Intro to Recursion

Recursion is a technique that leads to elegant solutions to problems that are difficult to solve using simple iteration(loops). In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem.

> Recursion in Computer Science is a method where the solution to a problem depends on the solution to smaller instances of the same problem.

This lesson introduces the concepts and techniques of recursive programming and illustrates with examples of how to **"think recursively"**. Let's understand this by a simple example:

```
Calculate factorial of a number by using recursion
```

Now, How do you find `n!` of a given number?

- To find `1!` is easy, because you know that `0!` is 1, and 1! is 1 × 0!.
- Assuming that you know `(n - 1)!`, you can obtain n! immediately by using `n × (n - 1)!`. Thus, the problem of computing n! is reduced to computing (n - 1)!. When computing `(n - 1)!`, you can apply the same idea recursively until n is reduced to 0.
- Let factorial(n) be the method for computing n!. If you call the method with n = 0, it immediately returns the result. The method knows how to solve the simplest case, which is referred to as the **base case** or the stopping condition.
- If you call the method with n > 0, it reduces the problem into a subproblem for computing the factorial of n - 1.
- The subproblem is essentially the same as the original problem, but it is simpler or smaller. Because the sub-problem has the same property as the original problem, you can call the method with a different argument, which is referred to as a recursive call.
- The recursive algorithm for computing factorial(n) can be simply described as follows:

```
if(n == 0){
    return 1;
}else{
```

```
        return n * factorial(n - 1);
    }
```

A recursive call can result in many more recursive calls, because the method keeps on dividing a subproblem into new subproblems. For a recursive method to terminate, the problem must eventually be reduced to a stopping case, at which point the method returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller. The original problem can now be solved by multiplying `n` by the result of **factorial(n - 1)**.

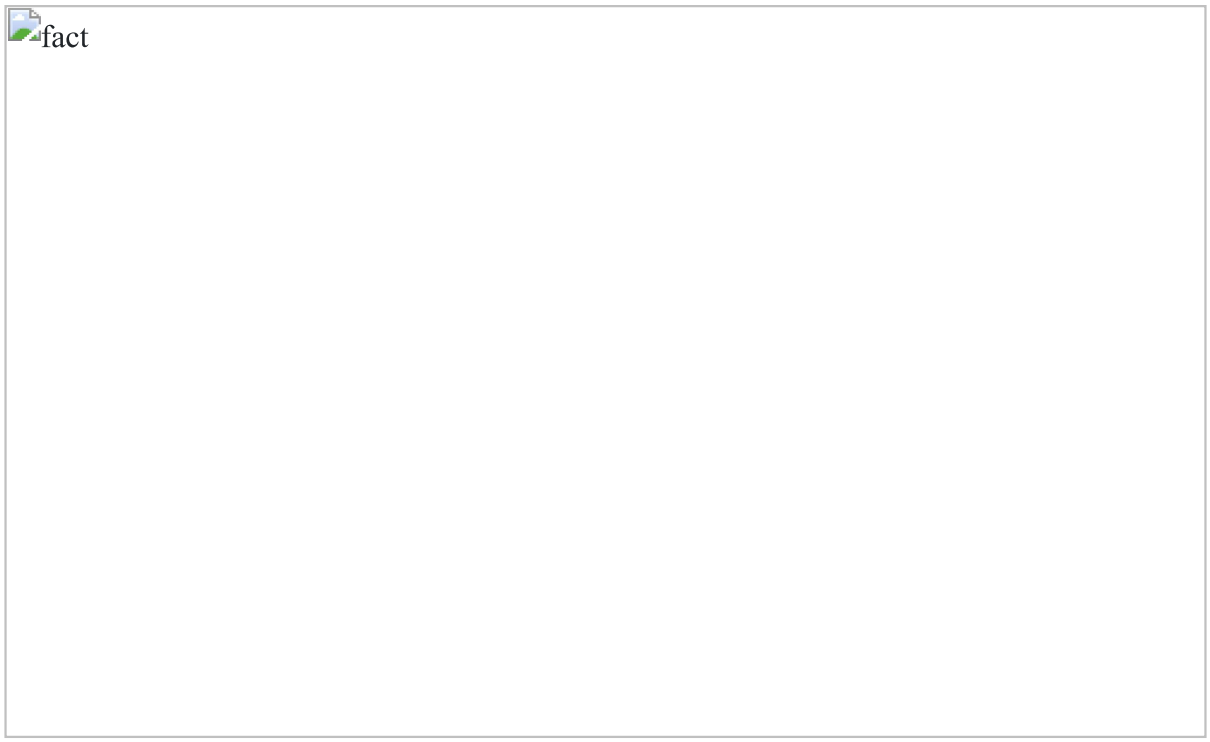
```
import java.util.Scanner;

public class CalculateFactorial {
/** Main method */
    public static void main(String[] args) {
    // Create a Scanner
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a nonnegative integer: ");
        int n = input.nextInt();

        // Display factorial
        System.out.println("Factorial of " + n + " is " + factorial(n));
    }

    /** Return the factorial for the specified number */
    public static long factorial(int n) {
        if (n == 0) // Base case
            return 1;
        else
            return n * factorial(n - 1); // Recursive call
    }
}
```

› Output:

```
eg-1:
Enter a nonnegative integer: 4
Factorial of 4 is 24
```

```
eg-2:
Enter a nonnegative integer: 10
Factorial of 10 is 3628800
```

The `factorial` method is essentially a direct translation of the recursive mathematical definition for the factorial into Java code. The call to factorial is recursive because it calls itself. The parameter passed to factorial is decremented until it reaches the base case of 0. You see how to write a recursive method. How does recursion work behind the scenes? Below diagram illustrates the execution of the recursive calls, starting with n = 4.




```
public static long factorial(int n) {
    return n * factorial(n - 1);
 }
```

The example discussed in this section shows a recursive method that invokes itself. This is known as direct recursion. It is also possible to create indirect recursion. This occurs when method A invokes method B, which in turn invokes method A. There can even be seeral more methods involved in the recursion. For example, method A invokes method B, which invoke method C, which invokes method A. Let'seee one more another example to grasp it better.

Computing Fibonacci Numbers

As I mentioned that Recursion enables we to create an intuitative straightforward, simple solution to a problem. In this section, we'll be looking an example for creating an intuitive solution to a problem using recursion. Consider the well-known Fibonacci-series problem:

```
The series: 0 1 1 2 3 5 8 13 21 34 55 89 …
 (indexes): 0 1 2 3 4 5 6 7 8 9 10 11
```

The Fibonacci series begins with 0 and 1, and each subsequent number is the sum of the preceding two. The series can be recursively defined as:

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1);
for index >= 2
```

So now the problem is: How do we find `fib(index)` for a given index?

It is easy to find `fib(2)`, because we know `fib(0)` and `fib(1)`. Assuming that we know `fib(index - 2)` and `fib(index - 1)`, we can obtain `fib(index)` immediately. Thus, the problem of computing `fib(index)` is reduced to computing `fib(index - 2)` and `fib(index - 1)`. When doing so, we apply the idea recursively until index is reduced to `0` or `1`.

The base case is index = `0` or index = `1`. One thing here to keep in mind that we can have more than one **Base case** also. If we call the method with index = `0` or index = `1`, it immediately returns the result. If we call the method with index >= `2`, it divides the problem into two subproblems for computing `fib(index - 1)` and `fib(index - 2)` using recursive calls. The recursive algorithm for computing `fib(index)` can be simply described as follows:

```
if (index == 0) {
    return 0;
}
else if (index == 1) {
    return 1;
}
else {
```

```
        return fib(index - 1) + fib(index - 2);
    }
```

let's see how we can implement this recursive technique into the code:

```
import java.util.Scanner;

public class ComputeFibonacci {
    // Main method
    public static void main(String[] args) {
        // Create a Scanner
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an index for a Fibonacci number: ");
        int index = input.nextInt();

        // Find and display the Fibonacci number
        System.out.println("The Fibonacci number at index "
        + index + " is " + fib(index));
    }

    // The method for finding the Fibonacci number
    public static long fib(long index) {
        if (index == 0) {// Base case
            return 0;
        }
        else if (index == 1) {// Base case
            return 1;
        }
        else {// Reduction and recursive calls
            return fib(index - 1) + fib(index - 2);
        }
    }
 }
```

Output:

```
Enter an index for a Fibonacci number: 1 (entered by user)
The Fibonacci number at index 1 is 1
```

```
Enter an index for a Fibonacci number: 6 (entered by user)
The Fibonacci number at index 6 is 8

Enter an index for a Fibonacci number: 7 (entered by user)
The Fibonacci number at index 7 is 13
```

A little mind-melting, right? Let's break it down really quick.

 fibonacci

The successive recursive calls for evaluating `fib(4)`. The original method, `fib(4)`, makes two recursive calls, `fib(3)` and `fib(2)`, and then returns `fib(3) + fib(2)`. But in what order are these methods called? In Java, operands are evaluated from left to right, so `fib(2)` is called after `fib(3)` is completely evaluated. There are many duplicated recursive calls. For instance, `fib(2)` is called twice, `fib(1)` three times, and `fib(0)` twice. In general, computing `fib(index)` requires roughly twice as many recursive calls as does computing `fib(index - 1)`. As you try larger index values, the number of calls substantially increases, see below:

```
Table: Number of Recursive Calls in fib(index)
index: 2 3 4 10 20 30 40 50
number of calls: 3 5 9 177 21891 2,692,537 331,160,281 2,075,316,483
```

Okay so I hope you will able to understand the idea of Recursion and why it is used, so let's move forward to the next lesson to know more about the Recursion.