# Recursion Methods

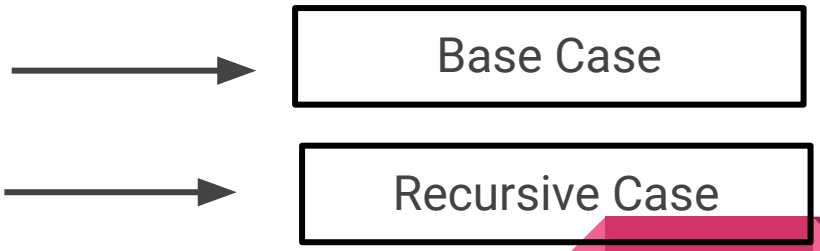
# Agenda

- Recall recursion
- Define tail recursion
- Define non-tail recursion
- Process a String recursively
- Examples
- Classwork: CodingBat

# Recall

**Recursion**: It is the process where a function calls itself to solve problems by breaking them into smaller subproblems.

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



# Tail Recursion

In tail recursion, **no other operation is needed** after the successful execution of a recursive function call.

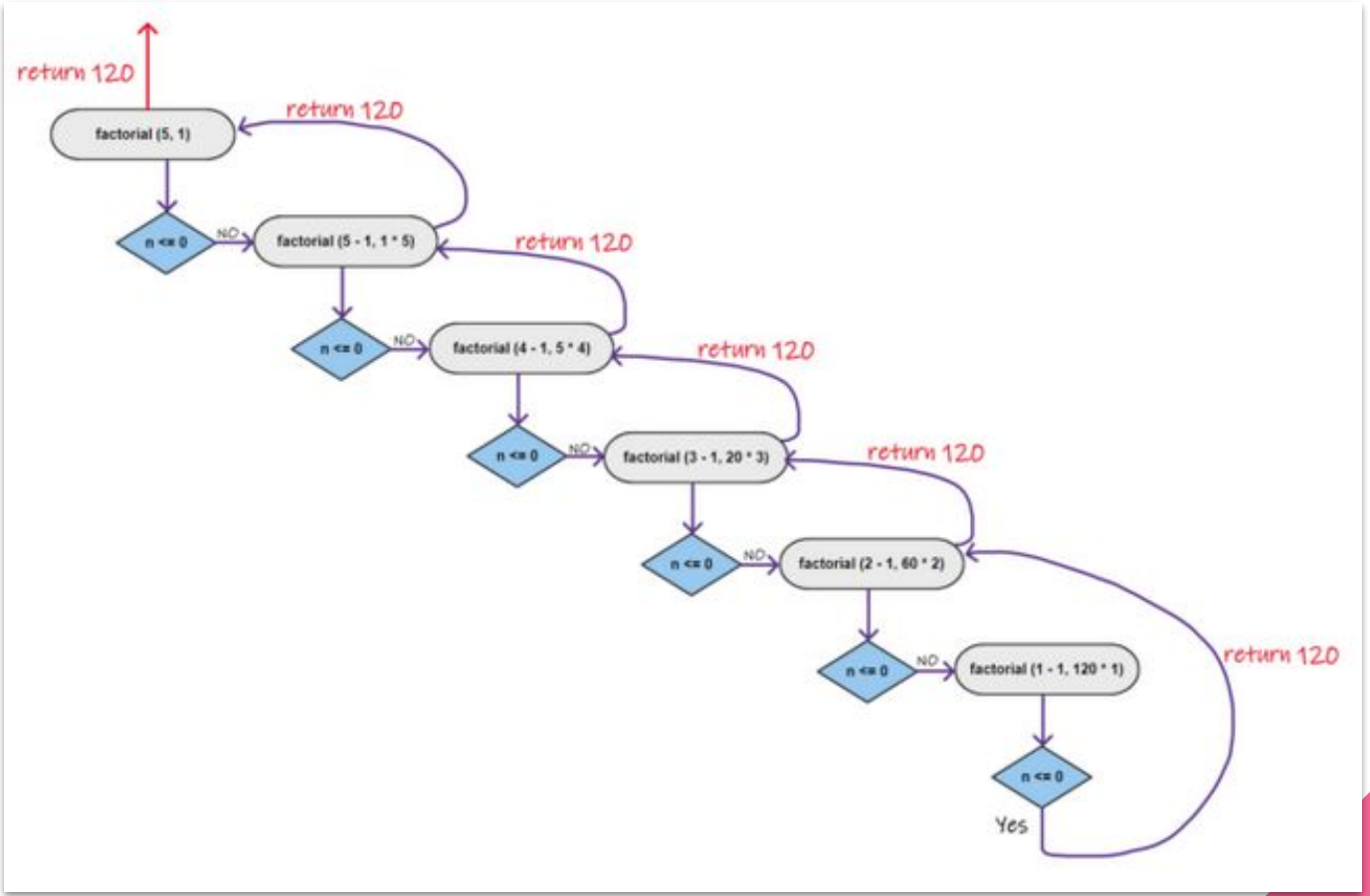
The **function directly returns the result** of a recursive call without performing any operations on it.

```
public static int fact(int x, int prevMult){
    if (x <= 0)
            return prevMult;
    else
        return fact(x - 1, x * prevMult);
}
```

The **prevMult** parameter accumulates the product as the recursion progresses.

The base case stops when x <= 0 and returns the accumulated product (prevMult). To ensure the multiplication starts correctly, **prevMult should be initialized to 1** because multiplying by 1 does not alter the result.

**Example:** fact(5, 1);



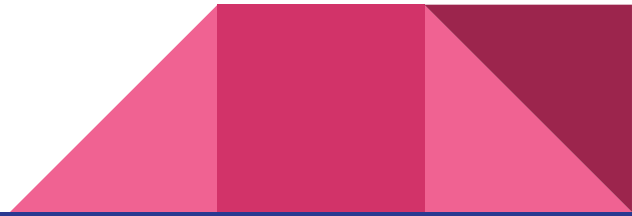

# Tail Recursion

# Non-Tail Recursion

In non-tail recursion, some operations must be performed after successfully executing a recursive function.

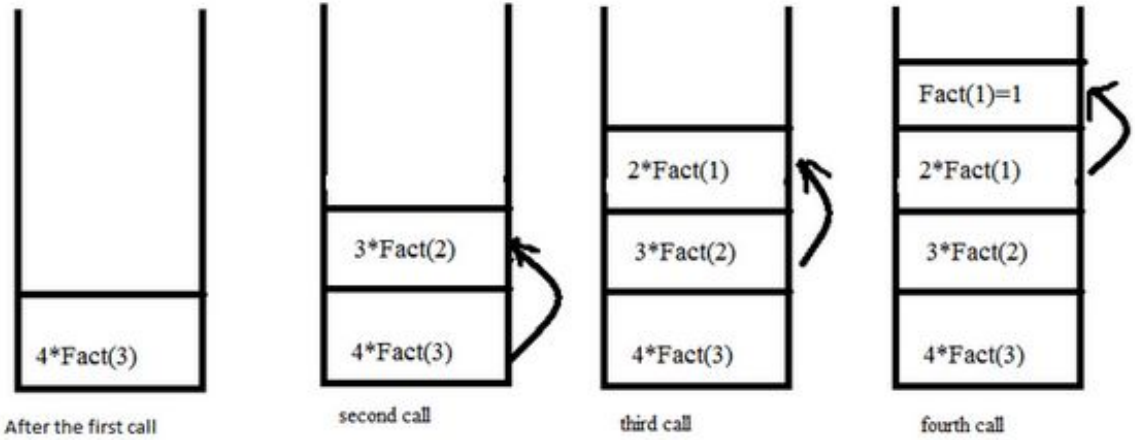
The function **never directly returns the result of a recursive call**. It performs some operations on the returned value of the recursive call to achieve the desired output.

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

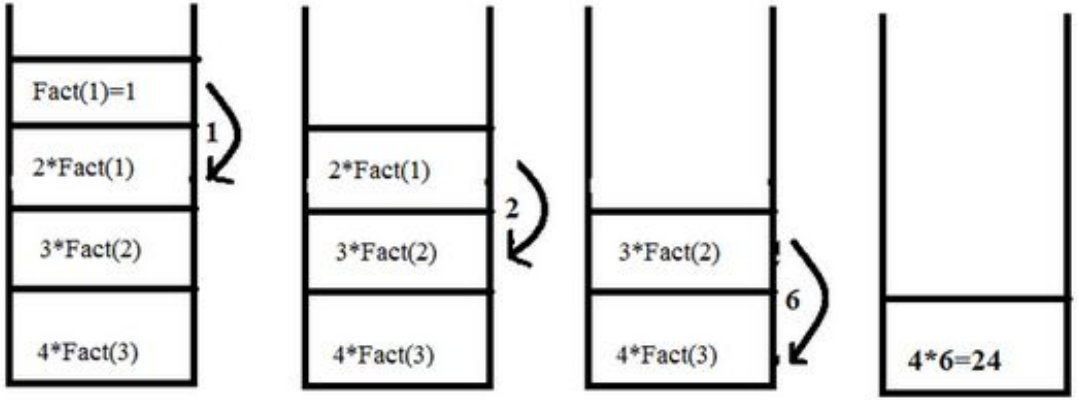


# Non-Tail Recursion

**When function call happens previous variables gets stored in stack**




**Returning values from base case to caller function**



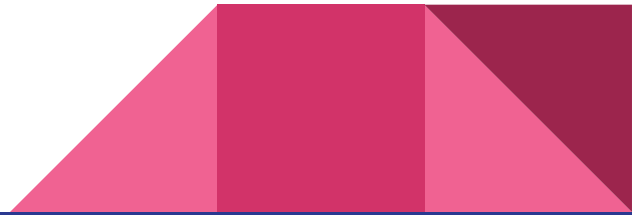
# Tail Recursion vs. Non-Tail Recursion

**Tail recursion is considered better than non-tail** recursion because tail recursive functions can be **optimized by modern compilers**.

**Tail recursion:** modern compilers know that since it is tail recursive and no other operations will be performed after the recursive call, there is **no need to maintain a stack**. Thus retaining the current function call in the stack frame is of no use. **It avoids stack overflow when calling a recursive method.**

**Non-tail recursion fully utilizes the stack** frame and uses the value returned from the recursive call. **JVM must retain all the stack frames**, no matter how many they are, to compute the end result correctly. This leads to memory overuse and sometimes results in errors.

**Java compiler does not currently optimize tail recursion.**



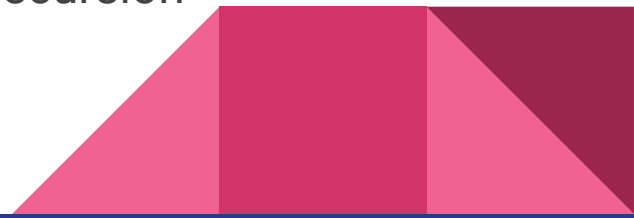
# Convert a Non-Tail Recursion to a Tail Recursion?

It is possible to make the conversion b**y passing additional parameter to maintain the state of the recursive call.**

The idea is to maintain the result of calculation throughout the recursive calls, from start to end. Do not wait to compute the result until the terminal condition is met.

```
public static int factorial(int n) ;                    // Non-tail recursion

public static int factorial(int n, int prevMult) ;      // Tail recursion
```



# How to process a String?

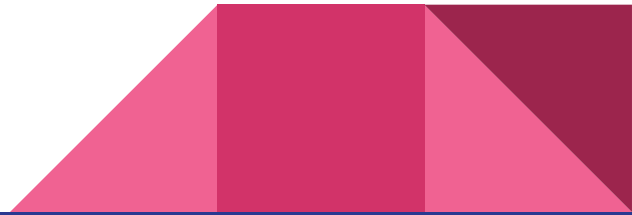
**You know:**

"A string is set of characters, so we can say a string is a **character followed by a string**."

Let's say myStr is a string variable, and  you have to write a recursive function that examines the character in that string.

**Strategy:**

1) Your recursive method should do something with the first element.
2) Use the rest of the string to make a recursive call to process it.

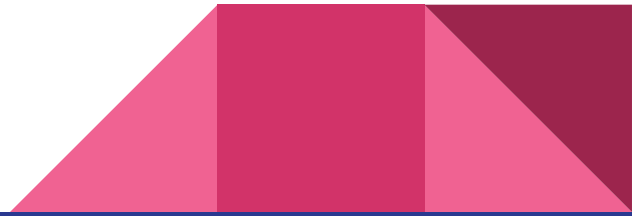


# Use recursion to find the length of a String

```
public static int myLen(String s){
        if (s.equals(""))         // Base Case
                ???
        else                      // Recursive Case
                ???
}
```

# Solution: myLength

```
public static int myLen(String s){
    if (s.equals(""))
            return 0;
    else
            return myLen(s.substring(1)) + 1;
}
```



# Step by step

**myLen("dog")**
s = "dog"
myLen("og") + 1

**myLen("og")**
s = "og"
myLen("g") + 1

**myLen("g")**
s = "g"
myLen("") + 1

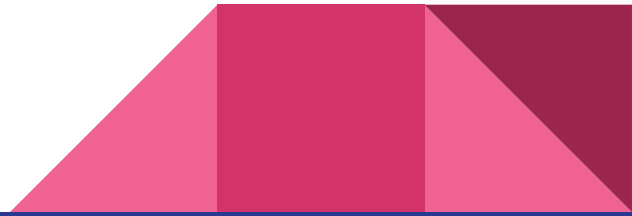
**myLen("")**
s = ""
Base case => return 0

```
public static int myLen(String s){
    if (s.equals(""))
        return 0;
    else
        return myLen(s.substring(1)) +1;
}
```

# Example: pairStar

Given a string, compute recursively a new string where identical chars that are adjacent in the original string are separated from each other by a "*".

pairStar("a") → "a"
pairStar("hello") → "hel*lo"
pairStar("xxyy") → "x*xy*y"
pairStar("aaaa") → "a*a*a*a"



pairStar("a") → "a
pairStar("hello") → "hel*lo"
pairStar("xxyy") → "x*xy*y"
pairStar("aaaa") → "a*a*a*a"

**Strategy:**

1) Your recursive method should do something with the first element.
2) Use the rest of the string to make a recursive call to process it.

1. What should be the base case? Hint: string length, what value should it be?
2. Recursive cases? Hint: Look at the strategy

```
public String pairStar(String str) {
    if(str.length() <= 1)
        return str;
    else if(str.charAt(0) == str.charAt(1))
        return str.charAt(0) + "*" + pairStar(str.substring(1));
    else
        return str.charAt(0) + pairStar(str.substring(1));
}
```

# Classwork

https://codingbat.com/home/jnovillo@stuy.edu/recursion_2_strings

https://codingbat.com/home/jnovillo@stuy.edu/recursion_3

Extra Problems:

https://codingbat.com/home/jnovillo@stuy.edu/apcsa_extra_recursion