RECURSION IN JAVA – AN INTRODUCTION

We saw how to create methods. Inside their bodies, we can include invocations of other methods. It may not have occurred to you, but you might reasonably wonder: Could a method invoke itself?

Self-invocation may at first sound useless or illegal: Isn't this defining something in terms of itself — what is called a circular definition? But self-invocation is legally, and it's actually quite useful. In fact, it's so useful that it gets its own special name: **recursion**. We'll explore recursion in this chapter.

A first example

Let us begin with an example and see how it works.

Figure 17.1: A `Mystery` program.

```
1  import acm.program.*;
2
3  public class Mystery extends Program {
4      public void run() {
5          int result = compute(4);
6          println(result);
7      }
8
9      public int compute(int n) {
10         if(n == 1) {
11             return 1;
12         } else {
13             return n * compute(n - 1); // here's the recursive invocation
14         }
15     }
16 }
```

The program of Figure 17.1 defines `compute`, which is recursive because it invokes itself on line 13. Let's step through the program and see how it will work.

1. **`run()`:** We start, of course, in the `run` method. This program immediately invokes `compute` with a parameter of 4. This will temporarily suspend work on `run` until the invocation `compute(4)` completes.

2. **`compute(4)`:** With the parameter variable `n` having the value 4 as assigned, we run through `compute`. Since 4 isn't 1 (line 10), we go into the **`else`** clause. On line 13, we find we must invoke a method named `compute` with a parameter of 3. Thus, we temporarily suspend our work until this recursive invocation `compute(3)` completes.
3. **`compute(3)`:** We now run through `compute` with the parameter `n` being 3. Since 3 isn't 1, we go into the **`else`**, where we find we must recursively invoke `compute` with a parameter of 2. Thus, we temporarily suspend our work until this recursive invocation `compute(2)` completes.
4. **`compute(2)`:** We now run through `compute` with the parameter `n` being 2. Since 2 isn't 1, we go into the **`else`**, where we find we must recursively invoke `compute` with a parameter of 1. Thus, we temporarily suspend our work until this recursive invocation `compute(1)` completes.
5. **`compute(1)`:** We now run through `compute` with the parameter `n` being 1. Since the **`if`** condition is turns out to be **`true`**, we go to line 11, which say we should return 1. This completes the invocation to `compute(1)`.
6. **`compute(2)`:** We had previously suspended the invocation of `compute(2)` at line 13 until `compute(1)` completed. It has now finished, so we pick up where we left off at line 13. This line says to return `n * compute(n - 1)`. We just finished with determining that `compute(n - 1)` returns a value of 1, so we now want to return `n * 1`.Since `n` has a value of 2 in this current invocation of `compute`, we end up returning the value of `2 * 1`, which is 2.
7. **`compute(3)`:** We had suspended the invocation of `compute(3)` at line 13 until `compute(2)` completed. It has now finished, returning a value of 2. We are to return`n * compute(n - 1)`. Since `n` has a value of 3 in this invocation of `compute`, and we just finished with determining that `compute(n - 1)` has a value of 2, we return 6 (that is, $3 \cdot 2$).
8. **`compute(4)`:** We had suspended the invocation of `compute(4)` at line 13 until `compute(3)` completed. It has now finished, returning a value of 6. We are to return`n * compute(n - 1)`. Since `n` has a value of 4 in this invocation of `compute`, and we just finished with determining that `compute(n - 1)` has a value of 6, we return 24 (that is, $4 \cdot 3$).
9. **`run()`:** We had suspended the invocation of `run()` at line 5 until `compute(4)` completed. It has now finished, returning a value of 24. Thus, we assign the `result` variable to refer to 24 and we continue to the next line, which will display 24 for the user to see.

What this program manages to do is to display the value of $4 \cdot (3 \cdot (2 \cdot 1))$. That is, it displays the product of the numbers from 4 down to 1. More generally, what this `compute`method does is to return the product of all the integers between 1 and its parameter `n`. Mathematicians call this product the **factorial** of `n`, and indeed the program would be better if its `compute` method were given the more descriptive name of `factorial` — but then what it does wouldn't be much of a mystery any more for those who knew the term.

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