Lesson 41: Recursion

What is recursion?

Software **recursion**, very simply, is the **process of a method calling itself**. This at first seems very baffling…somewhat like a snake swallowing its own tail. Would the snake eventually disappear?

The classical factorial problem:

We will begin with the classical problem of finding the factorial of a number. First, let us define what is meant by “factorial”. Three factorial is written as 3!, Four factorial is written as 4!, etc. But what, exactly, do they mean? Actually, the meaning is quite simple as the following demonstrates:

$$3! = 3 * 2 * 1 = 6$$
$$4! = 4 * 3 * 2 * 1 = 24$$

The only weird thing about factorials is that we define 0! = 1. There is nothing to “understand” about 0! = 1. It’s a definition, so just accept it.

Here is an iterative approach to calculating 4!.

```
int answer = 1;
for(int j = 1; j <= 4; j++)
{
	answer = answer * j;
}
System.out.println(answer); //24
```

Before we present the recursive way of calculating a factorial, we need to understand one more thing about factorials. Consider 6!.

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 6 * (5 * 4 * 3 * 2 * 1)$$

We recognize that the parenthesis could be rewritten as 5!, so 6! could be rewritten as

$$6! = 6 * (5!)$$

In general we can write $n! = n(n - 1)!$. It is this formula that we will use in our recursive code as follows:

```
public static int factorial(int n)
{
	if(n = = 1)
	{ return 1; }
	else
```

```
        { return n * factorial(n – 1); //notice we call factorial here }
}
```

Call this code with System.out.println(factorial(4)); //**24**

What really happens when the method calls itself? To understand this, we should pretend there are several copies of the *factorial* method. If it helps, you can think of the next one that is called as being named *factorial1*, and the next *factorial2*, etc. Actually, we need not pretend. This is very close to what really takes place. Analyzing the problem in this way, the last *factorial* method in this "chain" returns 1. The next to the last one returns 2, the next 3, and finally 4. These are all multiplied so the answer is 1 * 2 * 3 * 4 = 24.

Short cuts:

Let's look at some recursion examples using short cuts. For each problem, see if you can understand the pattern of how the answer (in bold print) was obtained.

1. System.out.println(adder(7)); // **46**

```
public static int adder(int n)
{
        if (n<=0)
                return 30;
        else
                return n + adder(n-2);
}
```

On the first call to *adder*, *n* is 7, and on the second call it's 5 (7 - 2), etc. Notice that in the *return* portion of the code that each *n* is **added** to the next one in the sequence of calls to *adder*. Finally, when the *n* parameter coming into *adder* gets to 0 or less, the returned value is 30. Thus, we have:
7 + 5 + 3 + 1 + 30 = 46

2. System.out.println(nertz(5)); // **120**

```
public static int nertz(int n)
{
        if (n = = 1)
                return 1;
        else
                return n * nertz(n-1);
}
```

On the first call to *nertz*, *n* is 5, and on the second call it's 4 (obtained with 5 - 1), etc. Notice that in the *return* portion of the code that each *n* is **multiplied** times the next one in the sequence of calls to *nertz*. Finally, when the *n* parameter coming into *nertz* gets to 1, the returned value is 1. Thus, we have:

5 * 4 * 3 * 2 * 1 = 120

3. System.out.println(nrd(0)); // **25**

```
public static int nrd(int n)
{
        if (n > 6)
                return n - 3;
        else
                return n + nrd(n +1);
}
```

On the first call to *nrd*, *n* is 0, and on the second call it’s 1 (obtained with 0 + 1), etc. Notice that in the *return* portion of the code that each *n* is **added** to the next one in the sequence of calls to *nrd*. Finally, when the *n* parameter coming into *adder* gets above 6, the returned value is n – 3 (obtained with 7 – 3 = 4). Thus, we have:

0 + 1 + 2 + 3 + 4 + 5 + 6 + 4 = 25

4. System.out.println(festus(0)); // **12**

```
public static int festus(int n)
{
        if (n > 6)
                return n - 3;
        else
        {
                n = n * 2;
                return n + festus(n + 1);
        }
}
```

On the first call to *festus*, *n* is 0 (and is modified to 0*2 = 0), and on the second call it’s 1 (0 + 1 = 1, but quickly modified to 1 * 2 = 2), etc. Notice that in the *return* portion of the code that each **modified** *n* is **added** to the next one in the sequence of calls to *festus*. Finally, when the *n* parameter coming into *festus* gets above 6, the returned value is n – 3 (7 – 3 = 4). Thus, we have:

0 + 2 + 6 + 4 = 12

5. What is displayed by *homer(9);* ? **1,2,4,9**

```
public static void homer(int n)
{
        if (n <= 1)
                System.out.print(n);
```

```
        else
        {
                homer(n / 2);
                System.out.print(“,” + n);
        }
}
```

Notice on this method that we successively pass in these values of *n*.

    9 4 2 1

Nothing is printed until the last time when we are down to a 1. Then we start coming back up the calling chain and printing.

6. What is displayed by *method1(7);* ? **1,3,5,7**

```
public static void method1(int n)
{
        if (n <= 1)
                System.out.print(n);
        else
        {
                method1(n-2);
                System.out.print(“,” + n);
        }
}
```

7. In this problem we will generate the Fibonacci sequence. This important sequence is found in nature and begins as follows:

    0, 1, 1, 2, 3, 5, 8, 13, 21, …

We notice that beginning with the third term, each term is the sum of the preceding two. Recursively, we can define the sequence as follows:

    fibonacci(0) = 0
    fibonacci(1) = 1
    fibonacci(n) = fibonacci(n - 1) + fibonacci(n -2)

Using these three lines, we can write a recursive method to find the kth term of this sequence with the call, *System.out.println(fib(k));* :

```
public static int fib(int n)
{
        if (n = = 0)
        {
```

```
            return 0;
        }
        else if(n = = 1)
        {
            return 1;
        }
        else
        {
            return fib(n – 1) + fib(n – 2);
        }
    }
```

8. Let’s try one similar to #7. What is returned by *pls(4);* ? **85**

```
    public static int pls(int n)
    {
        if (n = = 0)
        {
            return 5;
        }
        else if (n = = 1)
        {
            return 11;
        }
        else
        {
            return pls(n - 1) + 2 * pls(n - 2);
        }
    }
```

The way we approach this is to just build the sequence from the rules we see expressed in the code. Term 0 has a value of 5 and term 1 has a value of 11.

Term number	→	0	1	2	3	4
Value	→	5	11			

How will we get term 2? Well, the rule in the code says it’s the previous term plus twice the term before that. That gives us 11 + 2*5 = 21. Continue this to obtain the other terms.

Term number	→	0	1	2	3	4
Value	→	5	11	21	43	**85**

9. We are going to use these same ideas to easily work the next problem that in the beginning just looks hopelessly complicated.

```
public void f(int z)
{
        if(z = = 0)
        {
                System.out.print(“x”);
        }
        else
        {
                System.out.print(“{”);
                f(z-1);
                System.out.print(“}”);
        }
}
```

Let’s begin analyzing this by observing the output of *f(0)*. It simply prints an “x”.

Term number	→	0	1	2	3
Value	→	x			

Now, what about *f(1)*? It first prints a “{” followed by *f(z-1)*. But *f(z-1)* is simply the previous term, and we already know that it’s an “x”. A “}” follows. So our 2nd term is “{x}”.

Term number	→	0	1	2	3
Value	→	x	{x}		

Similarly, each subsequent term is the previous term sandwiched in between “{” and “}” and so we have:

Term number	→	0	1	2	3
Value	→	x	{x}	{{x}}	{{{x}}}

So, if we are asked for *f(3)* the answer is **{{{x}}}.**

10. What is returned by *g(6, 2)*?

```
public static void g(int x, int y)
{
        if (x/y != 0)
        {
                g(x/y, y);
        }
        System.out.print(x / y + 1);
}
```

To analyze this problem the following pairs will represent the parameters on subsequent recursive calls to *g*. Under each pair is what's printed.

6, 2	3, 2	1, 2
4	2	1

Realizing that we don't print until we reach the end of the calling chain, we see that **124** is printed as we "back-out" of the chain.