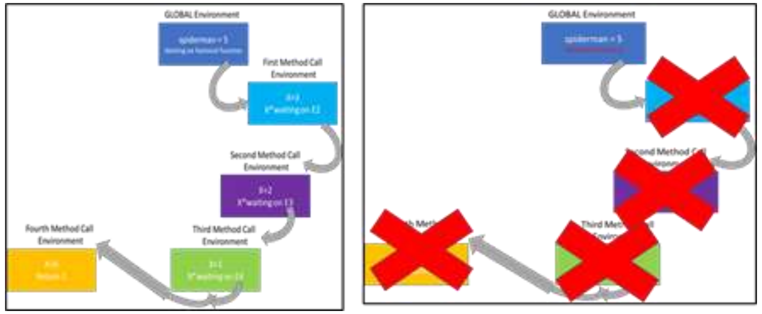
# CSE 1729:Principles of Programming

# Lecture 4: Recursion




Kaleel Mahmood
Department of Computer Science and Engineering
University of Connecticut


Recursion is horrifying!
…or is it?

# Factorial Function

$$n! = n * (n-1) * (n-2) * \cdots * 1$$

Can you code the factorial function in Python?

```
#Define the factorial function
def factorial(n):
    x = 1 #start with inital solution value
    for i in range(n, 1, -1): #Goes from n, n-1,... all the way to 1
        x = x * i #This is same as n*(n-1)...
    return x

solution = factorial(5)
print(solution)
```

# *Let’s figure out what is going on line by line…*

```
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

*Let's figure out what is going on line by line…*

```
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

*Let's figure out what is going on line by line…*

```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

Method Environment

*Let’s figure out what is going on line by line…*

```
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

Method Environment

x=1

*Let's figure out what is going on line by line…*

```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

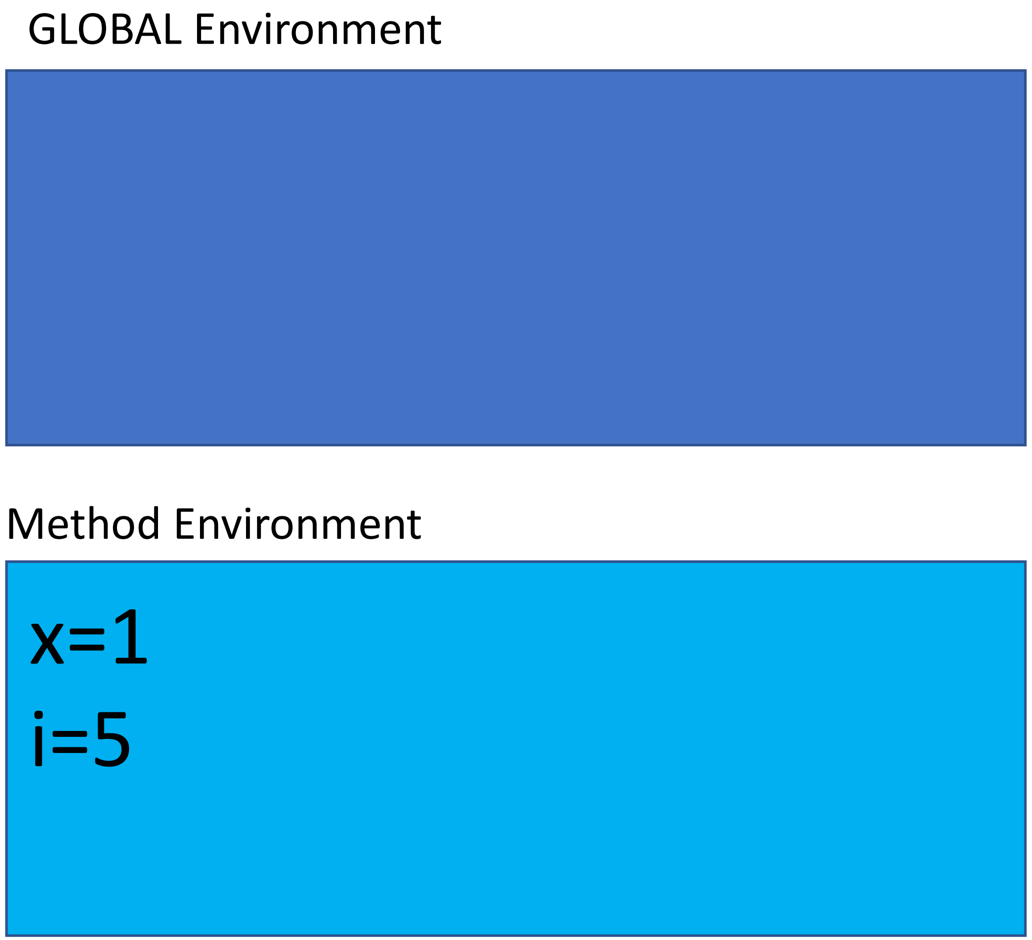
Method Environment

x=1

i=5

# *Let's figure out what is going on line by line…*

```
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```



# *Let's figure out what is going on line by line…*

```
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

Method Environment

x=5

i=5

*Let’s figure out what is going on line by line…*

```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

Method Environment

x=5

i=5

*Let's figure out what is going on line by line…*

```
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

Method Environment

x=5

i=4

*Let's figure out what is going on line by line…*

```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```

GLOBAL Environment

Method Environment

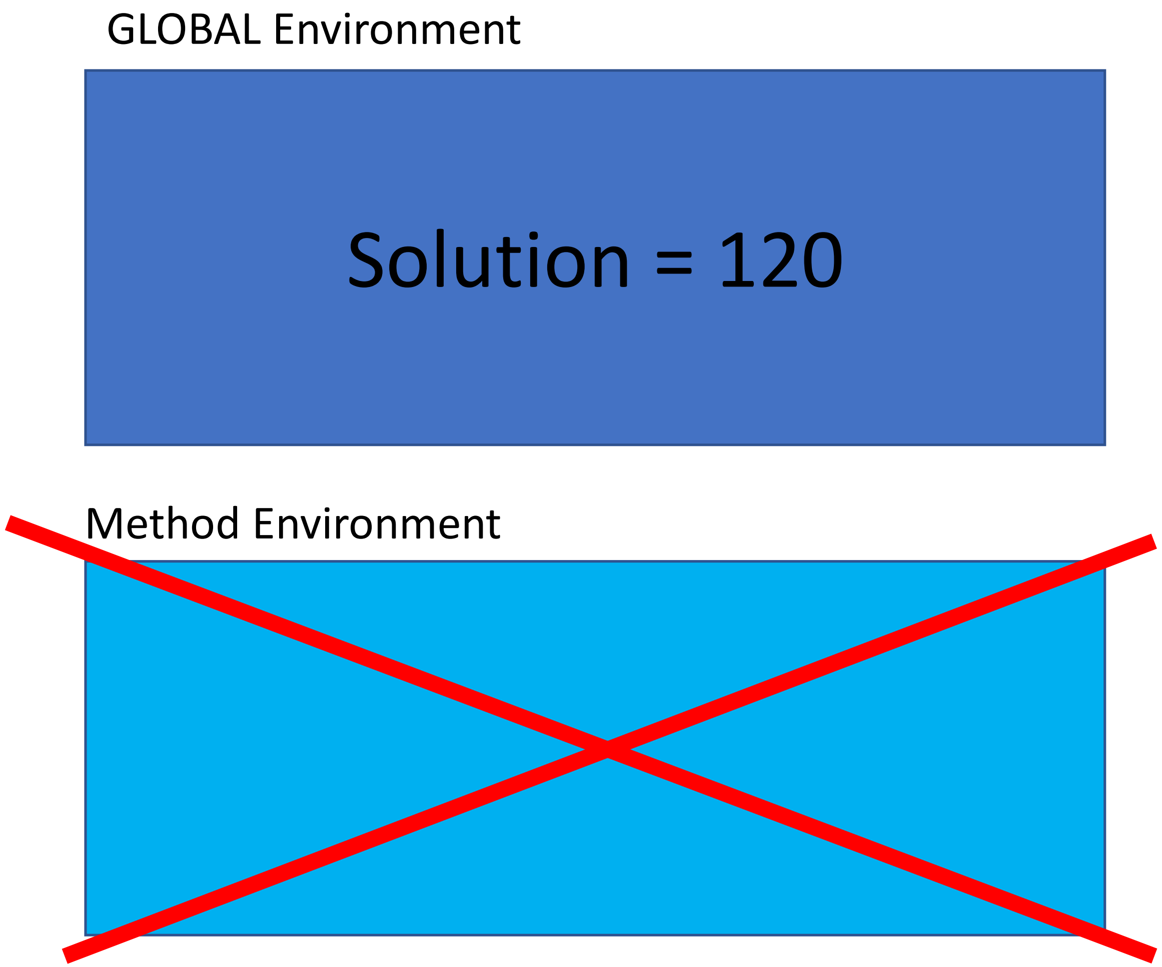
x=20

i=4


A FEW
MOMENTS LATER

# *Let's figure out what is going on line by line…*

```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x

solution = factorial(5)
print(solution)
```



Important Question: *How many local method environments were created?*

# Factorial Function

$$n! = n * (n-1) * (n-2) * \cdots * 1$$

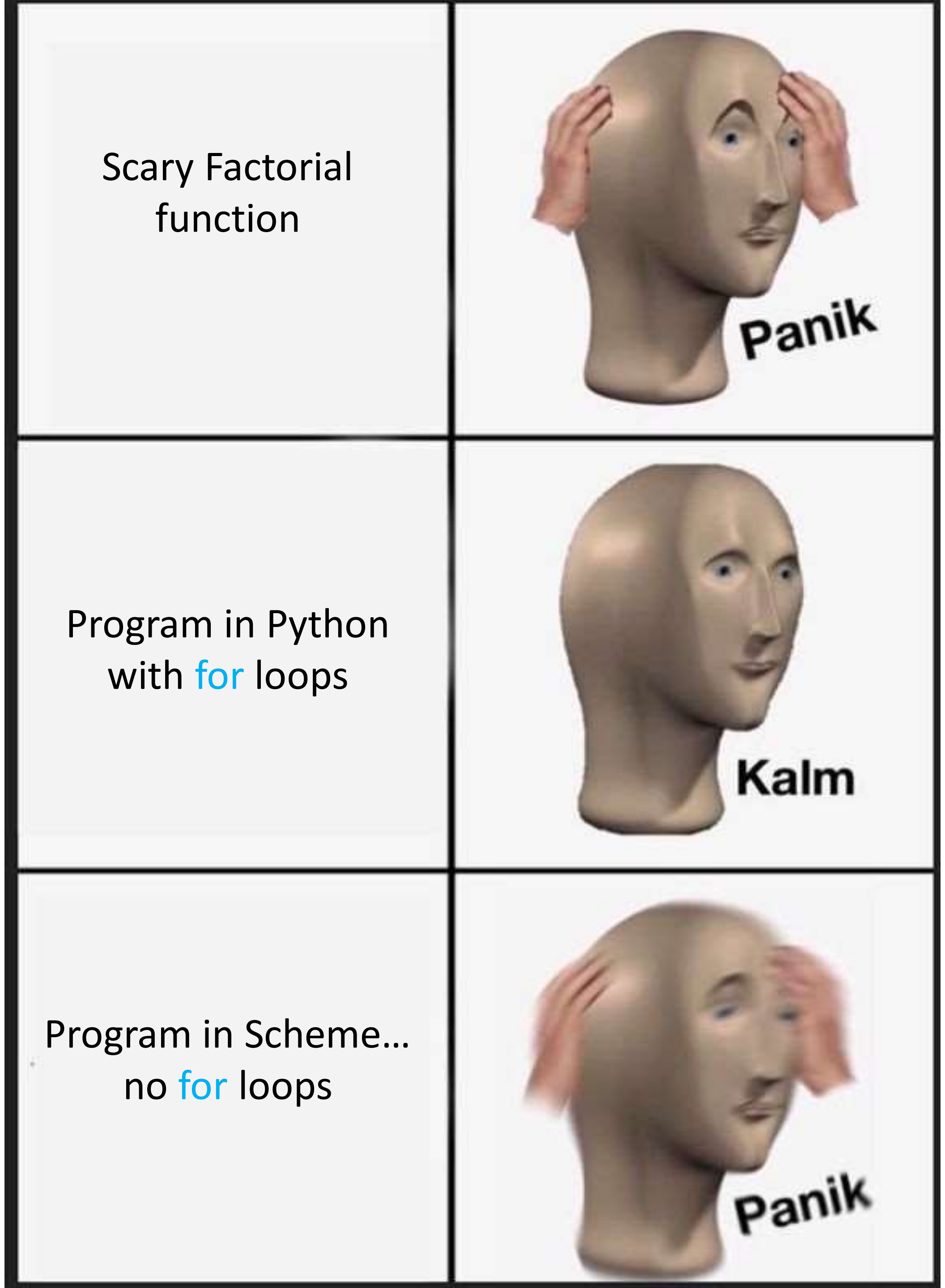
Can you code the factorial function in the same way in Scheme?

```python
#Define the factorial function
def factorial(n):
    x = 1 #start with inital solution value
    for i in range(n, 1, -1): #Goes from n, n-1,... all the way to 1
        x = x * i #This is same as n*(n-1)...
    return x

solution = factorial(5)
print(solution)
```

Do we know how to write functions in Scheme?

Do we know how to write multiplication in Scheme?

*For loops?*


Scary Factorial function
Panik
Program in Python with for loops
Kalm
Program in Scheme… no for loops
Panik

# When you don't have FOR loops…you must use recursion….

- Simple idea behind recursion:

1. Write a method and figure out a base case.
2. Reduce the problem until you reach the base case.
3. Then go from the base case back up.

How do you reduce the problem? By calling the SAME method repeatedly with smaller and smaller inputs.

# Step 1: Figure out the base case

$$n! = n * (n-1) * (n-2) * \cdots * 1$$

Think about some examples of the factorial function:

$$5! = 5 * 4 * 3 * 2 * 1$$

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

$$3! = 3 * 2 * 1$$

$$2! = 2 * 1$$

OK 1 is common among all computations…let’s make 1 our base case!

# Step 1: Figure out the base case

$$n! = n * (n-1) * (n-2) * \cdots * 1$$

```
(define (factorial x)
  (if (= x 0)
      1))
```


What if $n = 1$?

# Step 2: Reducing to the Base Case

$$n! = n * (n-1) * (n-2) * \cdots * 1$$

## What if $n = 1$?

```
(define (factorial x)
  (if (= x 0)
      1))
```

```
(define (factorial x)
  (if (= x 0)
      1)
  (if (= x 1)
      1)
  )
```


## What if $n = 2$?

# Step 2: Reducing to the Base Case

## What if $n = 2$?

```
(define (factorial x)
  (if (= x 0)
      1)
  (if (= x 1)
      1)
  )
```

```
(define (factorial x)
  (if (= x 0)
      1)
  (if (= x 1)
      1)
  (if (= x 2)
      2)
  )
```

Obviously this is a bad approach…

# THE FACTORIAL FUNCTION

- Recall the factorial function:

$$n! = n \cdot (n-1) \cdot \ldots \cdot 1$$

- Alternatively, we could write:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

# Step 2: Reducing to the Base Case

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

```
(define (factorial x)
  (if (= x 0)
      1)
  (if (= x 1)
      1)
  (if (= x 2)
      2)
  )
```

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )
```

# Recall the steps in designing a recursive function:

1. Write a method and figure out a base case.
2. Reduce the problem until you reach the base case.
3. Then go from the base case back up.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )
```

# Question: These two codes return the same values. Are they functionally the same?

```racket
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )
```

```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x
```


NOOOOOO!!

# Understanding the recursive calls…

GLOBAL Environment

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```

# Understanding the recursive calls…

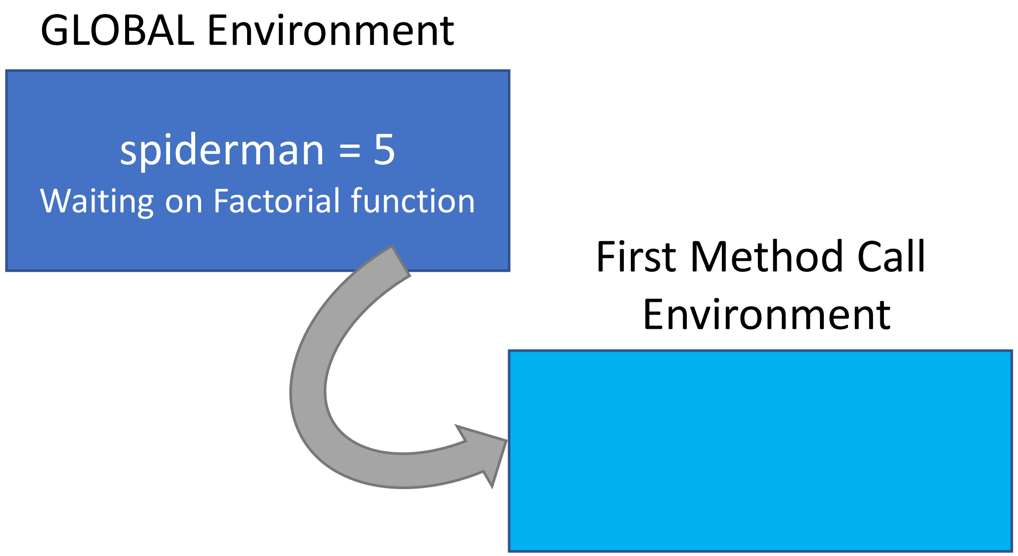
```scheme
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```

GLOBAL Environment

spiderman = 5

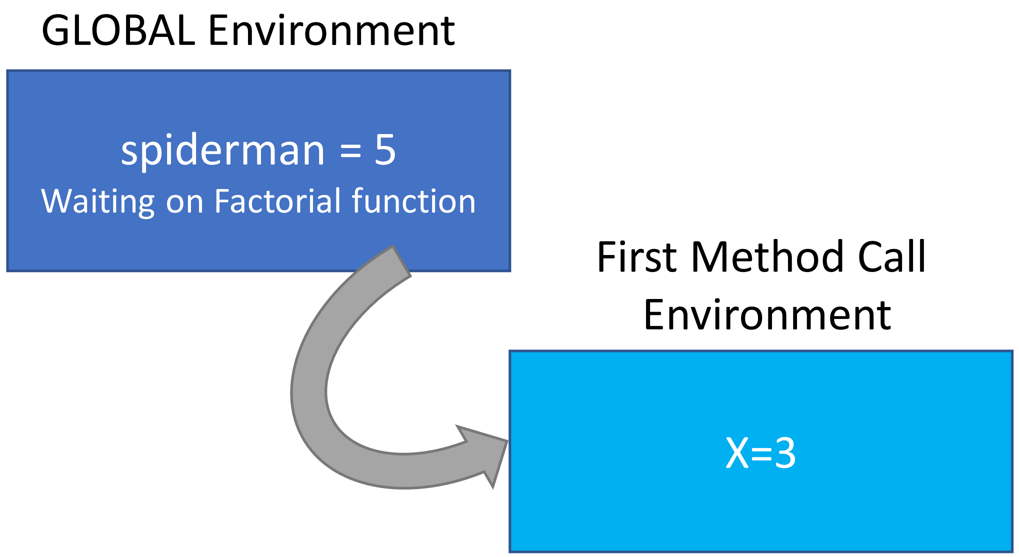
# Understanding the recursive calls...

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



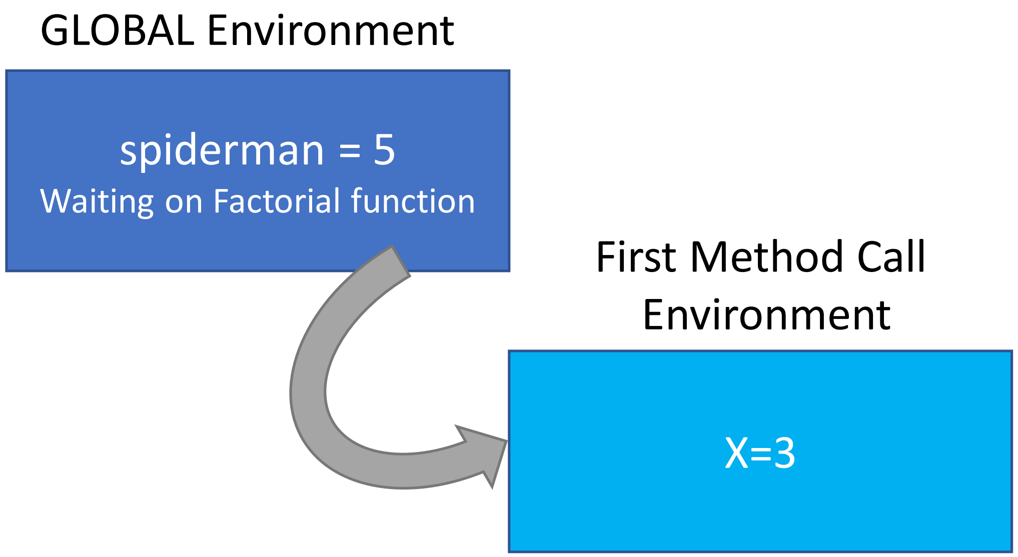
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



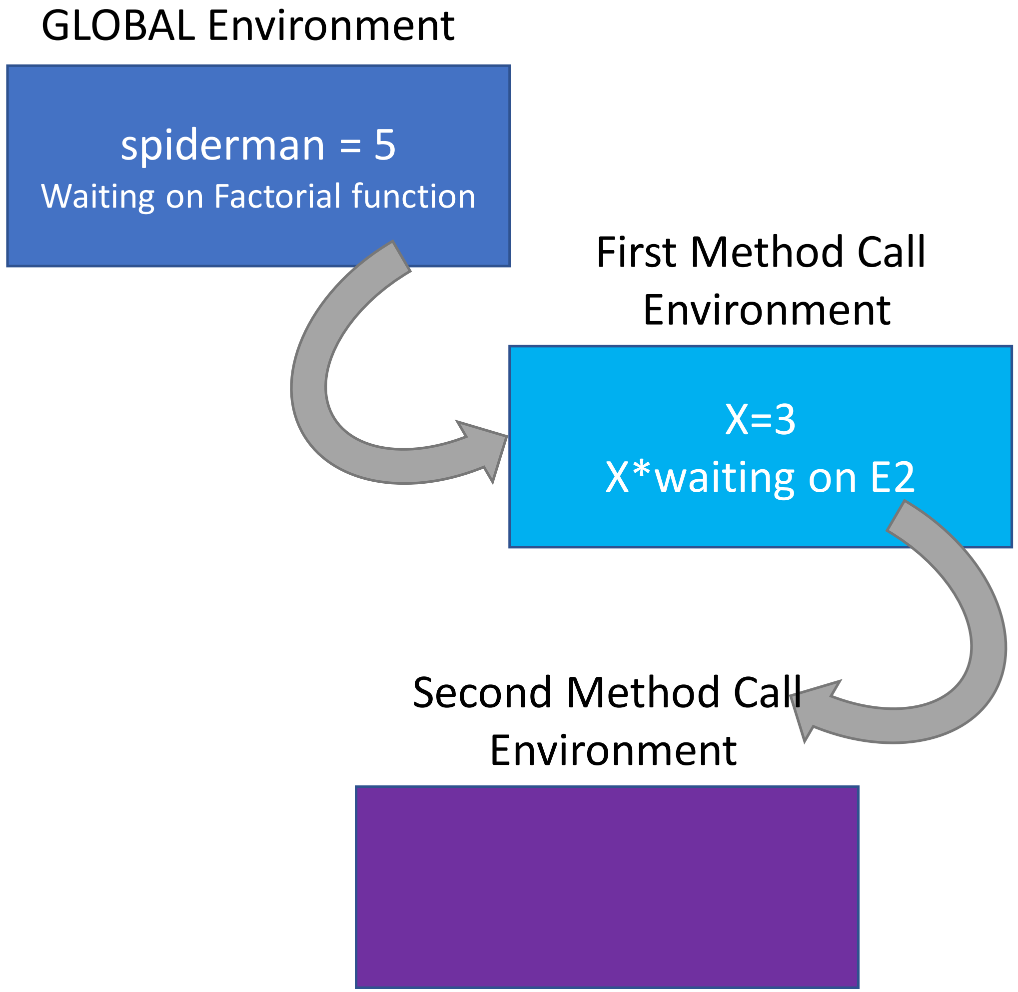
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



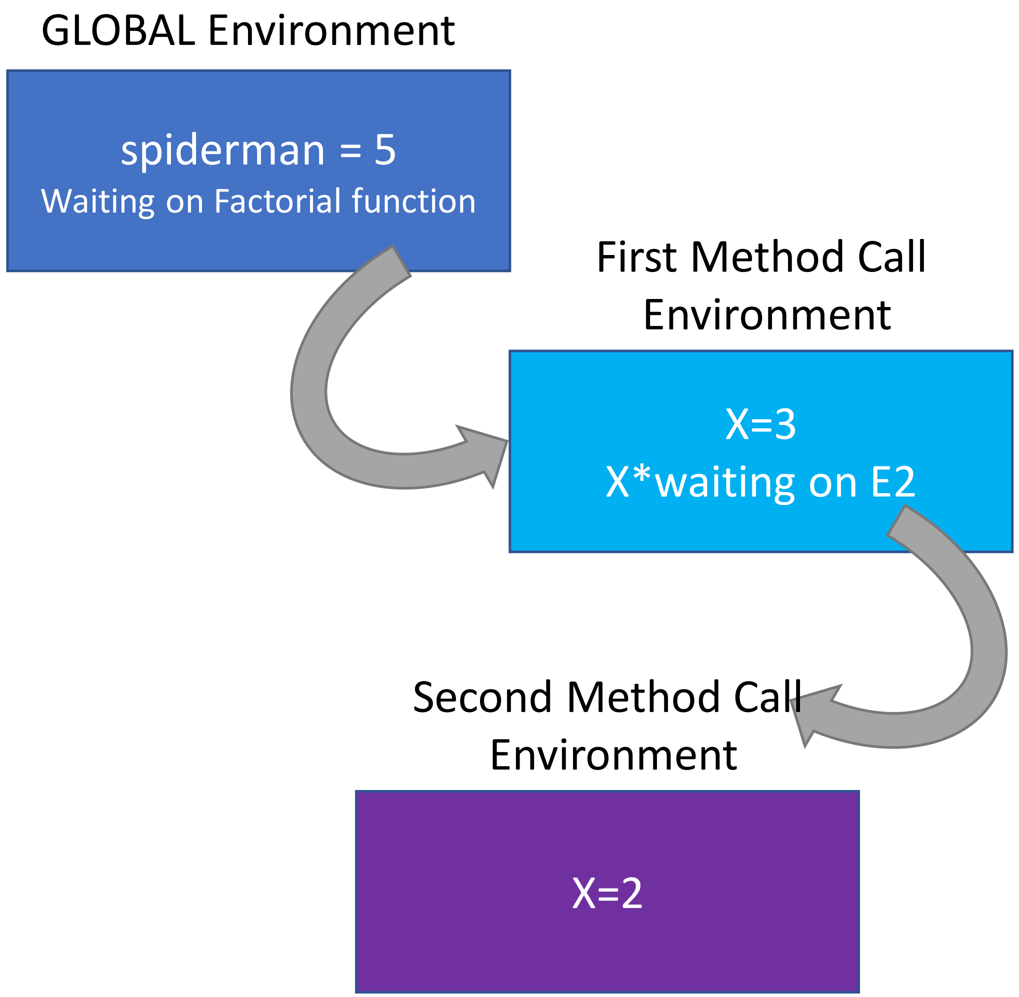
# Understanding the recursive calls...

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



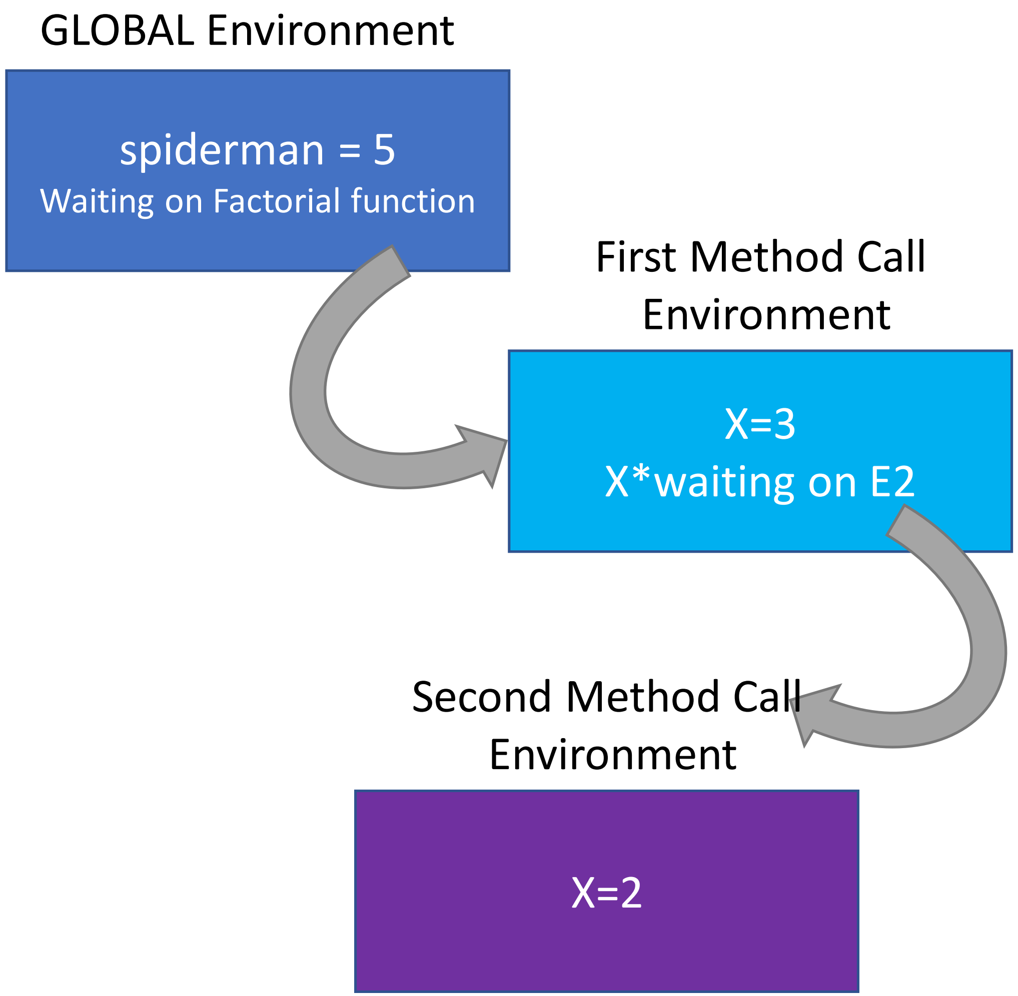
# Understanding the recursive calls...

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



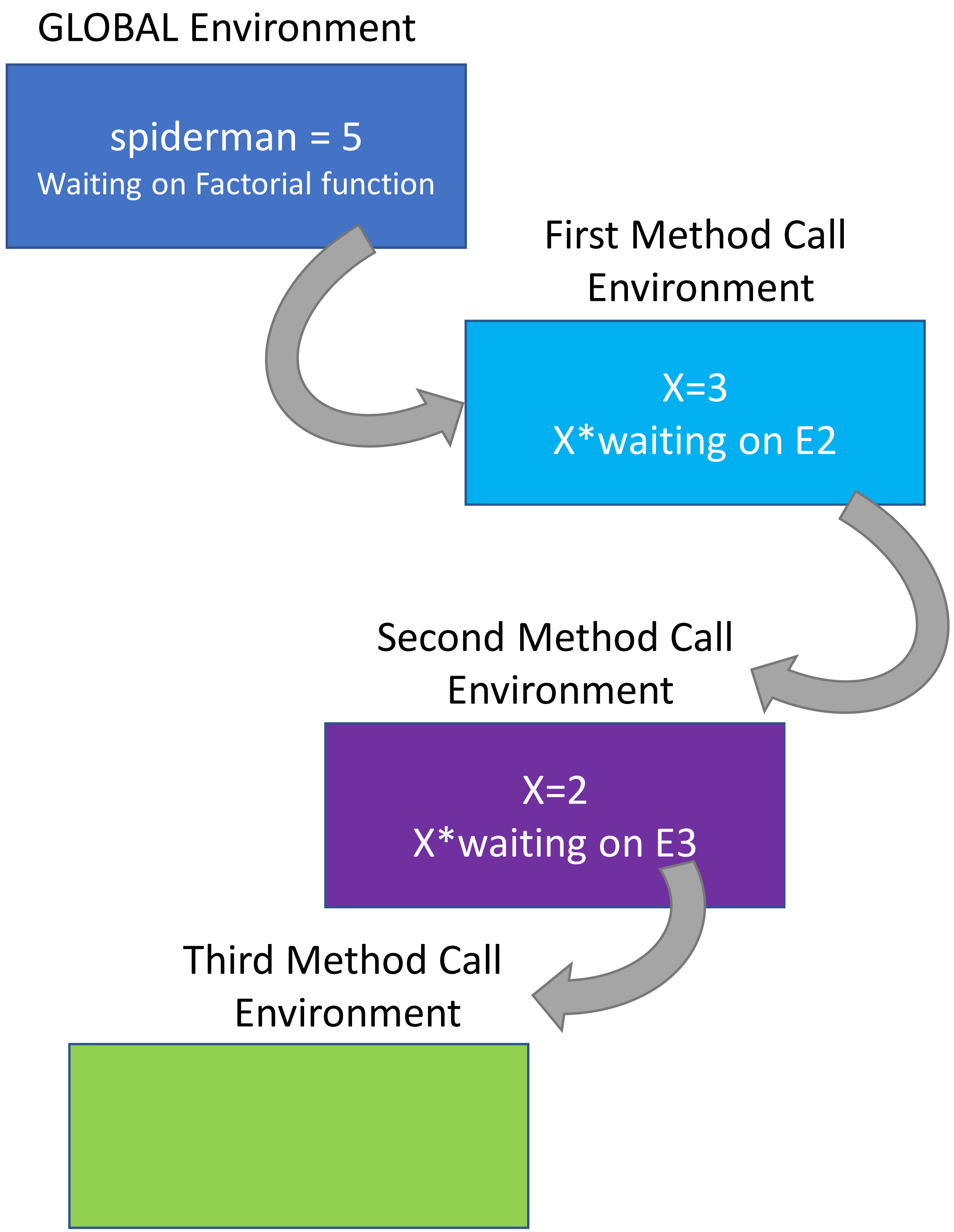
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



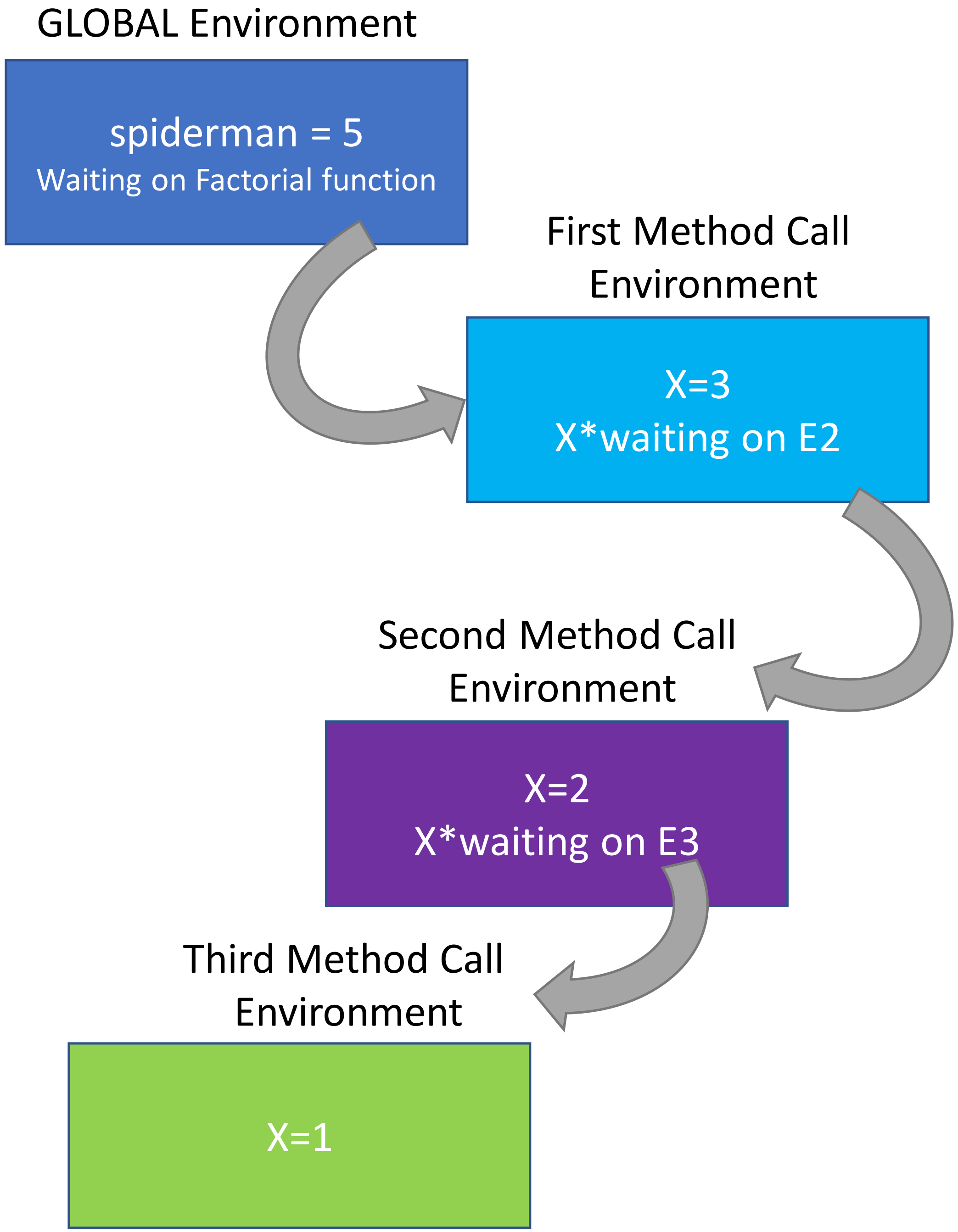
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



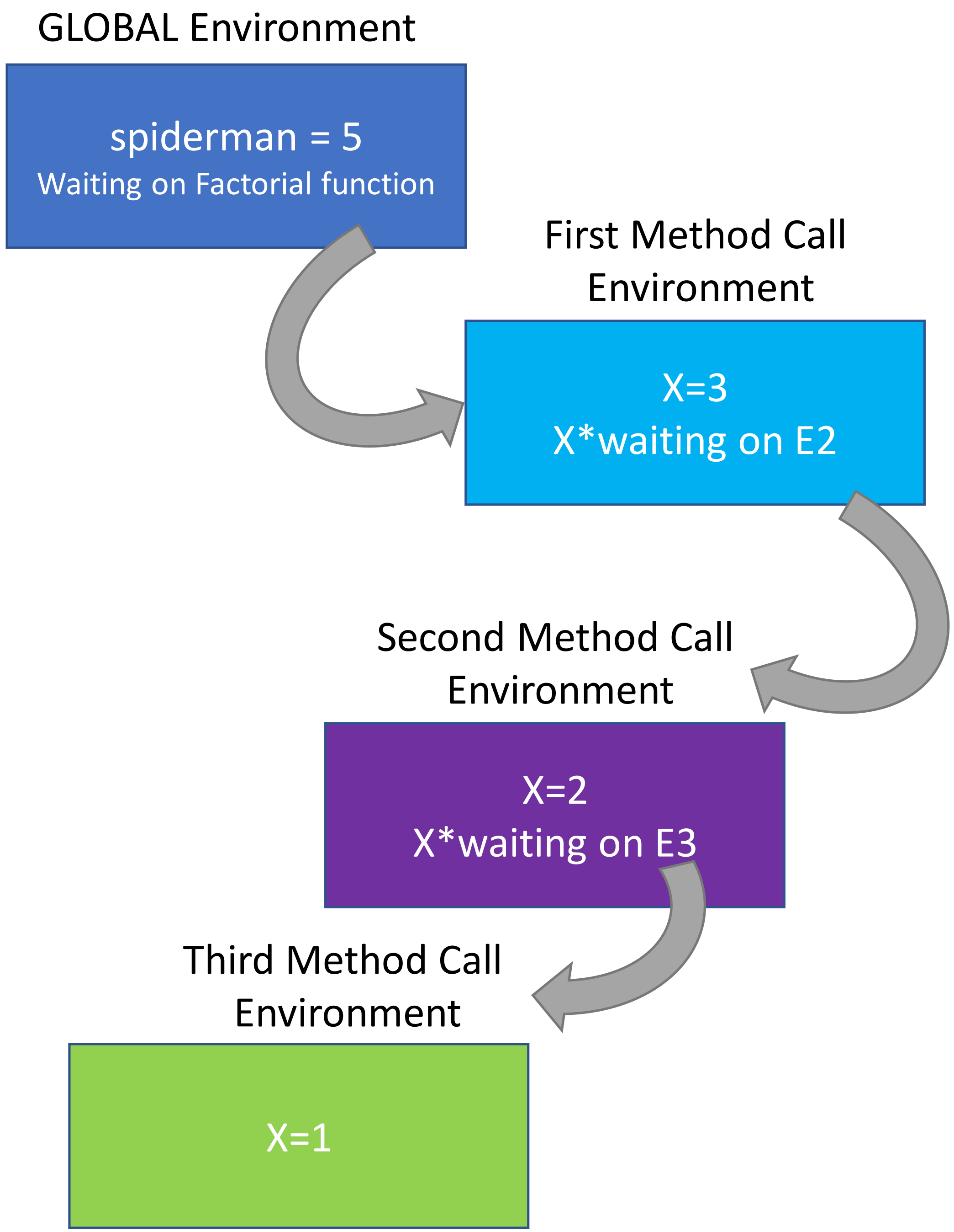
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



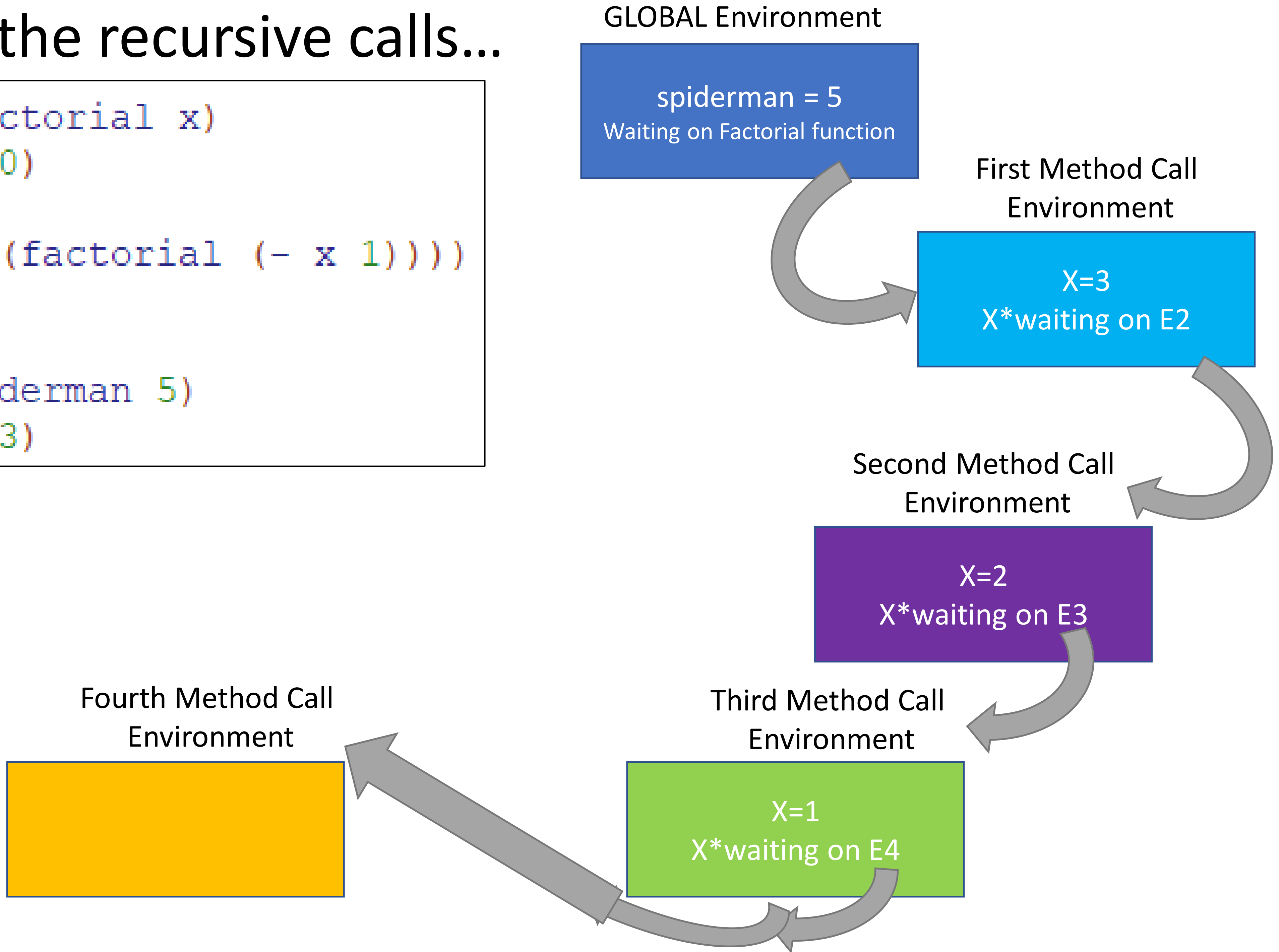
# Understanding the recursive calls...

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



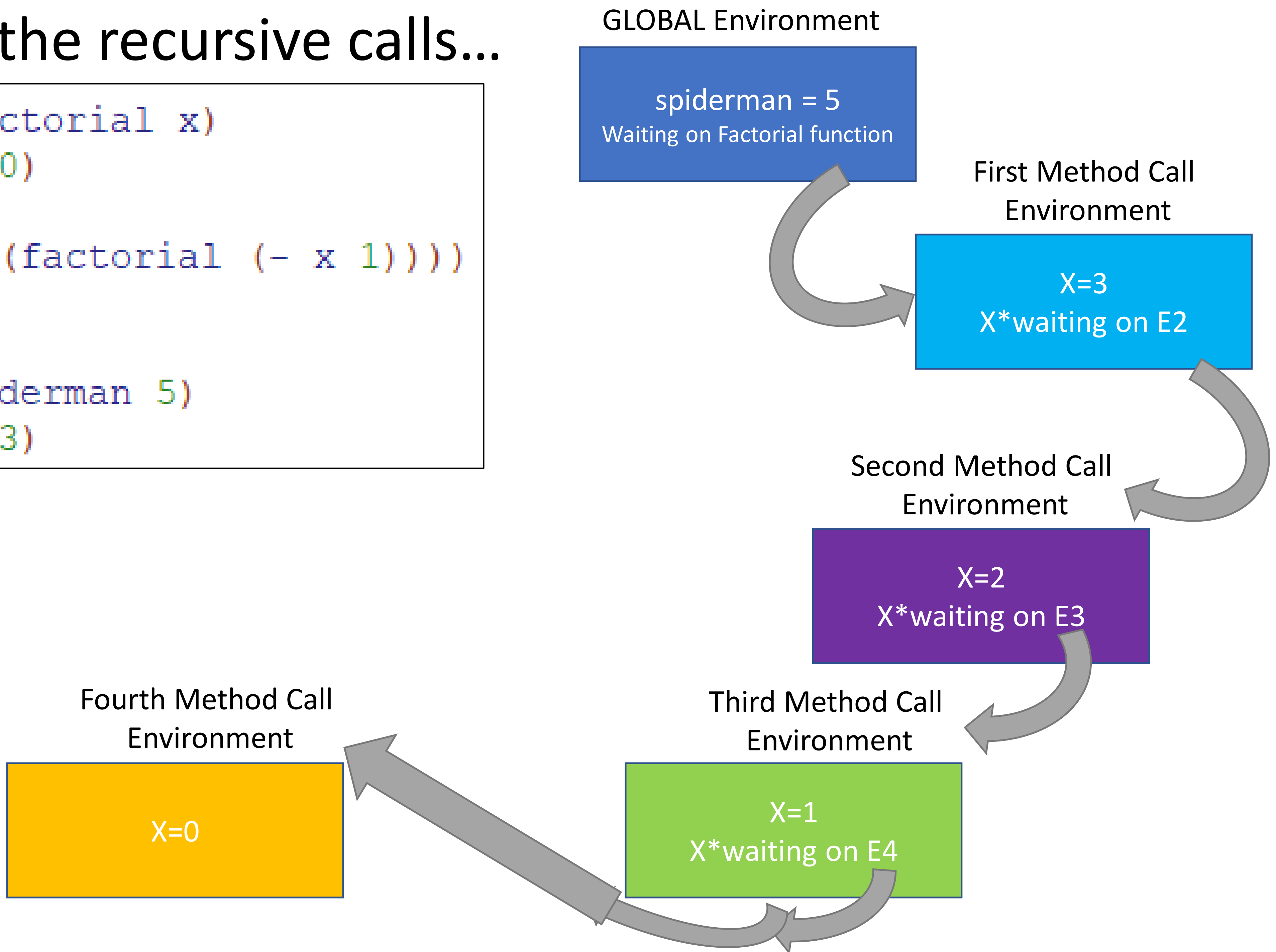
# Understanding the recursive calls...

```scheme
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



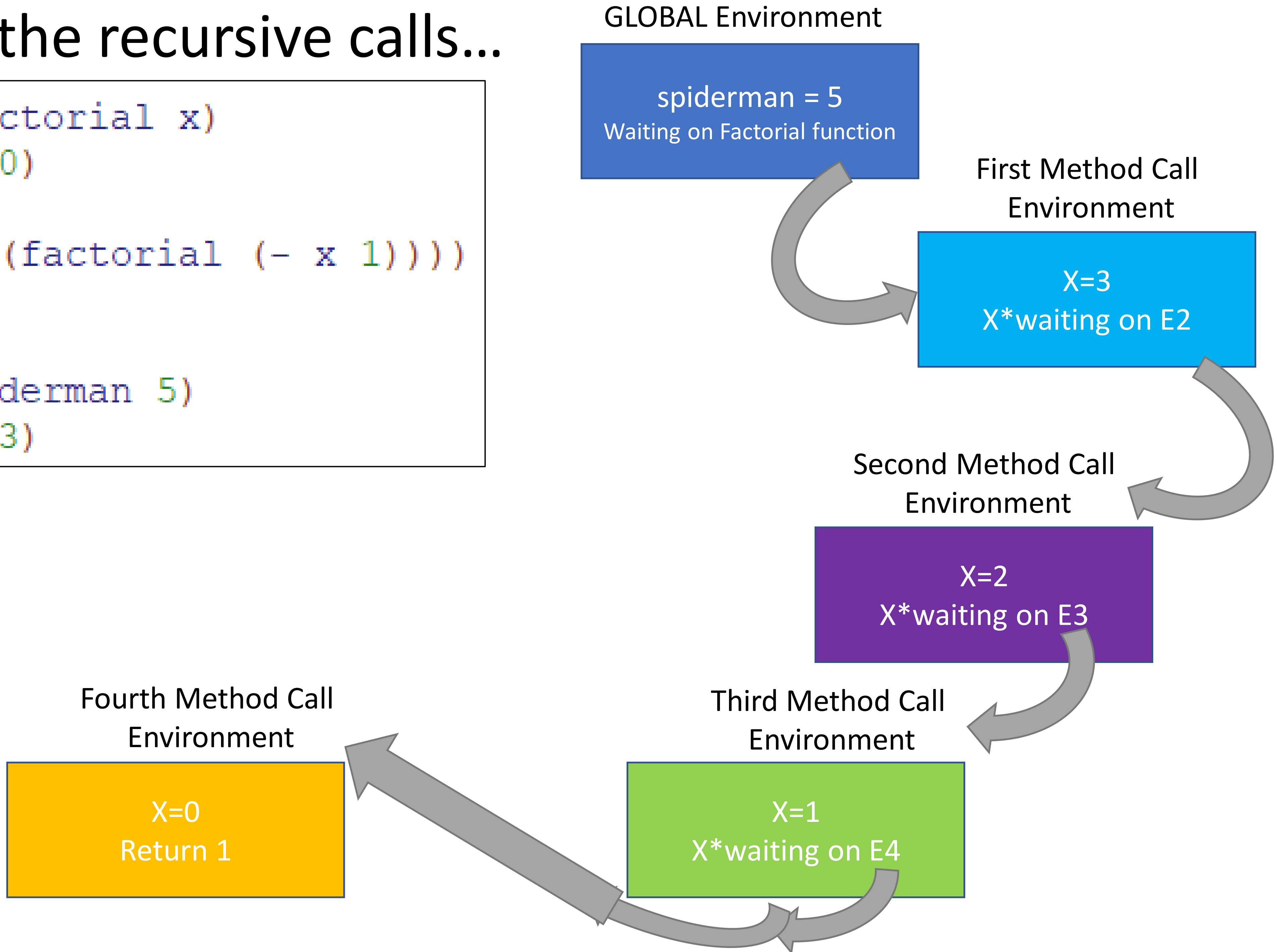
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



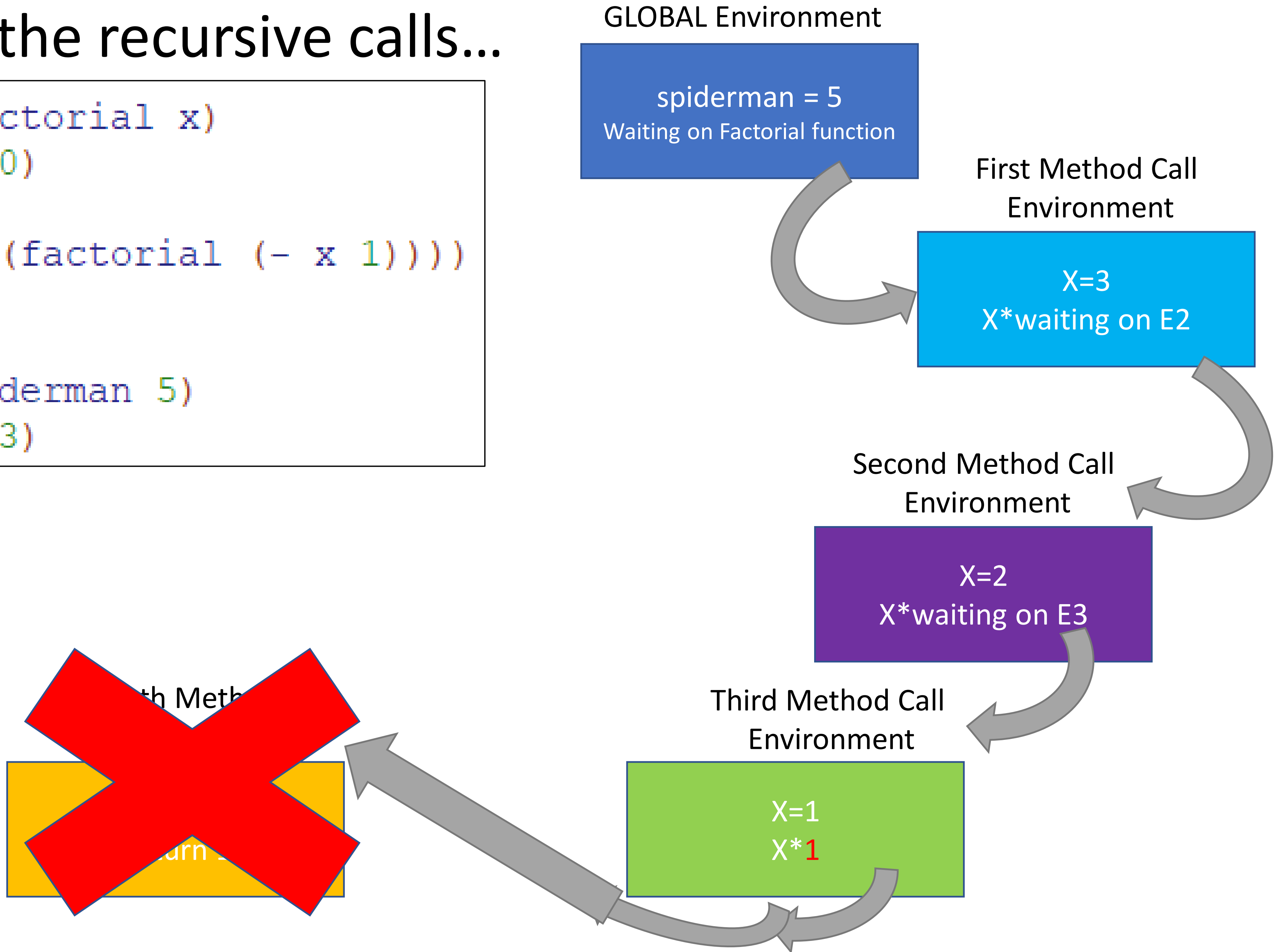
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



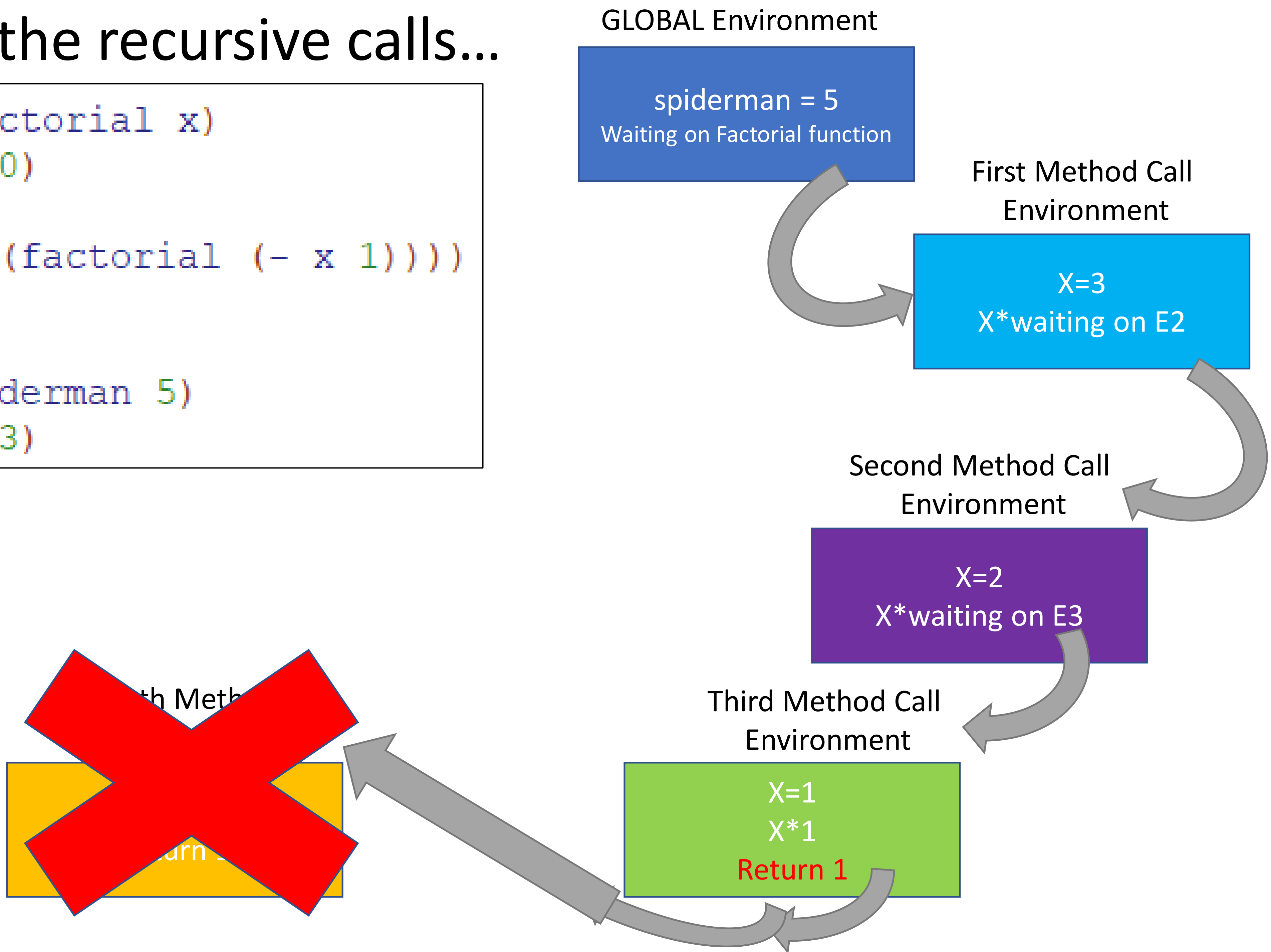
# Understanding the recursive calls...

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



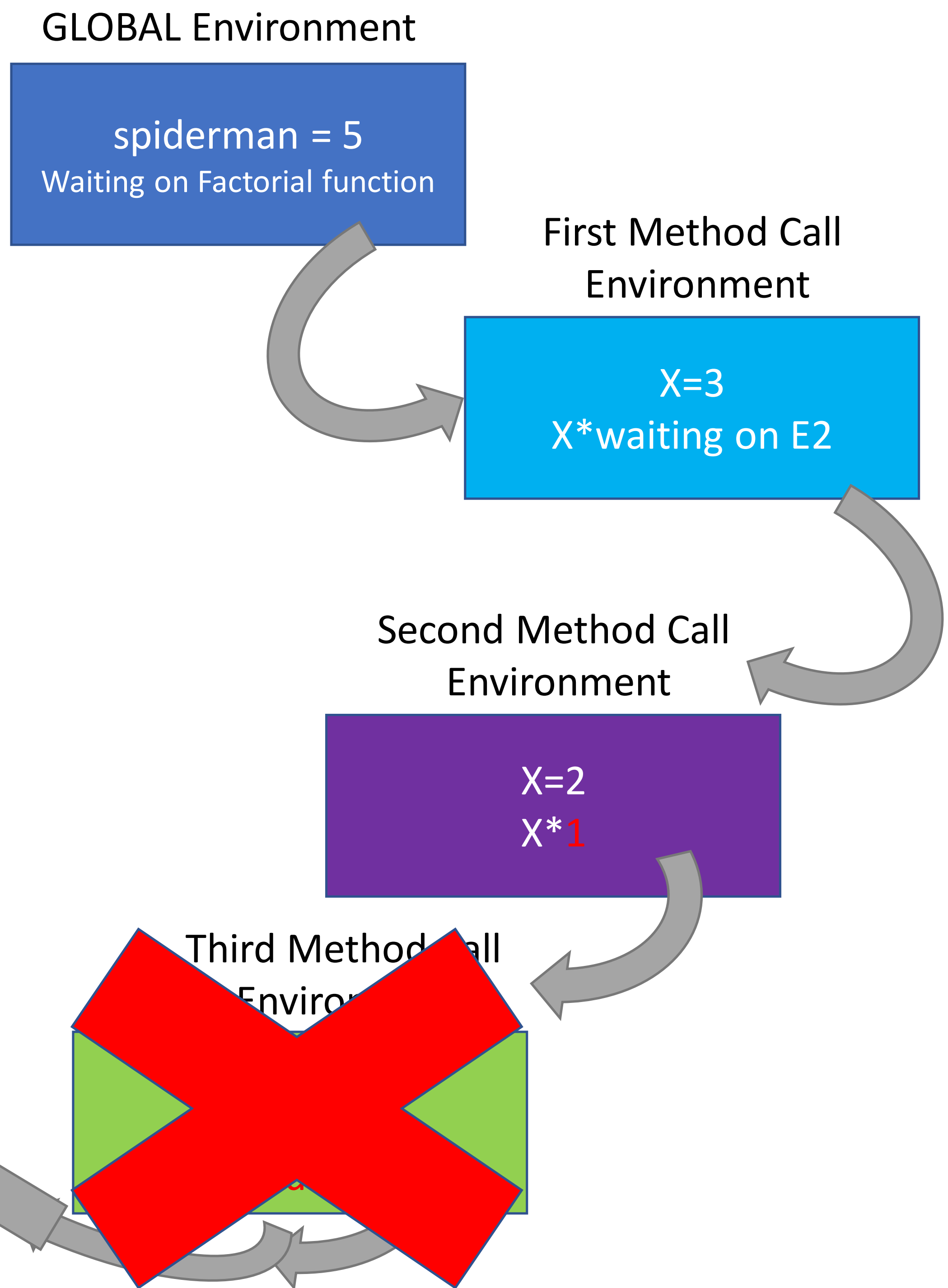
# Understanding the recursive calls…

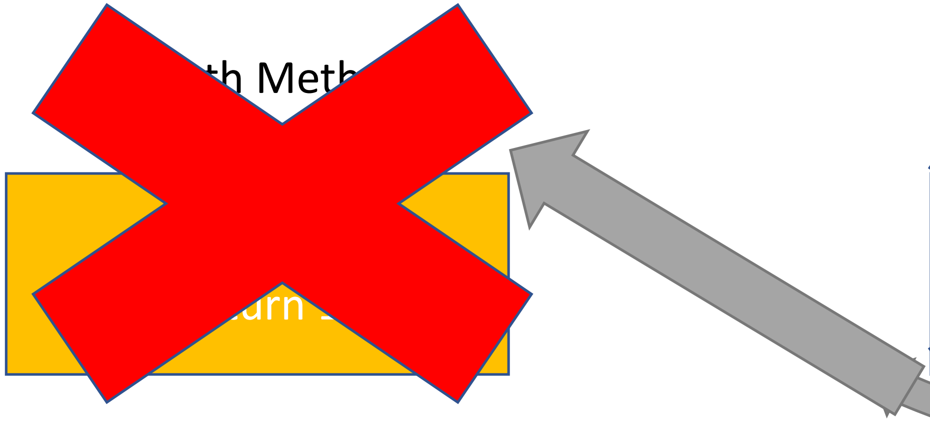
```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



# Understanding the recursive calls…

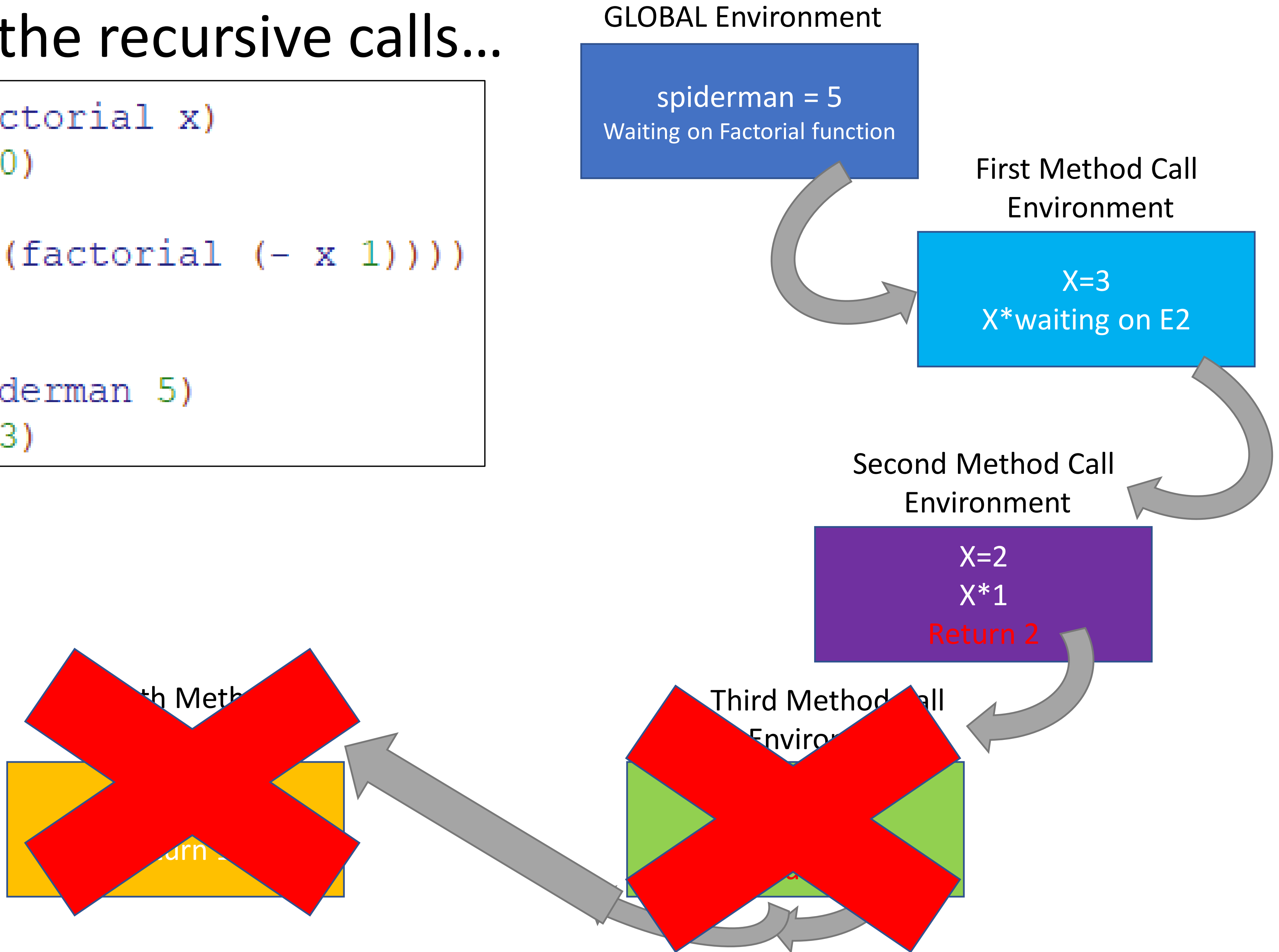
```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



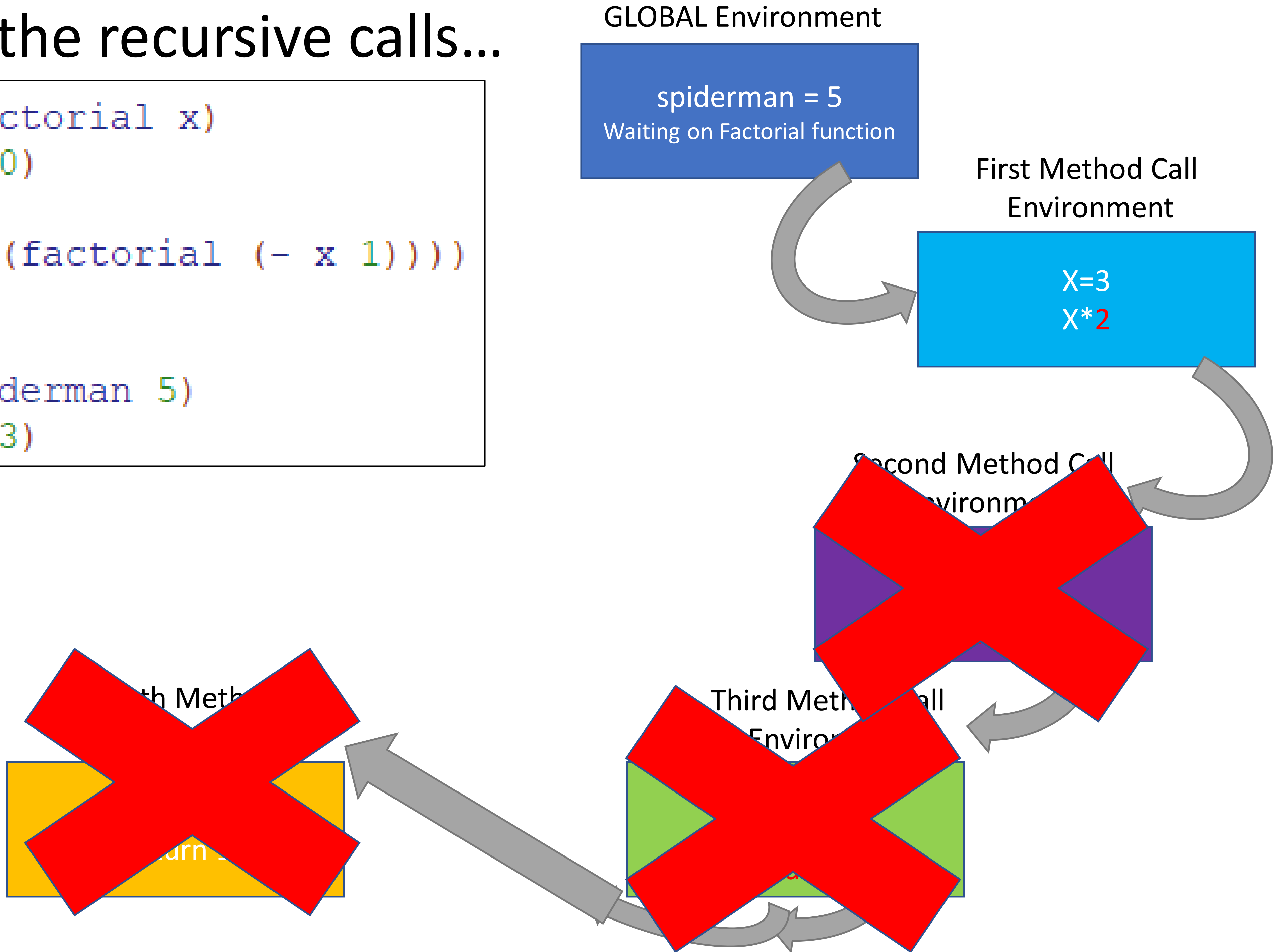
# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



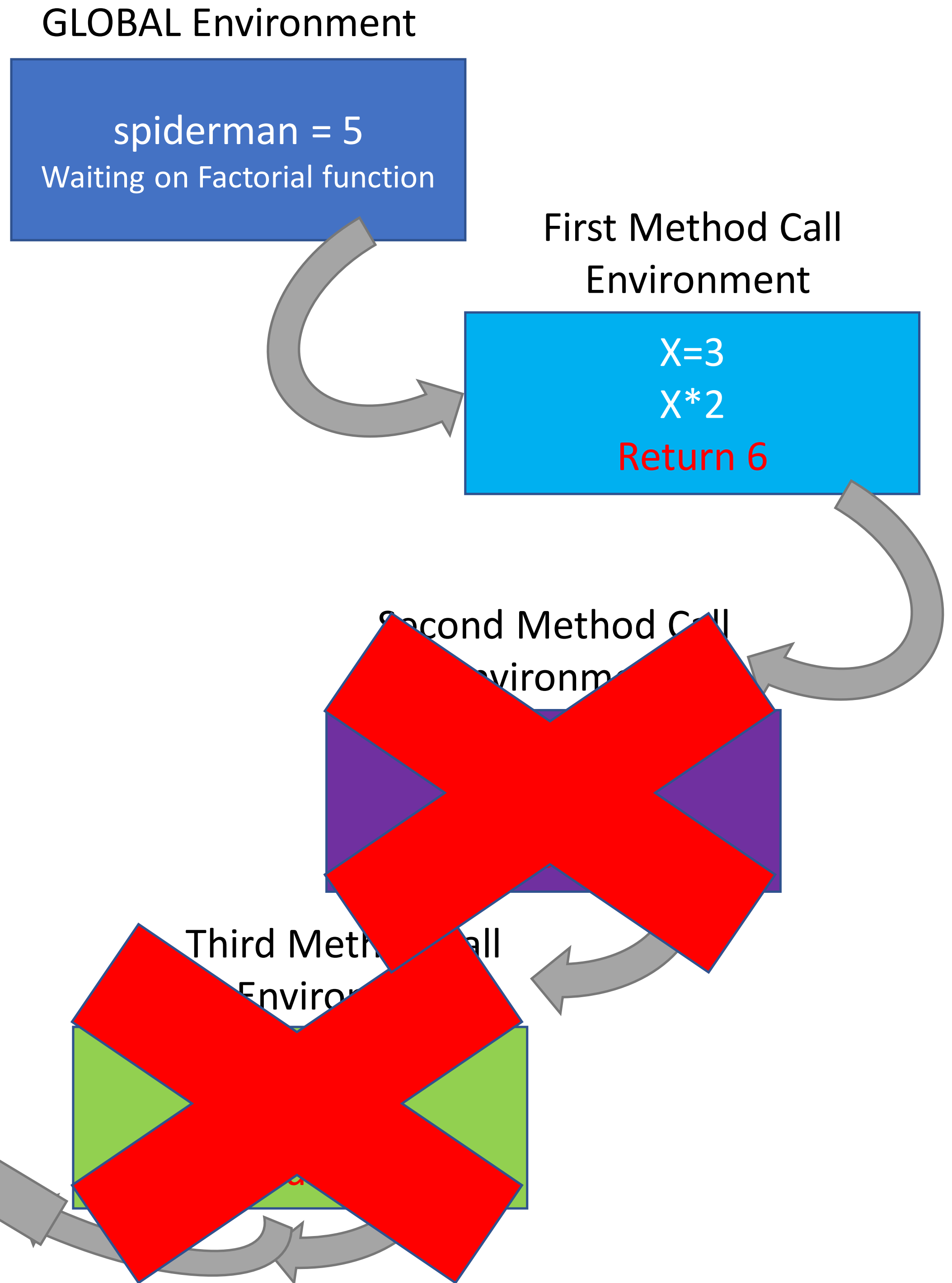
# Understanding the recursive calls…

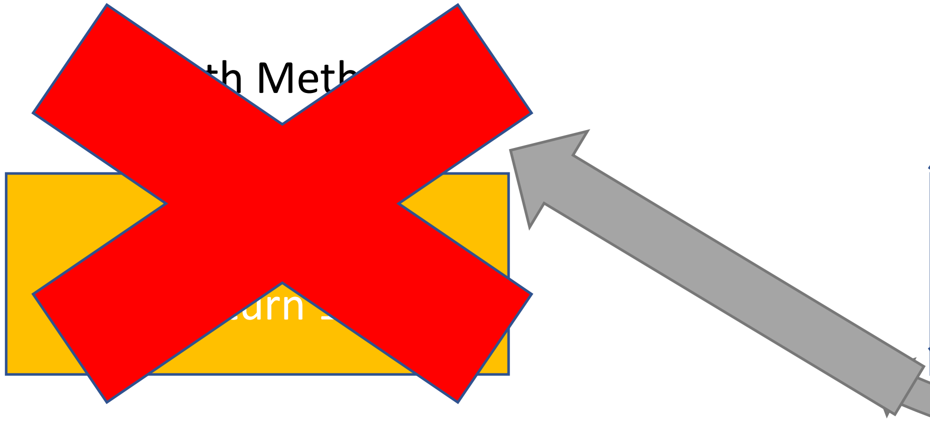
```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```



# Understanding the recursive calls…

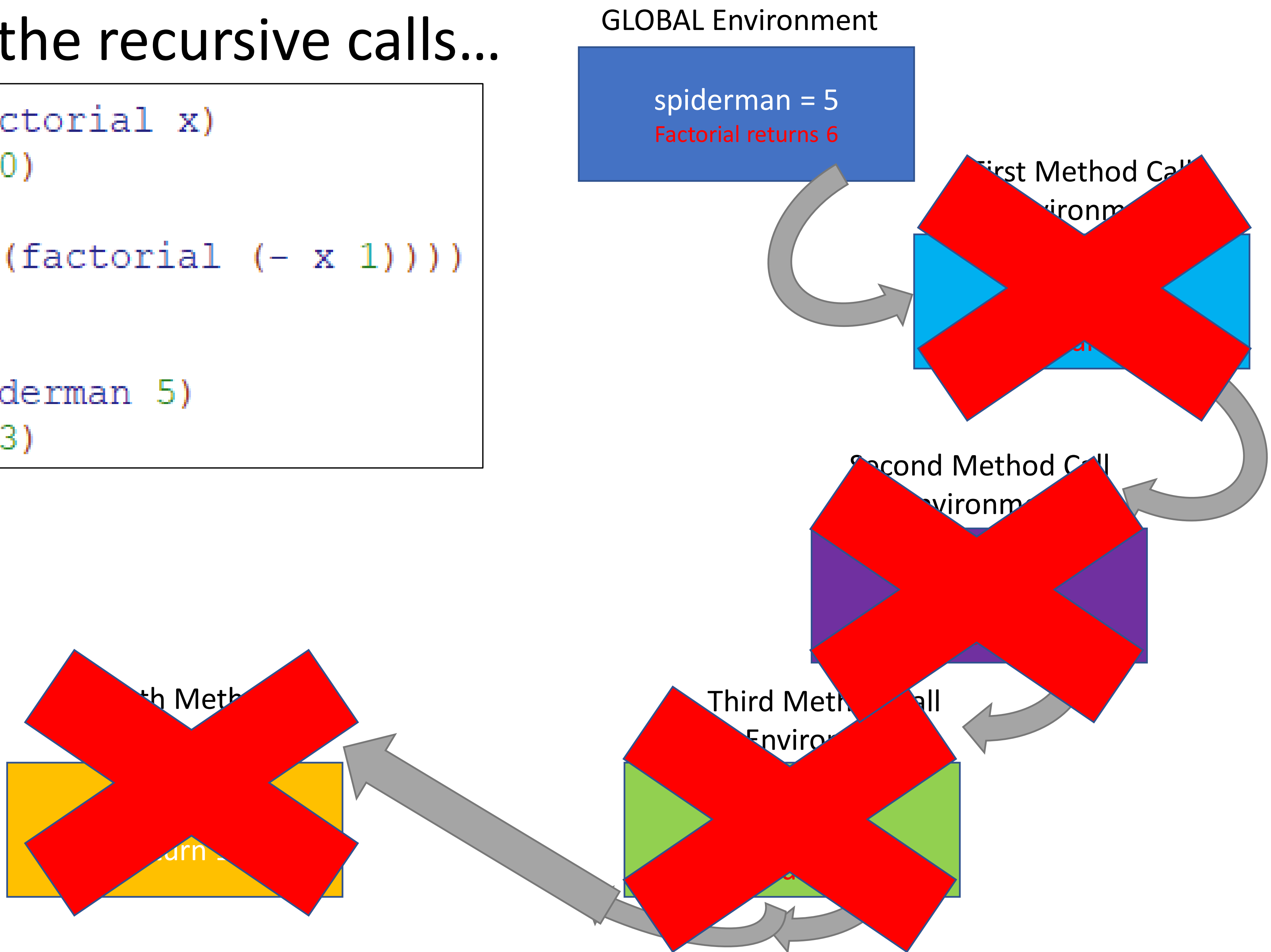
```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```

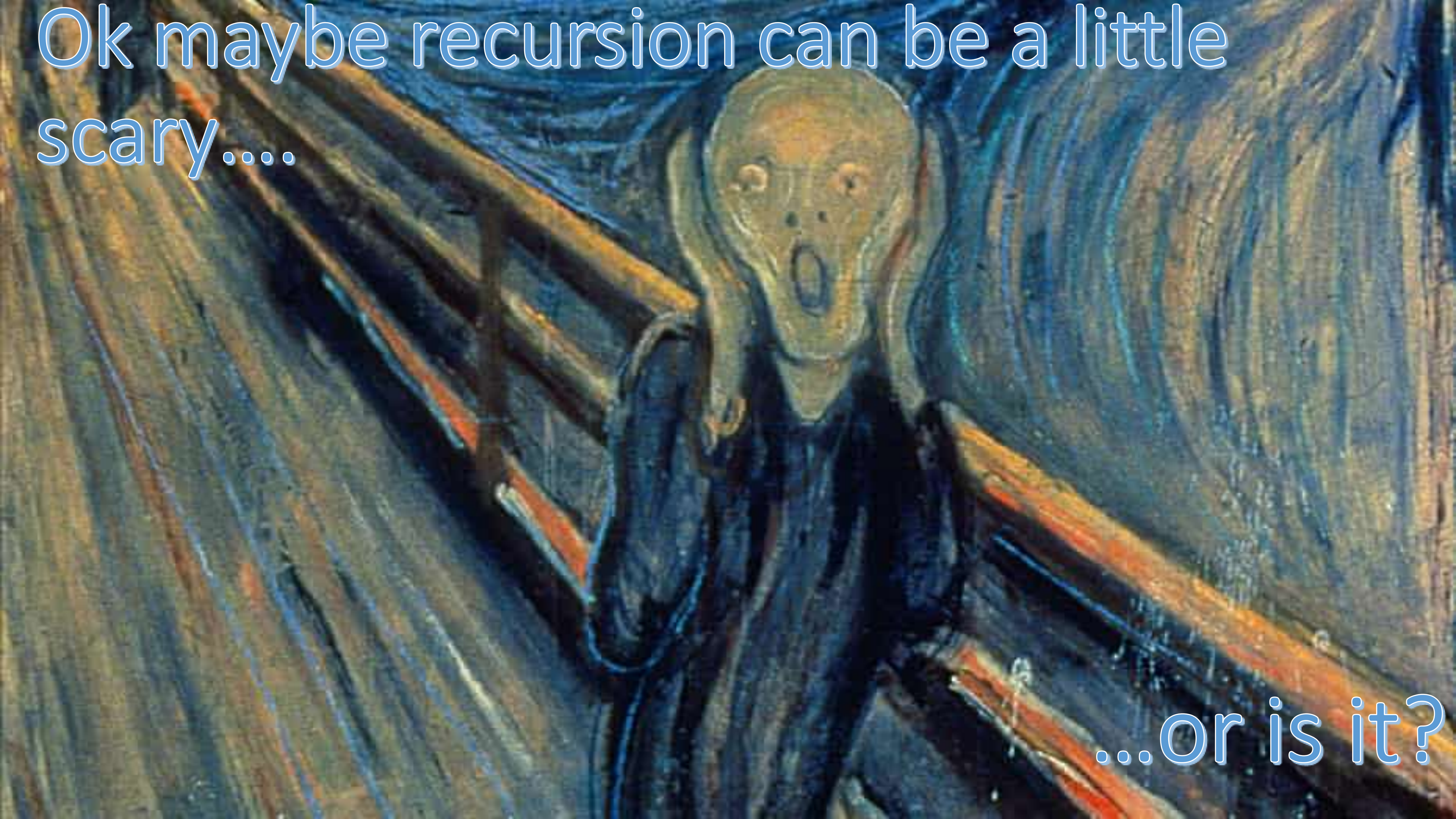


# Understanding the recursive calls…

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )

(define spiderman 5)
(factorial 3)
```

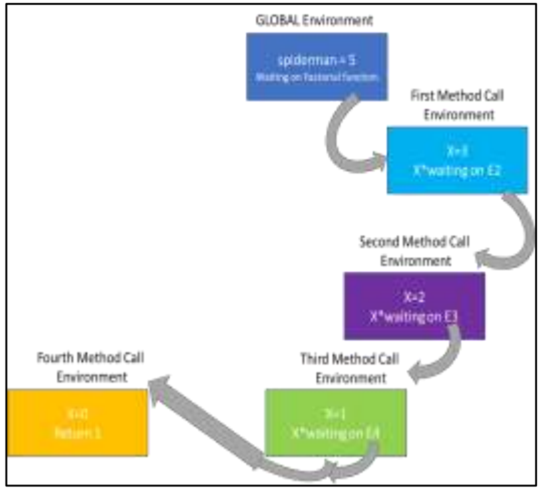



Ok maybe recursion can be a little scary….
…or is it?

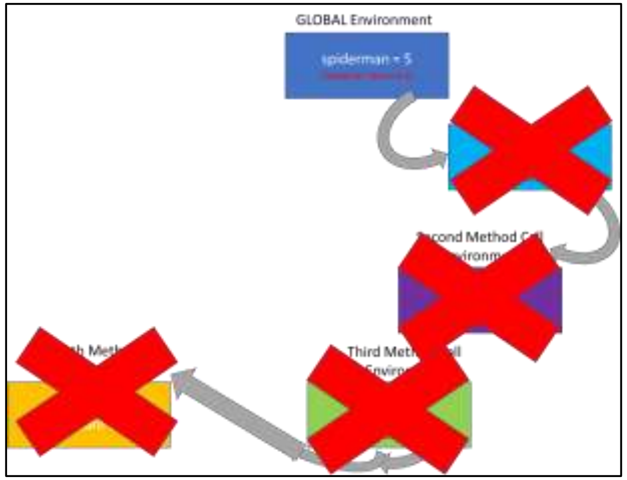
# Recall the Original Question: Are these two codes functionally the same?

```scheme
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1))))
  )
```

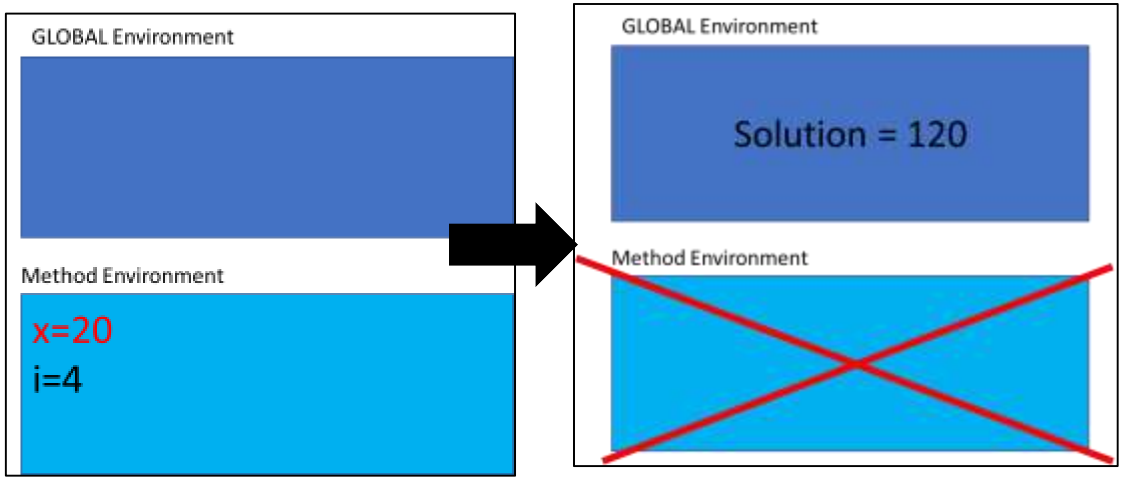
Step 1: Recur until you hit the base case.




Step 2: Return upward until you hit the original function call




```python
#Define the factorial functio
def factorial(n):
    x = 1 #start with inital
    for i in range(n, 1, -1):
        x = x * i #This is sa
    return x
```



# What have we actually discussed so far?

- Two extremely important things.

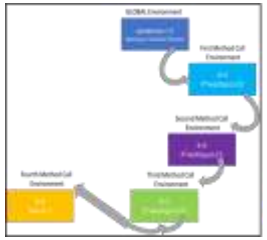
1. How to program functions recursively:

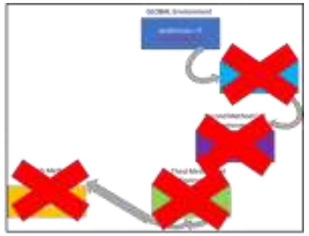
A. Write a method and figure out a base case.

B. Reduce the problem until you reach the base case.

C. Then go from the base case back up.

2. *HOW* to trace through a recursive function call:

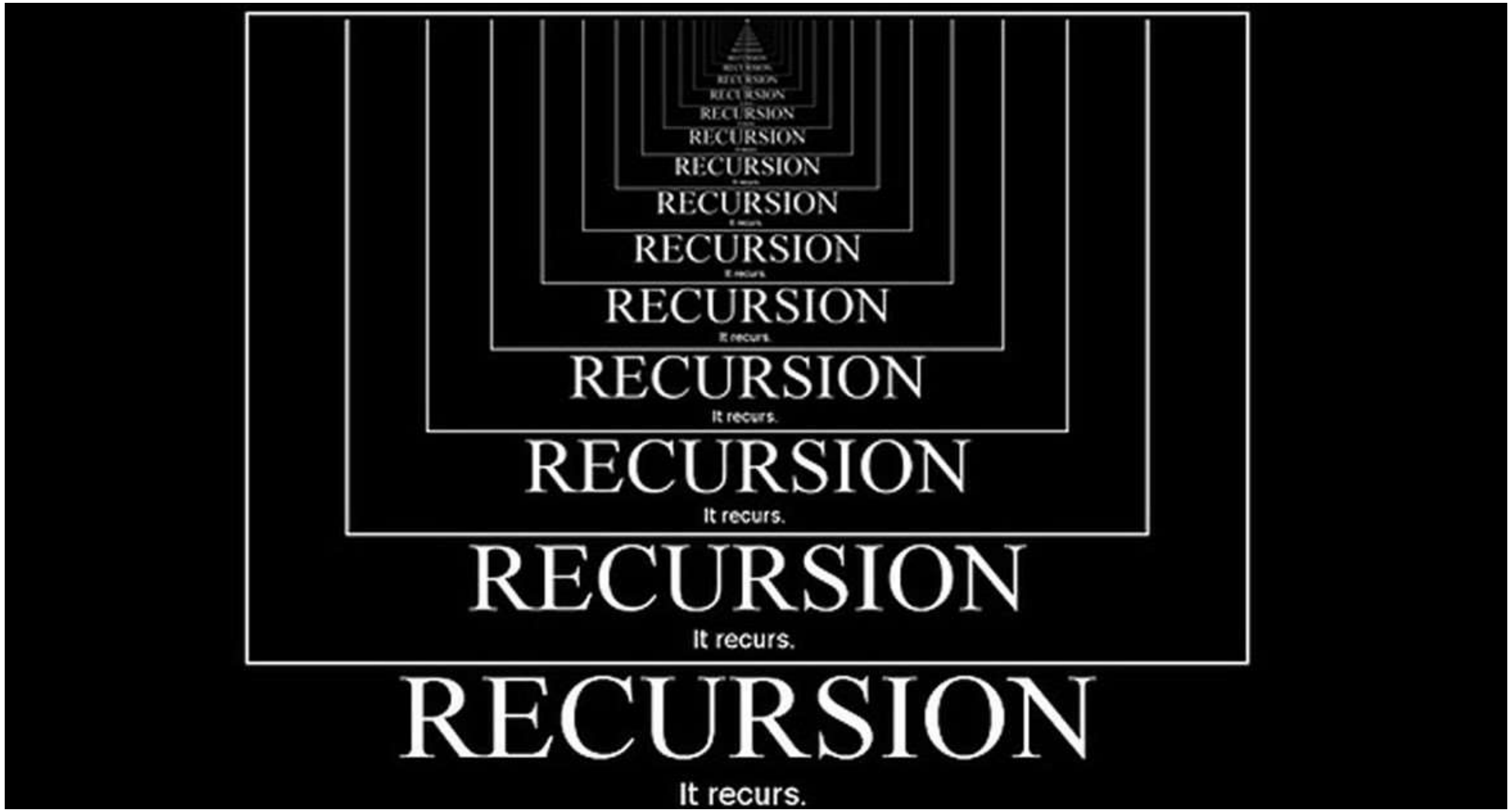




Question: *Why do we need the If statement to NOT evaluate both the true and the false claim?*

Similar Question: *What happens if we don't have a base case?*

Hint: think ∞.



# `if`, IT'S JUST GOT TO BE SPECIAL

- The special evaluation rule for if is critical for this to work.
- Suppose that `(if <pred> <exp1> <exp2>)` evaluated all of its arguments (as per usual evaluation). Then...

```
(factorial 0)
```

expands to

```
(if (= 0 0) 1 (* 0 (factorial (- 0 1))))
```

which would require evaluation of...

`(= 0 0)` and `1` and `(factorial -1)`

**This will never terminate...**

# "SPECIAL" TREATMENT OF OTHER PRIMITIVE FUNCTIONS

- Thus, special "incomplete" evaluation is essential for meaningful recursive programming. For this reason, other primitive functions whose values can be determined by "incomplete" evaluation are also treated as special forms:
- `(and <x_1> <x_2> ... <x_n>)` uses "short-circuited" evaluation. The expressions `<x_1>`, ... are evaluated one at a time, left to right. If any evaluate to `#f`, evaluation stops (and `#f` is returned). Otherwise, `#t` is returned.
- `(or <x_1> <x_2> ... <x_n>)` uses "short-circuited" evaluation. The expressions `<x_1>`, ... are evaluated one at a time, left to right. If any evaluate to `#t`, evaluation stops (and `#t` is returned).

Other small points about variable substitution on the next slides…

# SOME CONCLUSIONS ABOUT SCHEME FROM SUBSTITUTION SEMANTICS

- The name of "local variables" does not matter. Why? They are just placeholders for substitution!
- As far as Scheme is concerned

```
(define (double x) (* x 2))
```

and

```
(define (double y) (* y 2))
```

are identical!

- Why? For any value v,

```
[x/v](* x 2)
```

and

```
[y/v](* y 2)
```

are identical!

# VARIABLE SHADOWS

- Substitution semantics explain what happens when a local variable has the same name as a variable in the enclosing environment.
  Question: How does the following code snippet behave?

```
> (define x 100)
> (define y 200)
> (define (add-to-y x) (+ x y))
> (add-to-y 2)
```

Hint:

- With substitution semantics, variables are given values by two different processes:
  - Looking up in an environment, and
  - Substitution during function application.

# Figure Sources

- https://i.ytimg.com/vi/-2Z0Y3Kk8nU/maxresdefault.jpg
- https://freerangestock.com/sample/32841/russian-dolls-opened.jpg
- https://m.media-amazon.com/images/I/812RM8P59QL._AC_SX425_.jpg
- https://i.imgflip.com/5mhwd2.jpg
- https://memegenerator.net/img/instances/55570377.jpg
- https://racket-lang.org/img/racket-logo.svg
- https://upload.wikimedia.org/wikipedia/commons/thumb/c/c3/Python-logo-notext.svg/800px-Python-logo-notext.svg.png
- https://attachments.f95zone.to/2020/12/965451_ff9a3c5b50a4115b50a23a69dc8cd7b3.jpg
- https://miro.medium.com/max/1170/1*zXpSJCI4hV6FUZkFOnUQJQ.jpeg