would show up in the diagrams if you used the `Trace` utility) are not shown. Note how we can trace the memory for each base type variable in each stack frame. *Remember that in a recursive call, each invocation has its own copy of memory allocated (declared) within the function, including memory for variables that are declared as parameters to the function.*

Diagrams begin on the next page, and there is some further information at the end, after the diagrams, including a couple of exercises for you to try on your own.

```
==========================================
1.
      +------------+
main: | line: 37   | calling fib(4) PUSHING
      +------------+
==========================================
2.
      +------------+
      | line: 9    | calling fib(3) PUSHING
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
3.
      +------------+
      | line: 9    | calling fib(2) PUSHING
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
4.
      +------------+
      | line: 9    | calling fib(1) PUSHING
fib:  | n:       2 |
      +------------+
      | line: 9    | calling fib(2)
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
5.
      +------------+
      | line: 8    | return 1       POPPING
fib:  | n:       1 |
      +------------+
      | line: 9    | calling fib(1)
fib:  | n:       2 |
      +------------+
      | line: 9    | calling fib(2)
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
```

```
==========================================
6.
      +------------+
      | line: 10   | calling fib(0) PUSHING
      | n:       2 |
fib:  | nMinus1: 1 |
      +------------+
      | line: 9    | calling fib(2)
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+

==========================================
7.
      +------------+
      | line: 8    | return 0       POPPING
fib:  | n:       0 |
      +------------+
      | line: 10   | calling fib(0)
      | n:       2 |
fib:  | nMinus1: 1 |
      +------------+
      | line: 9    | calling fib(2)
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
8.
      +------------+
      | line: 11   | return 1      POPPING
      | n:       2 |
fib:  | nMinus1: 1 |
      | nMinus2: 0 |
      +------------+
      | line: 9    | calling fib(2)
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
```

```
==========================================
9.
      +------------+
      | line: 10   | calling fib(1) PUSHING
      | nMinus1: 1 |
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+

==========================================
10.
      +------------+
      | line: 8    | return 1       POPPING
fib:  | n:       1 |
      +------------+
      | line: 10   | calling fib(1)
      | nMinus1: 1 |
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
11.
      +------------+
      | line: 11   | return 2       POPPING
      | nMinus1: 1 |
      | nMinus2: 1 |
fib:  | n:       3 |
      +------------+
      | line: 9    | calling fib(3)
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
12.
      +------------+
      | line: 10   | calling fib(2) PUSHING
      | nMinus1: 2 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
```

```
==========================================
13.
      +------------+
      | line: 9    | calling fib(1) PUSHING
fib:  | n:       2 |
      +------------+
      | line: 10   | calling fib(2)
      | nMinus1: 2 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
14.
      +------------+
      | line: 8    | return 1       POPPING
fib:  | n:       1 |
      +------------+
      | line: 9    | calling fib(1)
fib:  | n:       2 |
      +------------+
      | line: 10   | calling fib(2)
      | nMinus1: 2 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
15.
      +------------+
      | line: 10   | calling fib(0) PUSHING
      | nMinus1: 1 |
fib:  | n:       2 |
      +------------+
      | line: 10   | calling fib(2)
      | nMinus1: 2 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
16.
      +------------+
      | line: 8    | return 0       POPPING
fib:  | n:       0 |
      +------------+
      | line: 10   | calling fib(0)
      | nMinus1: 1 |
fib:  | n:       2 |
      +------------+
      | line: 10   | calling fib(2)
      | nMinus1: 2 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
```

```
==========================================
17.
      +------------+
      | line: 11   | return 1       POPPING
      | nMinus1: 1 |
      | nMinus2: 0 |
fib:  | n:       2 |
      +------------+
      | line: 10   | calling fib(2)
      | nMinus1: 2 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
18.
      +------------+
      | line: 11   | return 3       POPPING
      | nMinus1: 2 |
      | nMinus2: 1 |
fib:  | n:       4 |
      +------------+
main: | line: 37   | calling fib(4)
      +------------+
==========================================
19.
      +------------+
main: | line: 38   |
      | f: 3       |
      +------------+
==========================================
```

Tracing recursion can be tedious! Aren't you glad we have computers to do these kinds of things? However, since we're studying computer science, it's important to understand how Java is implemented. Tracing with call stacks is an important skill. If you learn it, you will be able to answer a question on an exam like this:

Given the following function, what is printed when `doSomething(7)` **is executed?**

```
public static void doSomething(int n) {
    if(n <= 1) {
        StdOut.print(n);
    } else if(n > 3) {
        StdOut.print(n);
        doSomething(n/2);
        StdOut.print(n);
    } else {
        StdOut.print(n);
        doSomething(n-1);
        doSomething(n-2);
        StdOut.print(n);
    }
}
```

This function is in the companion code to this handout, starting at line 21.

The answer to this question is **732102137**.

You can, of course, run this code and turn in an answer for a quiz or homework question, and never think about it again. **But on an exam, you must be able to look at a problem like this, reason about it, and provide an answer.** *Points are subtracted if you do not communicate to me that you understand how recursion works and you don't include all of the output.*

I will not ask you to draw call stacks on an exam. However, for this homework, please draw the call stacks as shown in the example. This will reinforce the ideas involved with recursive thinking.

Incidentally, if your recursive Java function call never terminates (the recursion equivalent of an infinite loop), Java will continue building up these stack frames in memory until there's no more memory available. That's when you'll see a `StackOverflowError`!

Exercises

For each of the following function calls in `main()` provided in `MyRecursionTrace.java`, draw the call stack as we did in the `fib` example above. Your drawings should show the stack just before each **push** or **pop** operation. Clearly indicate the name of the function associated with the stack frame, the variables in each stack frame, the line number where the function is called in the case of a push, and the line number where the function returns in case of a pop.

1. `MyRecursionTrace.main`, line 40: `gcd(9, 24)`
2. `MyRecursionTrace.main`, line 43: `doSomething(7)`