HOW RECURSION WORKS IN JAVA

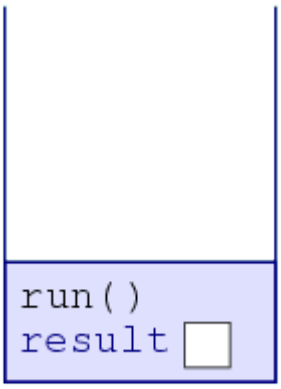
You'll notice that the `compute` method doesn't *always* recur: When its parameter `n` is 1, the method simply returns immediately without any recursive invocations.

With a bit of thought, you'll realize that any functional recursive method must have such a situation, since otherwise, the recursive method will never finish. In fact, these situations are important enough to merit a special term: Any condition where a recursive method does not invoke itself is called a **base case**.

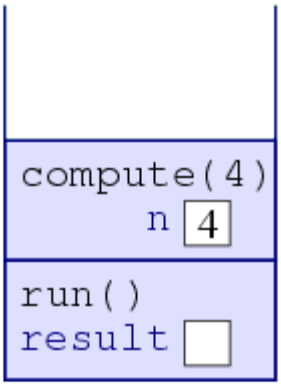
But what exactly happens when a recursive method lacks a base case? To understand this, we need to get some idea about how a computer handles method invocations.

In executing a program, the computer creates what is called the **program stack**. The program stack is a stack of **frames**, each frame corresponding to a method invocation. At all times, the computer works on executing whichever method is at the stack's top; but when there is a method invocation, the computer creates a new frame and places it atop the stack. When the method at the stack's top returns, the computer removes that method's frame from the stack's top, and resumes its work on the method now on the frame's top. (This removal process is sometimes called **popping the stack**; the addition process (when a method invocation takes place) is sometimes called **pushing**.)

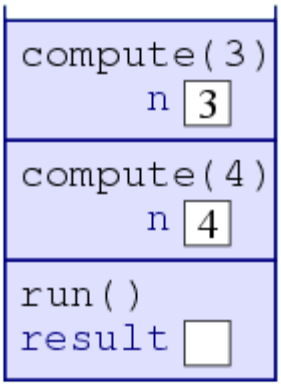
To see how this works, let's diagram how the `Mystery` program operates.

1. 

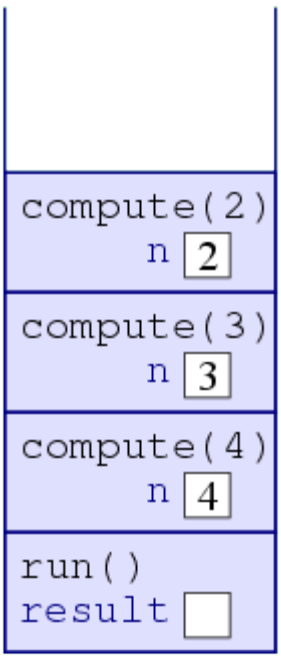
   To start off the program, the program pushes a frame corresponding to an invocation to `run()`. Notice how this frame includes room for `run`'s variable `result`.

2. 

When the computer sees that `run()` invokes `compute(4)`, the computer places a new frame atop the stack corresponding to `compute`; this frame will include the variable `n`, whose value is initially 4.

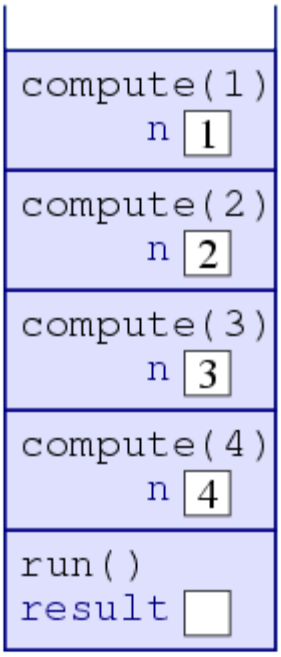
3. 

When the computer sees that `compute(4)` invokes `compute(3)`, the computer places a new frame atop the stack, containing the variable `n`, whose value is initially 3.

4. 

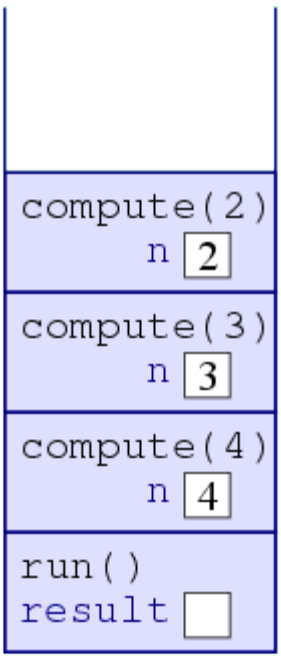
When the computer sees that `compute(3)` invokes `compute(2)`, the computer places a new frame atop the stack, containing the variable `n`, whose value is initially 2.

5.

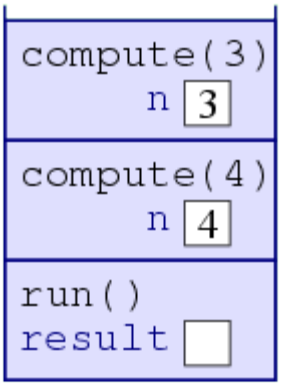


When the computer sees that `compute(2)` invokes `compute(1)`, the computer places a new frame atop the stack, containing the variable `n`, whose value is initially 1.

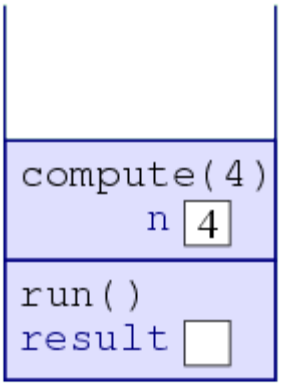
6.



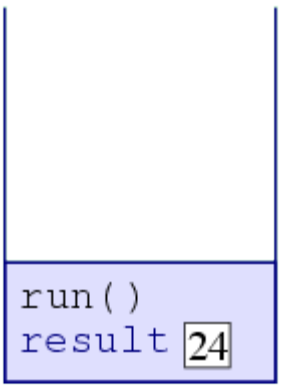
When the computer sees that `compute(1)` returns, it pops the top frame off the stack and resumes with whatever frame is now at the top — which happens to be the frame for `compute(2)`.

7. 

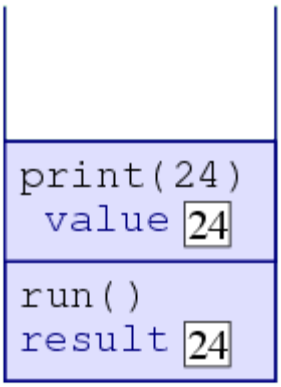
When the computer sees that `compute(2)` returns, it pops the top frame off the stack and resumes with `compute(3)`.

8. 

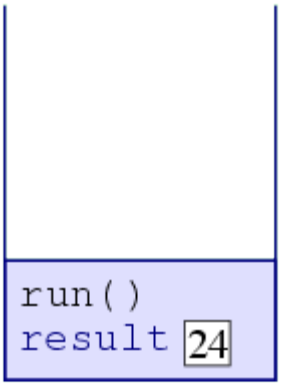
When the computer sees that `compute(3)` returns, it pops the top frame off the stack and resumes with `compute(4)`.

9. 

When the computer sees that `compute(4)` returns, it pops the top frame off the stack and resumes with `run()`. The `run()` invocation assigns the valued returned to its `result` variable, which modifies the variable in its frame.

10. 

As the computer executes the method atop the stack, `run`, it sees that the method invokes `print`. It thus pushes `print(24)` onto the stack. In fact,`print` will push additional methods onto the stack, which are all eventually popped off.

11. 

Once `print` returns, the computer pops its frame off the stack and continues executing `run`. In fact, `run` will return promptly (since there is nothing else to do in that method). Thus, its frame will be popped off, too. Once the stack is empty, the computer halts execution of the program.

So what happens if a recursive method never reaches a base case? The stack will never stop growing. The computer, however, limits the stack to a particular height, so that no program eats up too much memory. If a program's stack exceeds this size, the computer initiates an exception, which typically would crash the program. (From the operating system's point of view, crashing the program is preferable to allowing a program to eat up too much memory and interfere with other better-behaved programs that may be running.) The exception is labeled a `StackOverflowError`.

So any time you see a `StackOverflowError`, the most likely cause is that there is some sort of recursion going on, and that recursion never reaches a base case. In fact, this would occur with the `Mystery` program if we simply the 4 in line 5 to a 0.

Source : http://www.toves.org/books/java/ch17-recur/index.html