

# 159 XTENDED

## A WEEKLY REVIEW

### RECURSION

Pull up [Socrative.com](https://www.socrative.com)

and join room **XTEND159**

IF YOU CAN SOLVE A PROBLEM WITH A  
LOOP, DO IT.

Computer Scientists Everywhere

Consider this code...

```
public static void countdown(int n)
{
    if (n == 0)
    {
        System.out.print("Blastoff!");
    } else {
        System.out.print(n + " ");
        countdown(n - 1);
    }
}
```

What is the output of `countdown(4)`?

Pull up Socrative.com  
and join room **XTEND159**

## WHAT IS RECURSION?

- ▶ A Recursive Method is a method that calls itself.  
**That's it.**
- ▶ It consists of two things:  
A Recursive Call (when the method calls itself)  
and a Base Case (a stopping point)

## FROM OUR PREVIOUS EXAMPLE...

```
public static void countdown(int n)
{
    if (n == 0) This is our base case.
    {
        System.out.print("Blastoff!");
    }
    else {
        System.out.print(n + " ");
        countdown(n - 1);
    }
}
```

This is our recursive call

## DEFINING RECURSIVE CALLS

- ▶ A recursive call is a call to the current method.
- ▶ It should make progress towards a base case.

## DEFINING BASE CASE

- ▶ A base case is the stopping point for a recursive method.
- ▶ For certain inputs, the method will not perform a recursive call.
- ▶ Any chain of recursive calls must reach a base case, otherwise we will get a stack overflow exception.

**Consider this code...**

```
1 public static void fibonacci(int n)
2 {
3     if (n == 1 || n == 2)
4     {
5         return 1;
6     }
7     return fibonacci(n - 1) + fibonacci(n - 2);
8 }
```

**State the base case(s).**

Pull up Socrative.com  
and join room **XTEND159**

Consider this code...

```
1 public static void fibonacci(int n)
2 {
3     if (n == 1 || n == 2)
4     {
5         return 1;
6     }
7     return fibonacci(n - 1) + fibonacci(n - 2);
8 }
```

Pull up Socrative.com  
and join room **XTEND159**

State the recursive call(s).

## FROM ITERATION TO RECURSION

- ▶ Many problems that can be solved iteratively (with a loop) can also be solved recursively.
- ▶ Let's consider turning an iterative solution to a problem in to a recursive one.
- ▶ Factorial: The product of an integer and all integers below it.

$$3! = 3 \bullet 2! = 3 \bullet 2 \bullet 1!$$

## FROM ITERATION TO RECURSION

```
public static int factorial_v1(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

This method computes the product of a positive integer  $n$  and all integers beneath it.

$$\text{factorial}(n) = n * n-1 * n-2 * n-3 * \dots * 1$$

## FROM ITERATION TO RECURSION

```
public static int factorial_v1(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

Now, consider this method:

```
public static int factorial_v2(int n)
{
    return n * factorial_v1(n-1);
}
```

Note that  $8!$  is the same as  $8 * 7!$

If we know that `factorial_v1` correctly computes the nth factorial, then we know `factorial_v2` will work.

## FROM ITERATION TO RECURSION

```
public static int factorial_v1(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
```

```
public static int factorial_v2(int n)
{
    return n * factorial_v1(n-1);
}
```

Consider another one:

```
public static int factorial_v3(int n)
{
    return n * factorial_v2(n-1);
}
```

Note that  $8!$  is the same as  $8 * 7 * 6!$

If we know that `factorial_v2` correctly computes the nth factorial, then we know `factorial_v2` will work.

## FROM ITERATION TO RECURSION

```
public static int factorial_v1(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

Pull up Socrative.com and join room **XTEND159**

```
public static int factorial_v2(int n)
{
    return n * factorial_v1(n-1);
}
```

```
public static int factorial_v3(int n)
{
    return n * factorial_v2(n-1);
}
```

```
public static int factorial_v4(int n)
{
    // YOUR CODE HERE
}
```

Write a line of code that completes factorial\_v4.

## FROM ITERATION TO RECURSION

```
public static int factorial_v1(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

Pull up Socrative.com and join room **XTEND159**

```
public static int factorial_v2(int n)
{
    return n * factorial_v1(n-1);
}
```

```
public static int factorial_v3(int n)
{
    return n * factorial_v2(n-1);
}
```

```
public static int factorial_v4(int n)
{
    return n * factorial_v3(n-1);
}
```

Explain why factorial\_v4 works.

## FROM ITERATION TO RECURSION

Since these three methods  
are almost exactly the same,  
can I simplify them in to  
one?

```
public static int factorial_v2(int n)
{
    return n * factorial_v1(n-1);
}
```

```
public static int factorial_v3(int n)
{
    return n * factorial_v2(n-1);
}
```

```
public static int factorial_v4(int n)
{
    return n * factorial_v3(n-1);
}
```

## FROM ITERATION TO RECURSION

Since these three methods are almost exactly the same, can I simplify them in to one?

```
public static int factorial_v2(int n)
{
    return n * factorial_v2(n-1);
}
```

Problem: This method will run forever and cause a stack overflow exception.

Can we create a stopping point for the method?

## FROM ITERATION TO RECURSION

Let's use our factorial\_v1 method as a stopping point for factorial\_v2.

```
public static int factorial_v2(int n)
{
    return n * factorial_v2(n-1);
}
```

```
public static int factorial_v1(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

## FROM ITERATION TO RECURSION

Let's use our factorial\_v1 method as a stopping point for factorial\_v2.

```
public static int factorial_v2(int n)
{
    if (n <= 10)
    {
        return n * factorial_v1(n-1);
    }

    return n * factorial_v2(n-1);
}
```

```
public static int factorial_v1(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

## FROM ITERATION TO RECURSION

We know that factorial\_v1 will give us the correct result for any positive integer we give it.

We also know that  $1! = 1$ .

```
public static int factorial_v2(int n)
{
    if (n <= 10)
    {
        return n * factorial_v1(n-1);
    }

    return n * factorial_v2(n-1);
}
```

```
public static int factorial_v2(int n)
{
    if (n == 1)
    {
        return 1;
    }

    return n * factorial_v2(n-1);
}
```

## FROM ITERATION TO RECURSION

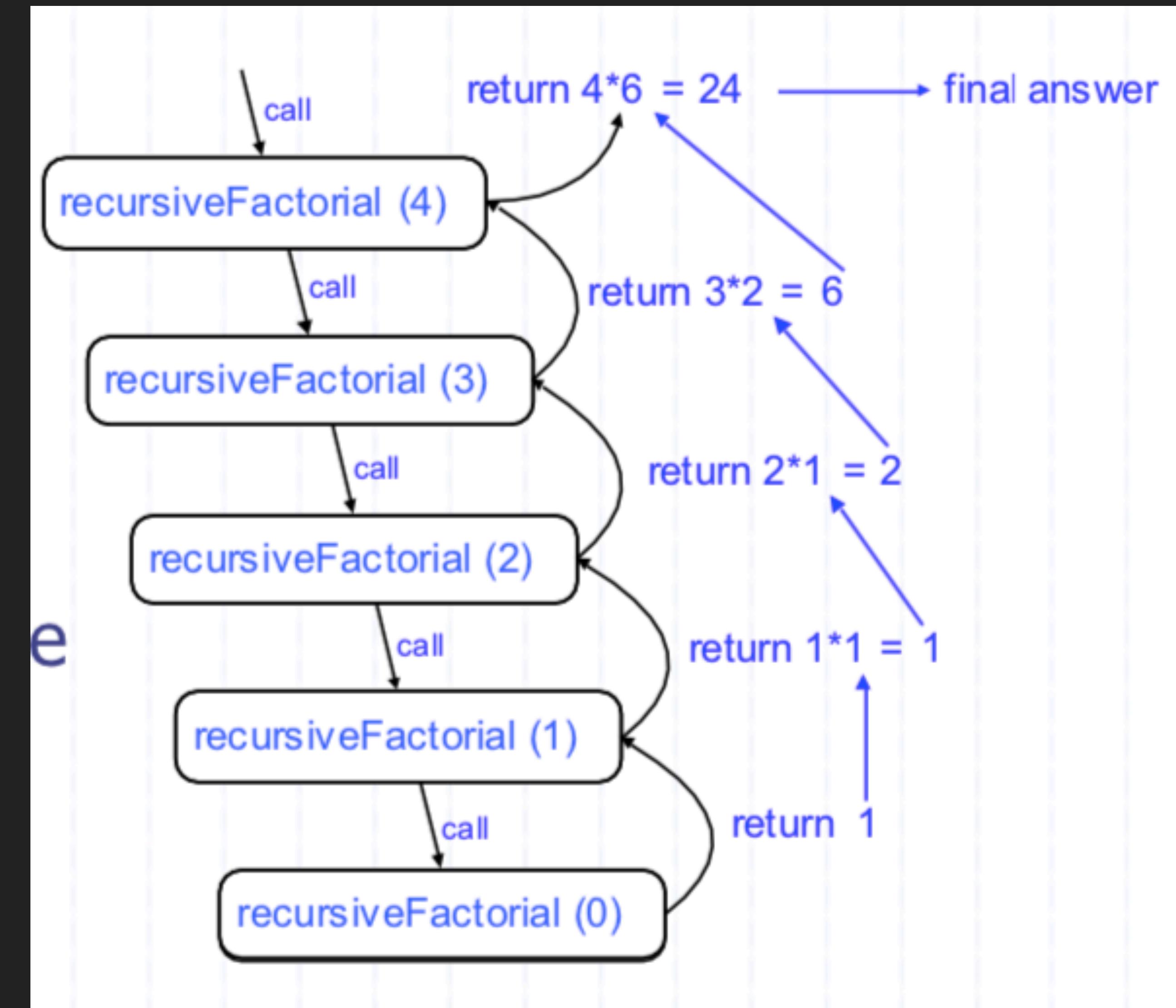
```
public static int factorial(int n)
{
    if (n == 1)
    {
        return 1;
    }
    return n * factorial(n-1);
}
```

Recursion is about  
breaking problems down in  
to smaller parts.

I know the  $n$  case works  
because the  $n-1$  case  
works.

## BACKTRACKING

- ▶ Each call to a recursive method gets its own box.
- ▶ An arrow is drawn from each calling function to each callee function.
- ▶ Another arrow shows the return value for each method call.
- ▶ This demonstrates the flow of execution of a recursive call.



## WATCH DR. FOX'S VIDEO!

- ▶ <https://www.youtube.com/watch?v=fb9yzbMC59g>



# Introducing...

# The Recursion Fairy

*She'll Solve ALL\* Your Problems!*

## EXPLAINING THE RECURSION FAIRY

- ▶ Sometimes, we can break a problem down into smaller versions of the same problem.
- ▶ Let's consider Factorial again.
- ▶ Factorial: The product of an integer and all integers below it.

$$3! = 3 \bullet 2! = 3 \bullet 2 \bullet 1!$$

...the recursion fairy can help solve this...

## EXPLAINING THE RECURSION FAIRY

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

We handle the first part of the problem...

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

...and then hand the rest off to the recursion fairy.

## ACTIVITY - DIVIDEBY2

Given a integer, return the number of times you need to divide by two before you reach the number 1.

```
public int divideByTwo(int n)
{
    // TODO
}
```

Example: divideByTwo(8) → 3

## ACTIVITY - DIVIDE BY 2

```
public int divideByTwo(int n)
{
    if (n <= 1)
    {
        return 0;
    }

    return 1 + divideByTwo(n / 2);
}
```

## ACTIVITY - COUNTX

Given a string, compute recursively (no loops) the number of lowercase 'x' chars in the string.

```
public int countX(String str)
{
    // TODO
}
```

HINT: You may find the methods `charAt(int index)`, `substring(int startIndex)`, and `length()` from the `String` class helpful in solving this problem.

## ACTIVITY - COUNTX

```
public int countX(String str)
{
    if (str.length == 0)
    {
        return 0;
    }

    if (str.charAt(0) == 'x')
    {
        return 1 + countX(str.substring(1));
    }

    return countX(str.substring(1));
}
```

## ACTIVITY - ARRAY11

Given an array of ints, compute recursively the number of times that the value 11 appears in the array. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass index+1 to move down the array. The initial call will pass in index as 0.

```
public int array11(int[] nums, int index)
{
    // TODO
}
```

Example:  $\text{array11}([1, 2, 11], 0) \rightarrow 1$

## ACTIVITY - ARRAY11

```
public int array11(int[] nums, int index)
{
    if (index == nums.length)
    {
        return 0;
    }

    if (nums[index] == 11)
    {
        return 1 + array11(nums, index + 1);
    }

    return array11(nums, index + 1);
}
```