# Recursion

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result) // expect 6
}
```

- Write a method named sumToN that takes an Int "n" as an input and returns the sum of number from 1 to n
  - ex: if n == 3, the sum is 1+2+3 == 6

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- Base Case:
  - An input with a trivial output
  - Sum of 0 is 0
  - We could also add 1 -> 1 as a base case

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- Recursive Case:
  - If we have not reached the base case, write code that will compute the sum **with** the use of recursive calls

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- Here's where the magic starts
  - The sumToN method is going to **call itself**!
  - Whenever a method calls itself, it is a recursive method

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- You can make any recursive calls you want as long as they **get closer to the base case**
- Any call that gets closer to 0 will get closer to our base case

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- Keep it simple by only getting *1 step* closer to your base case
- The smallest step closer to our base case is subtracting 1

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- Assume the recursive call will return the correct value
- Write your code based on this assumption
  - If you have faith in recursion, this assumption will be true!

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

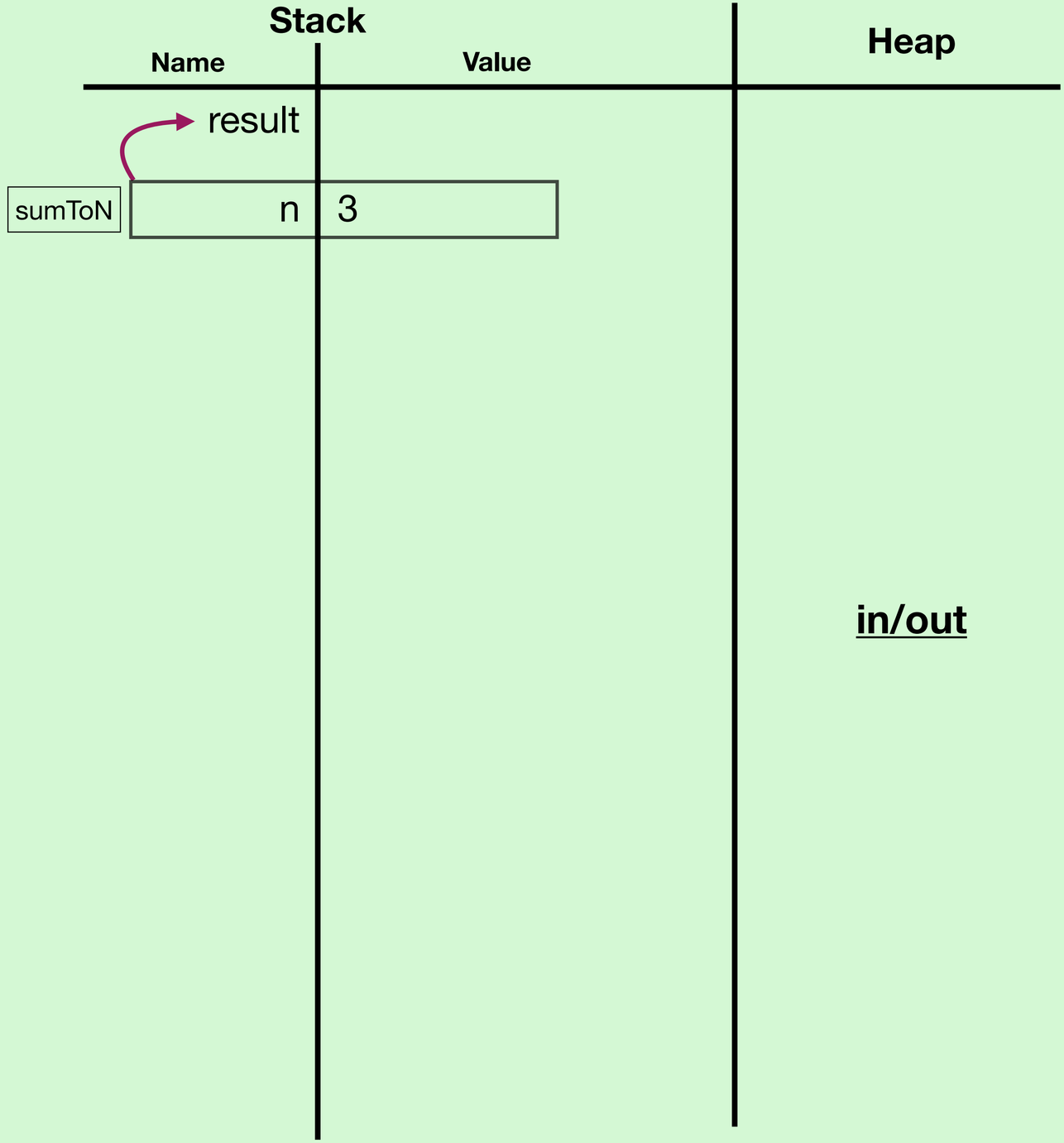
- This assumption will be true
  - If you've reached the base case, it's trivially true
  - If you haven't reached the base case, your logic will compute the correct value

# Recursion By Example

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

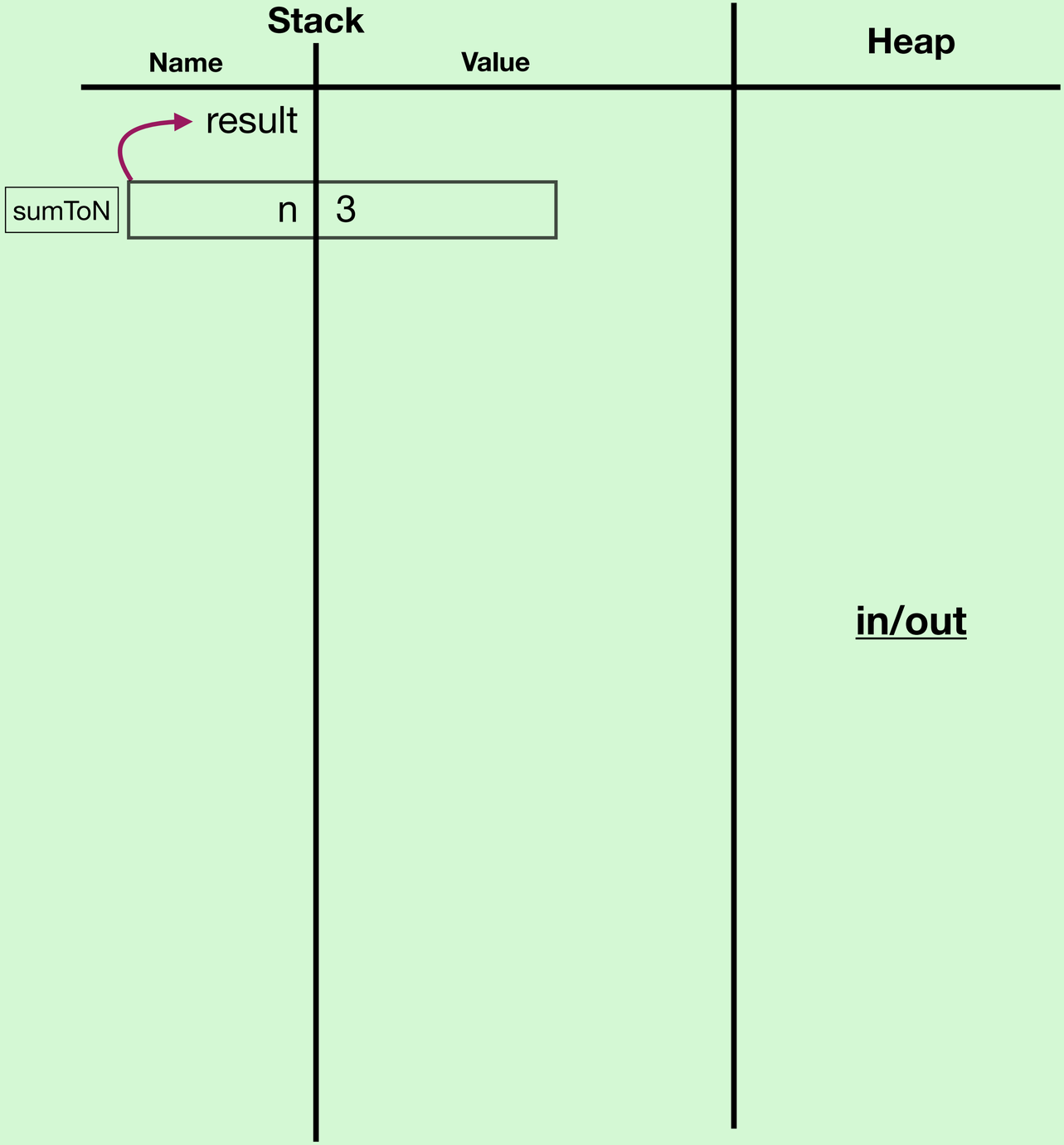
Let's memory diagram this thing!

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```



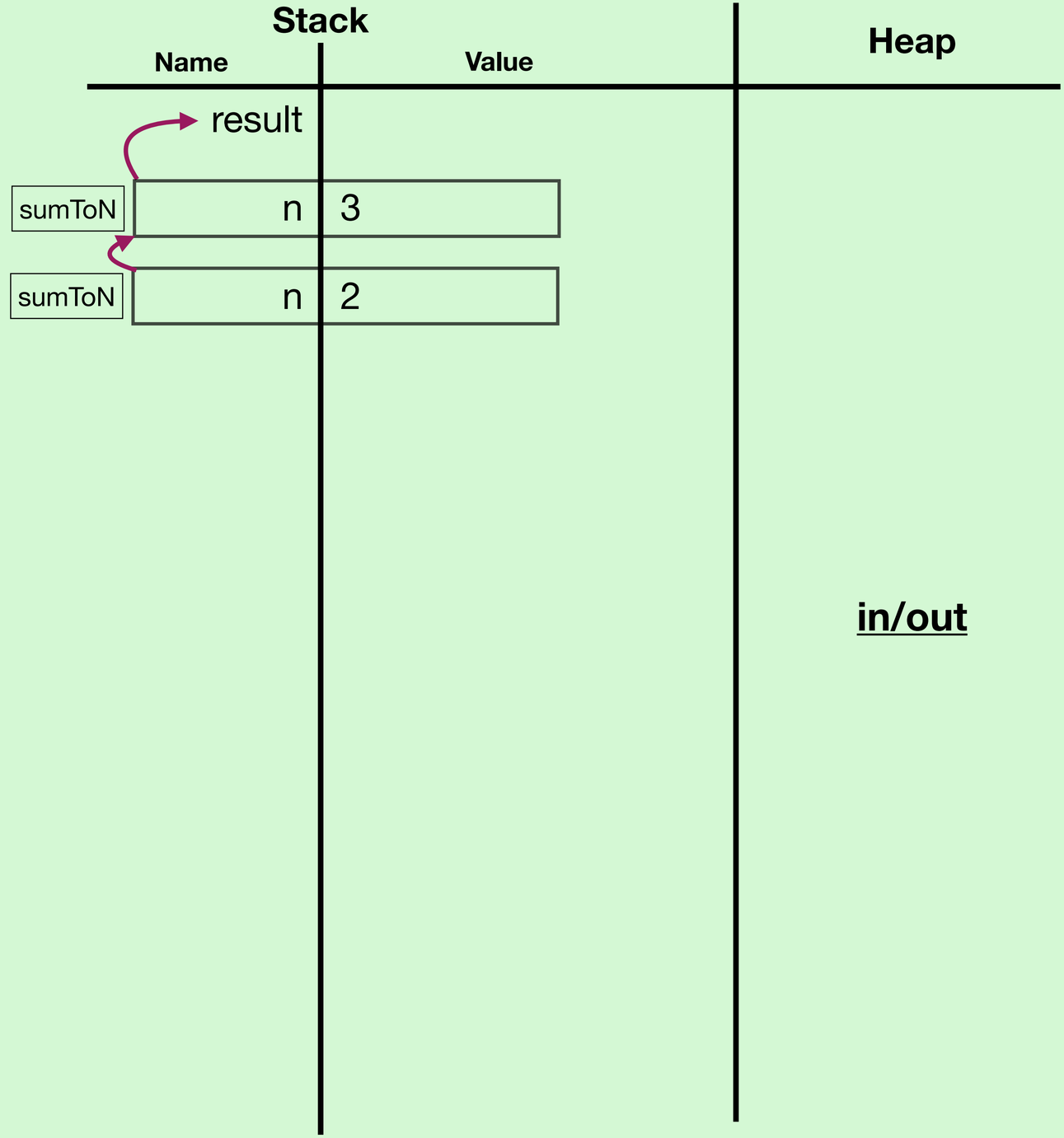

- Start out familiar enough
- Setup the stack
- add a stack frame for the first method call

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```



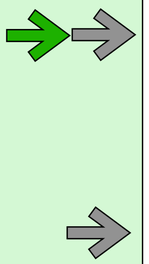

- We have not reached the base case, so we run the recursive step
- We don't draw the dashed box for the if code block since no variables are declared within it

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```



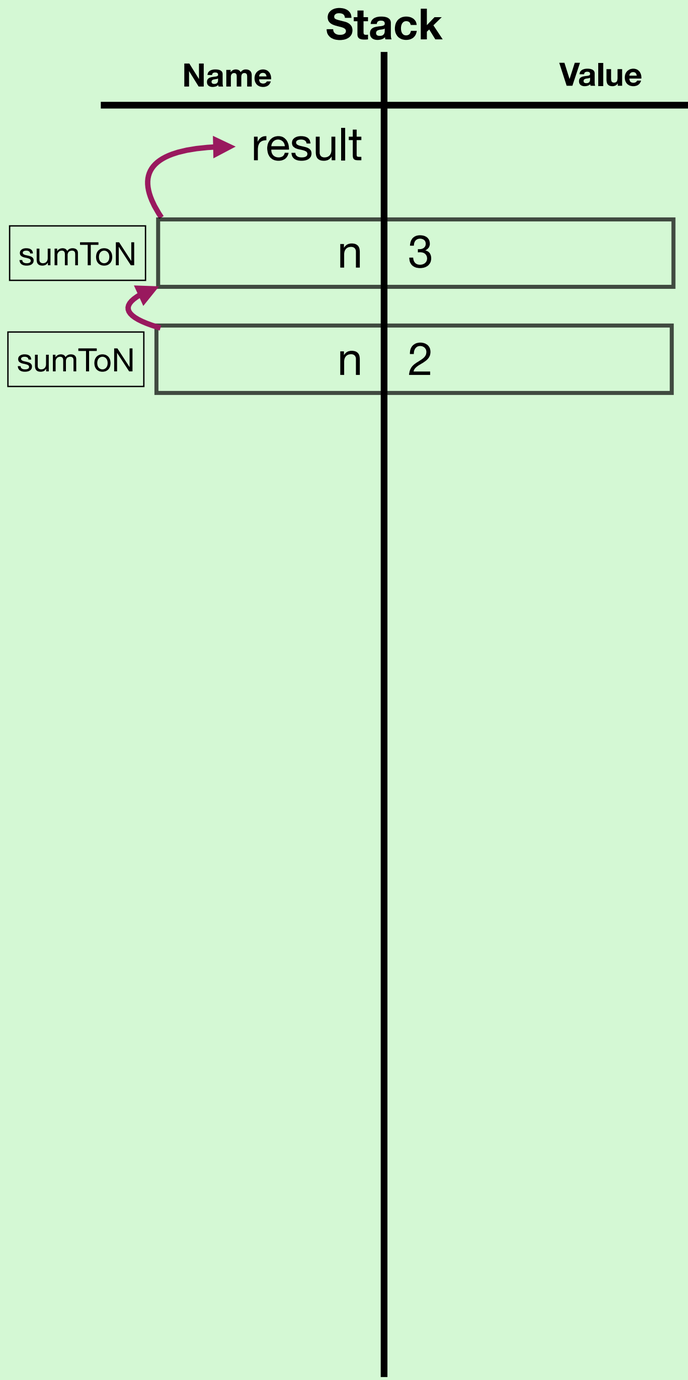

- We reach our first recursive call
- Add a new stack frame just like any other method call
- We draw the return arrow pointing to the stack frame from which it was called

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```





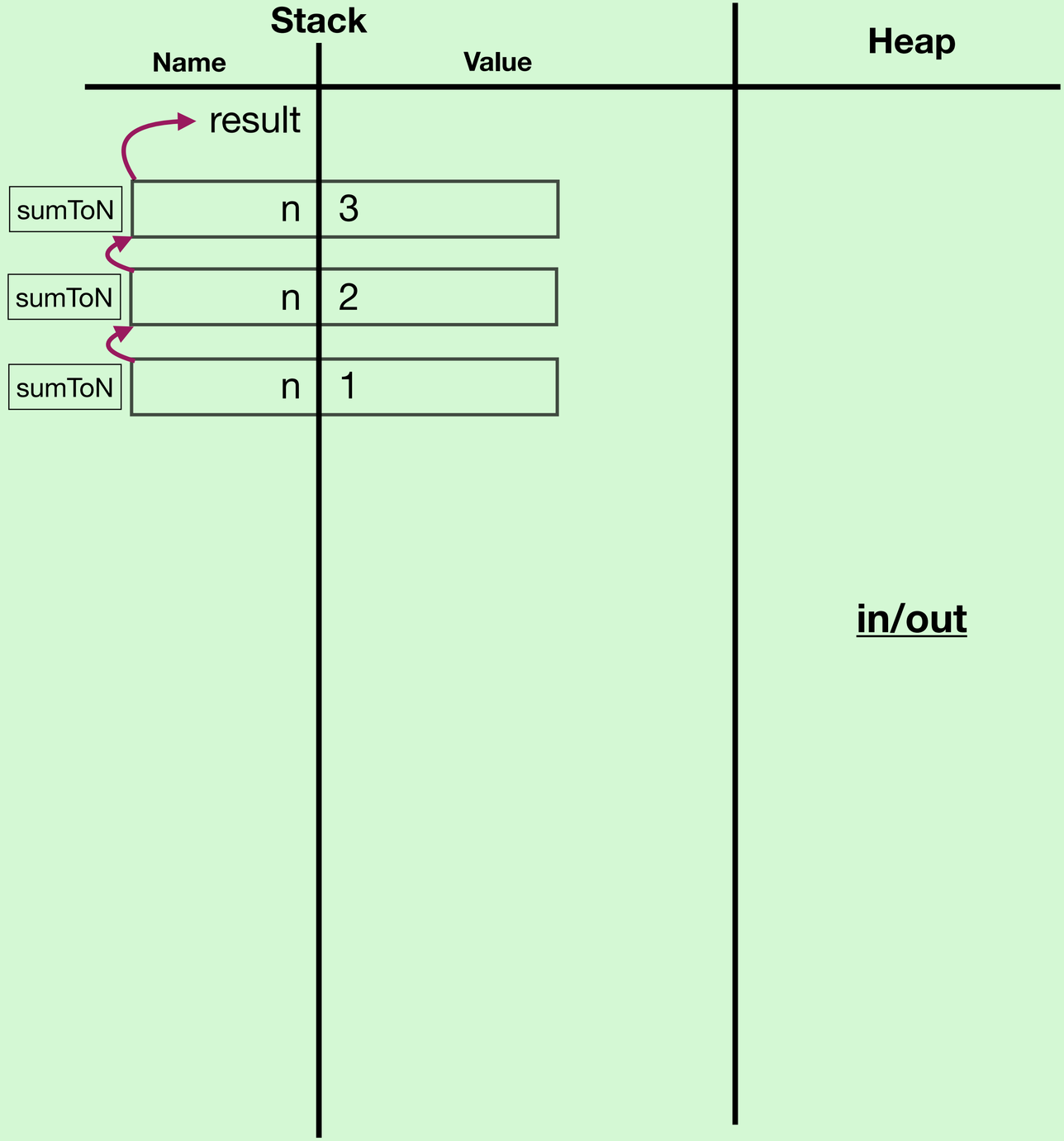





**in/out**

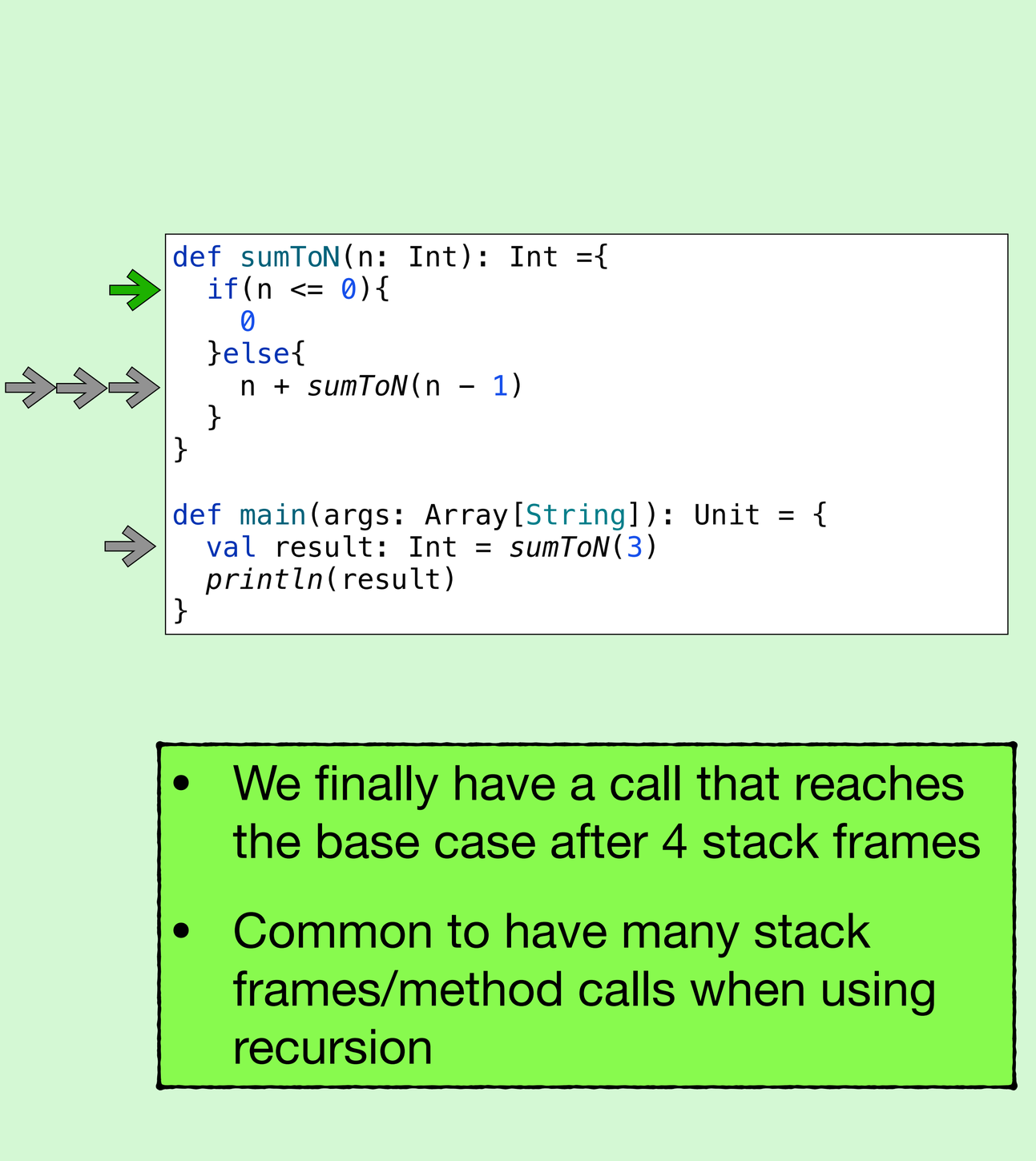
- 2 <= 0 is still false so make another recursive call

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```



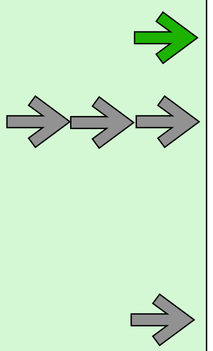

- Keep making recursive calls
- Keep adding stack frames to the stack
- Repeat until we reach the base case

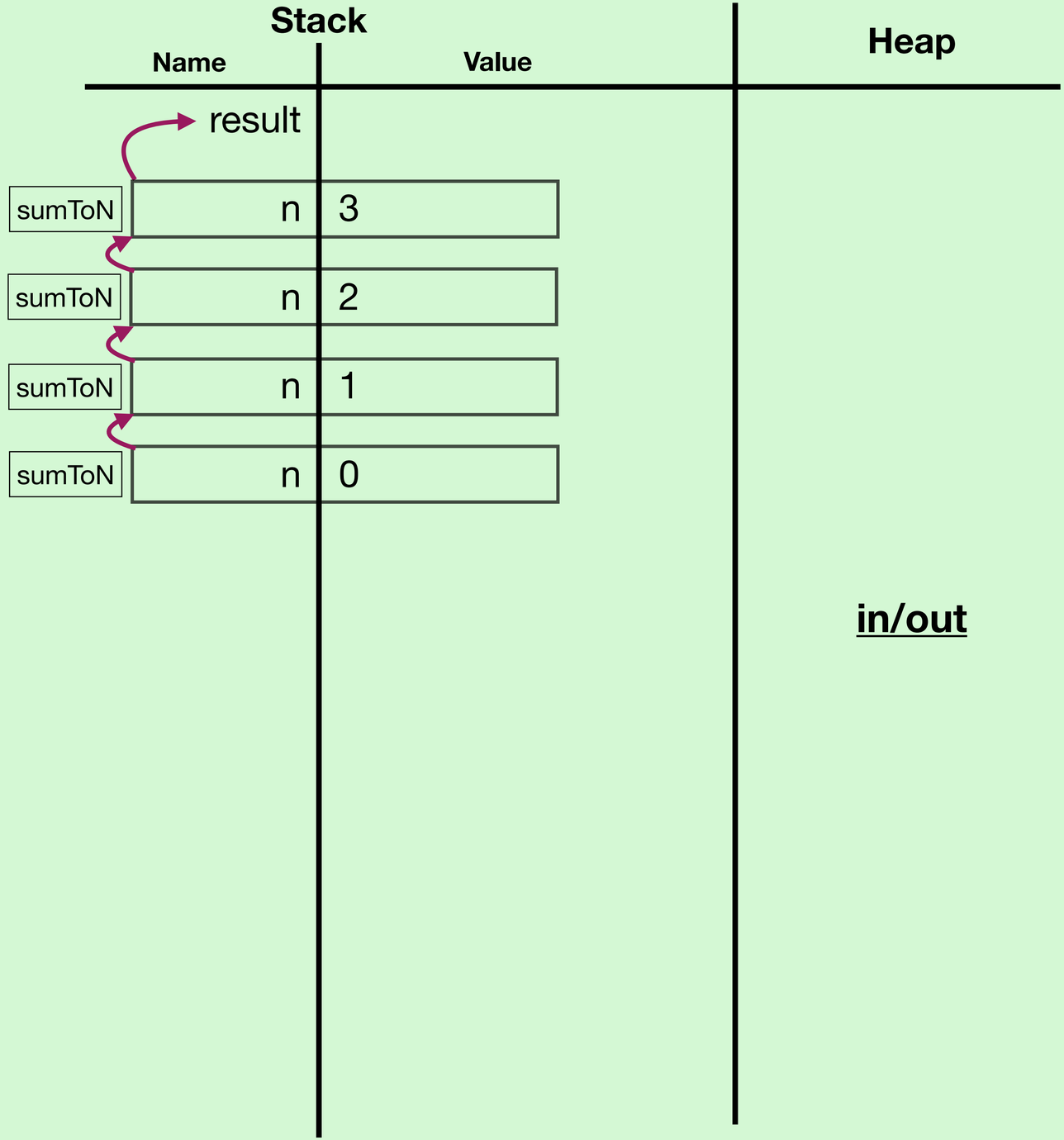
```scala
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




- We finally have a call that reaches the base case after 4 stack frames
- Common to have many stack frames/method calls when using recursion

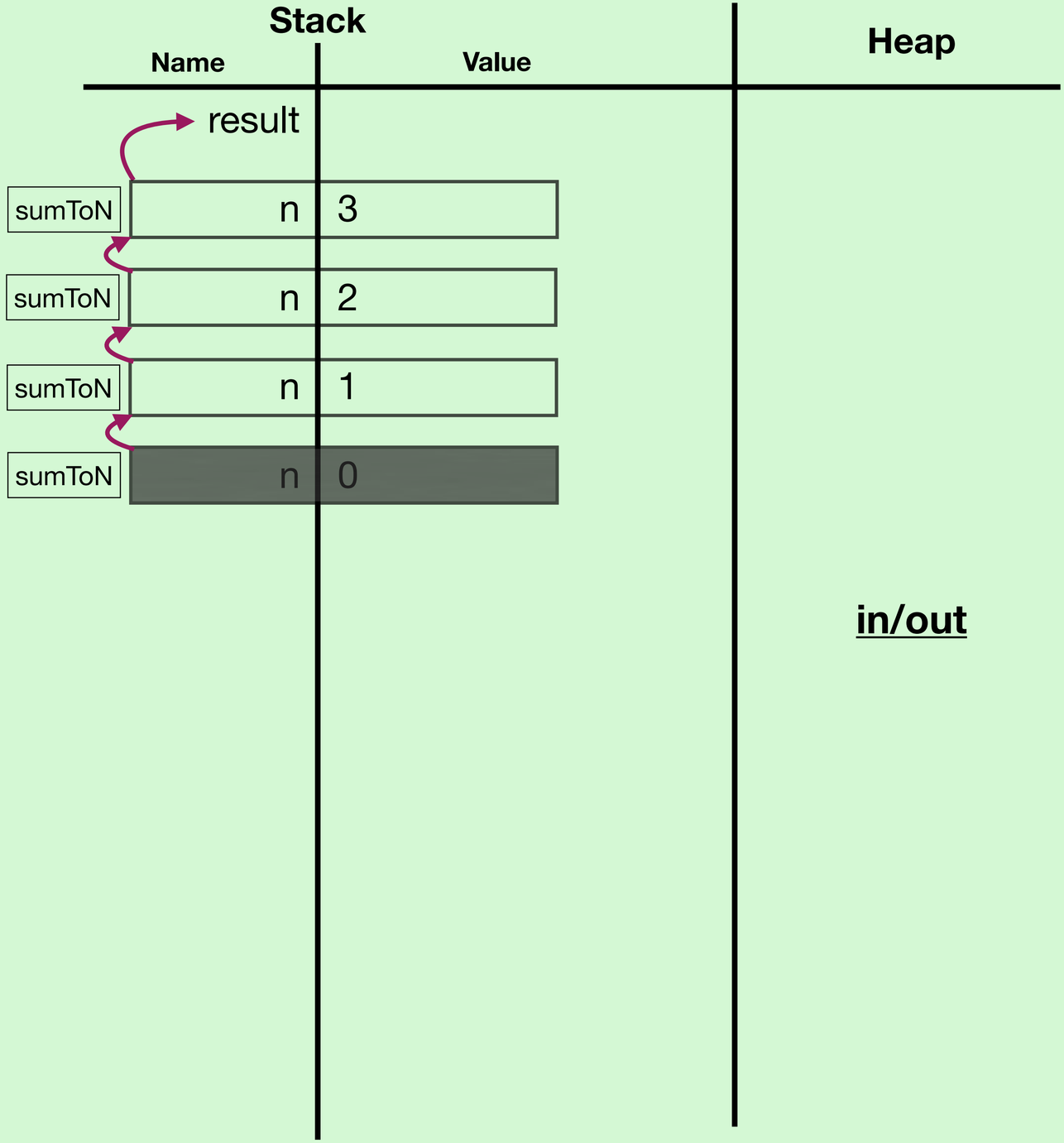
```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```





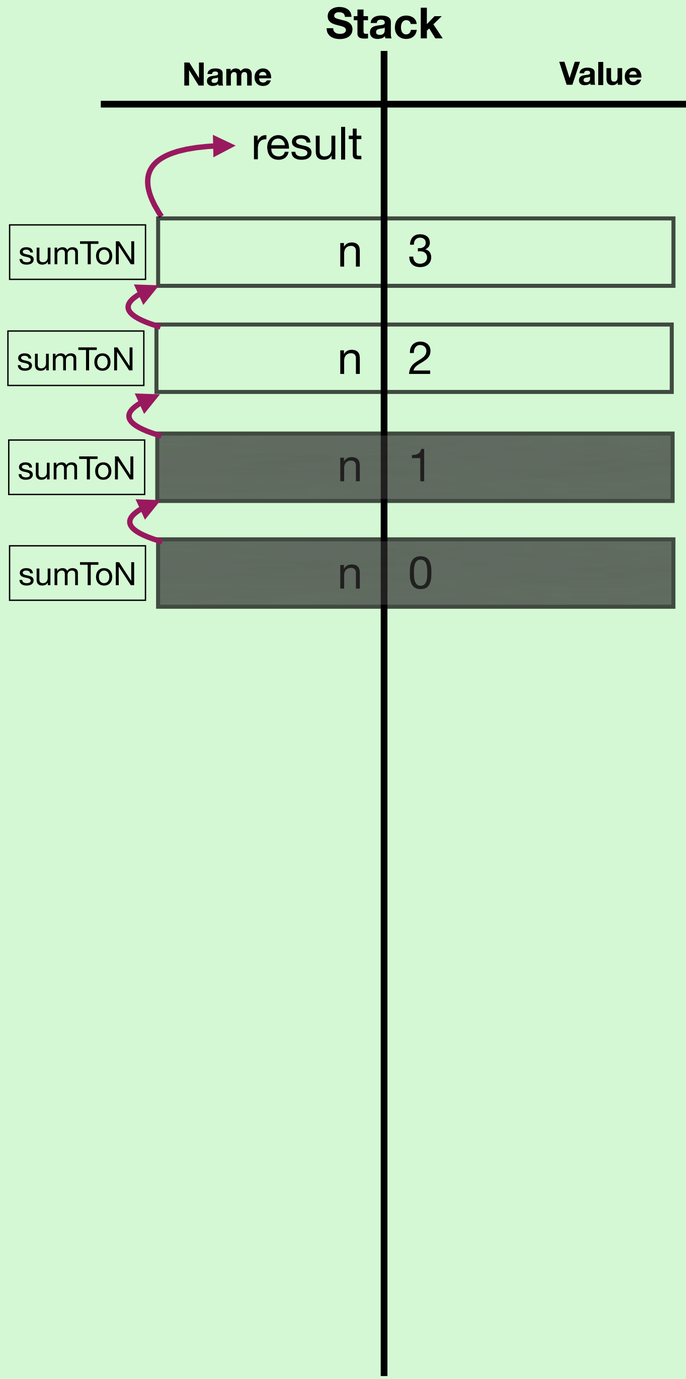

- Base case returns 0

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




- n + sumToN(0)
- n is 1 in this stack frame
- sumToN(0) returned 0 <-- assumption of correctness was accurate!
- This frame returns 1 + 0 == 1

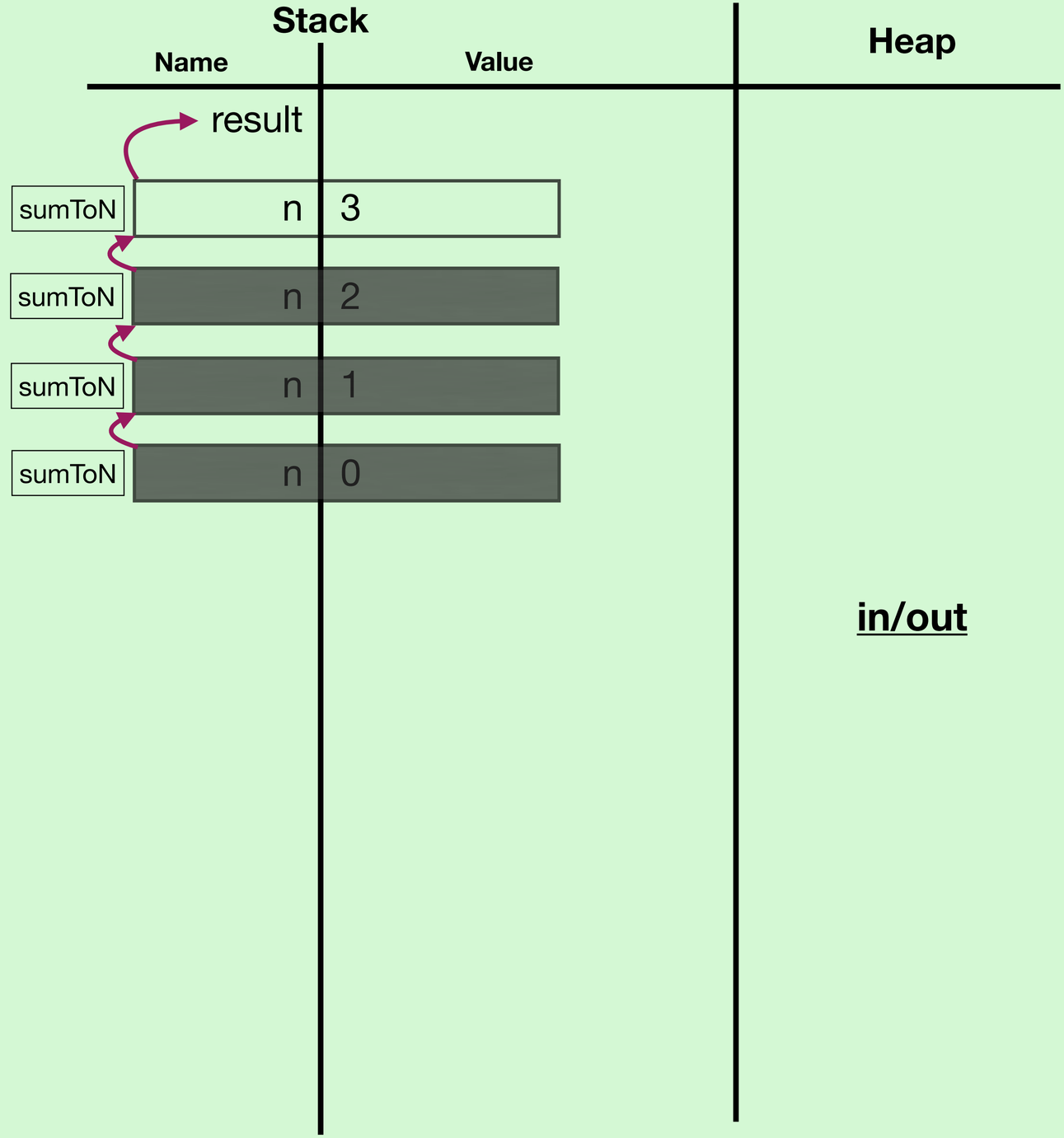
```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




- 2 + sumToN(1)
- sumToN(1) returned 1 <-- assumption still correct!
- This frame returns 2 + 1 == 3

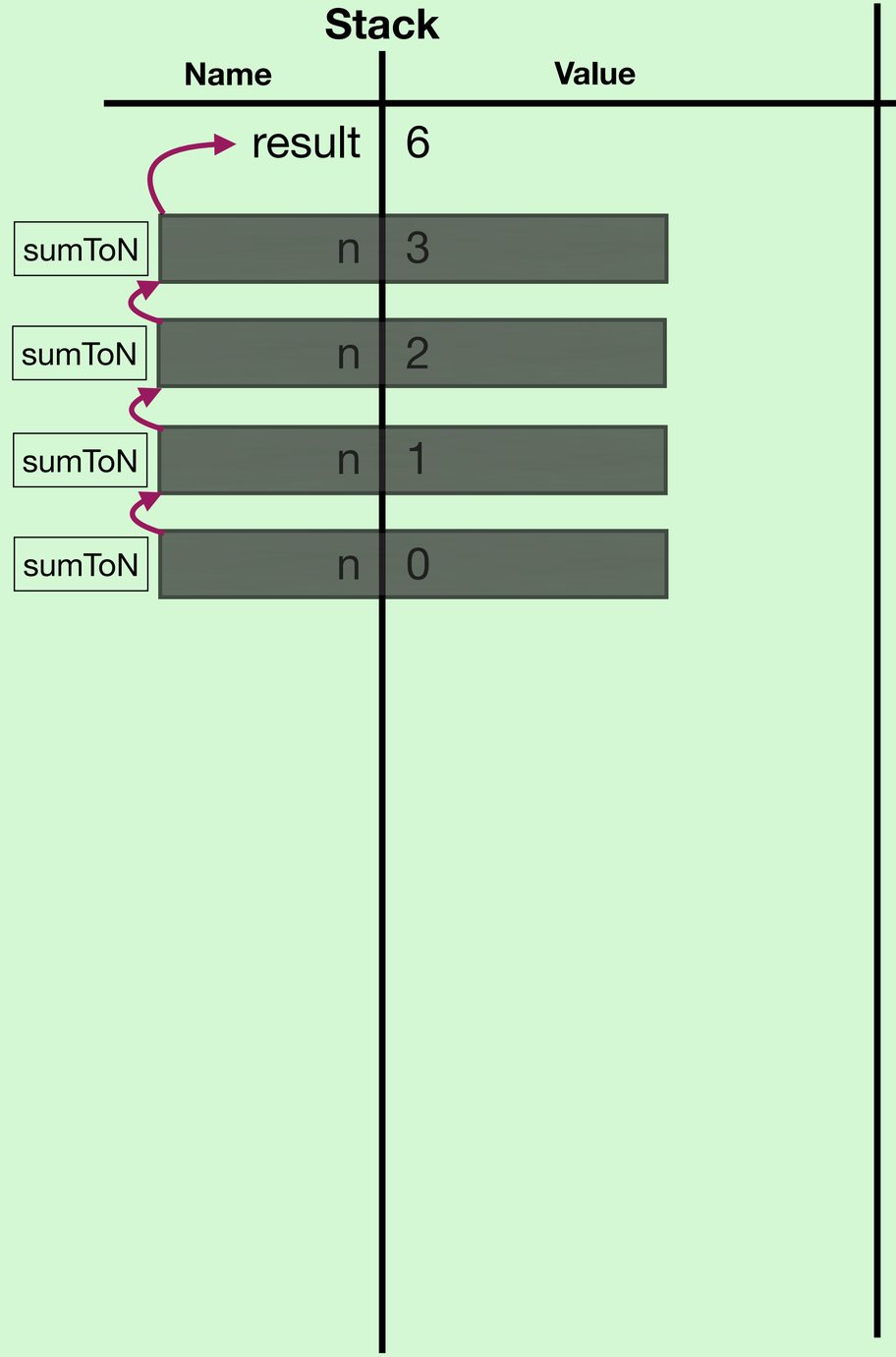
**in/out**

```scala
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




- 3 + sumToN(2)
- sumToN(2) returned 3 <-- assumption still correct!
- This frame returns 3 + 3 == 6

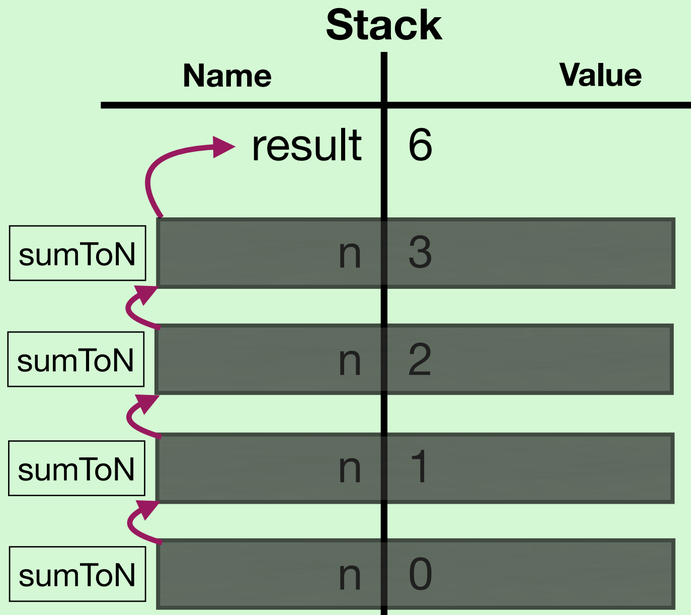
```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




**in/out**

- 6 is returned and stored in result
- We assumed the recursive calls would return the correct values
  - And we were right!

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




Heap

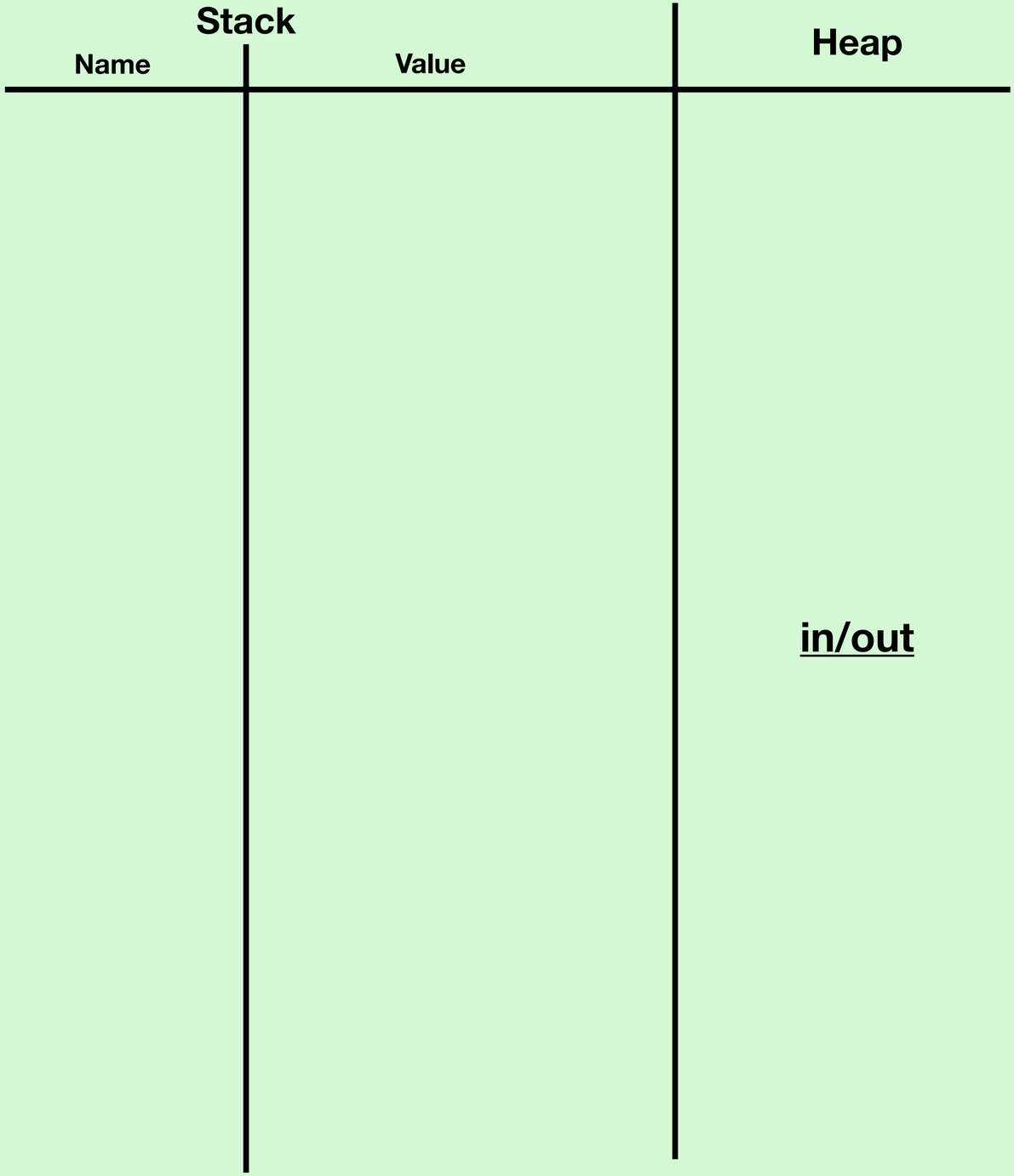
**in/out**

6

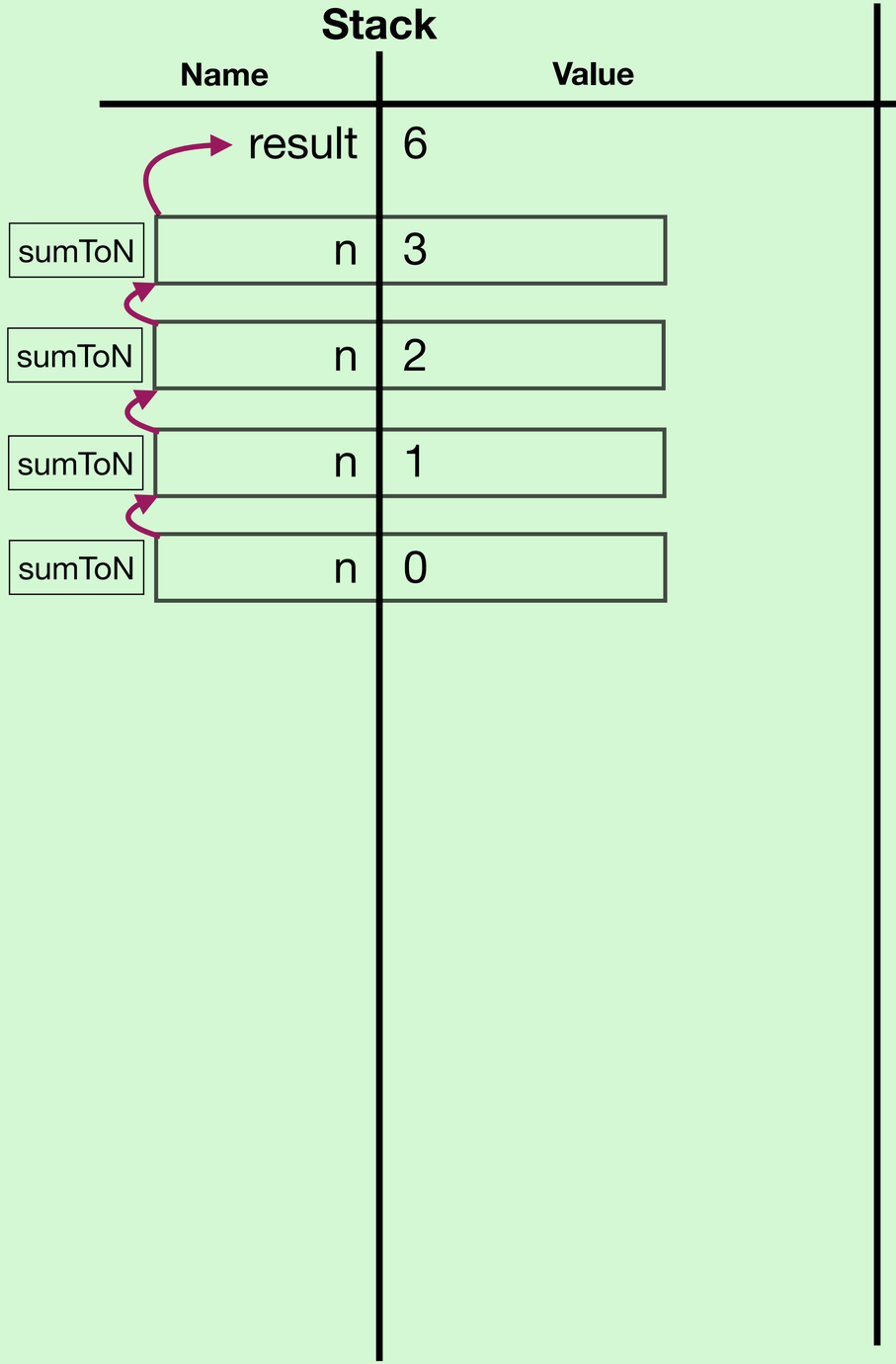
- Print 6 to the screen

```
def sumToN(n: Int): Int ={
  if(n <= 0){
    0
  }else{
    n + sumToN(n - 1)
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- When writing recursive methods
  - Make sure you always reach the base case!!



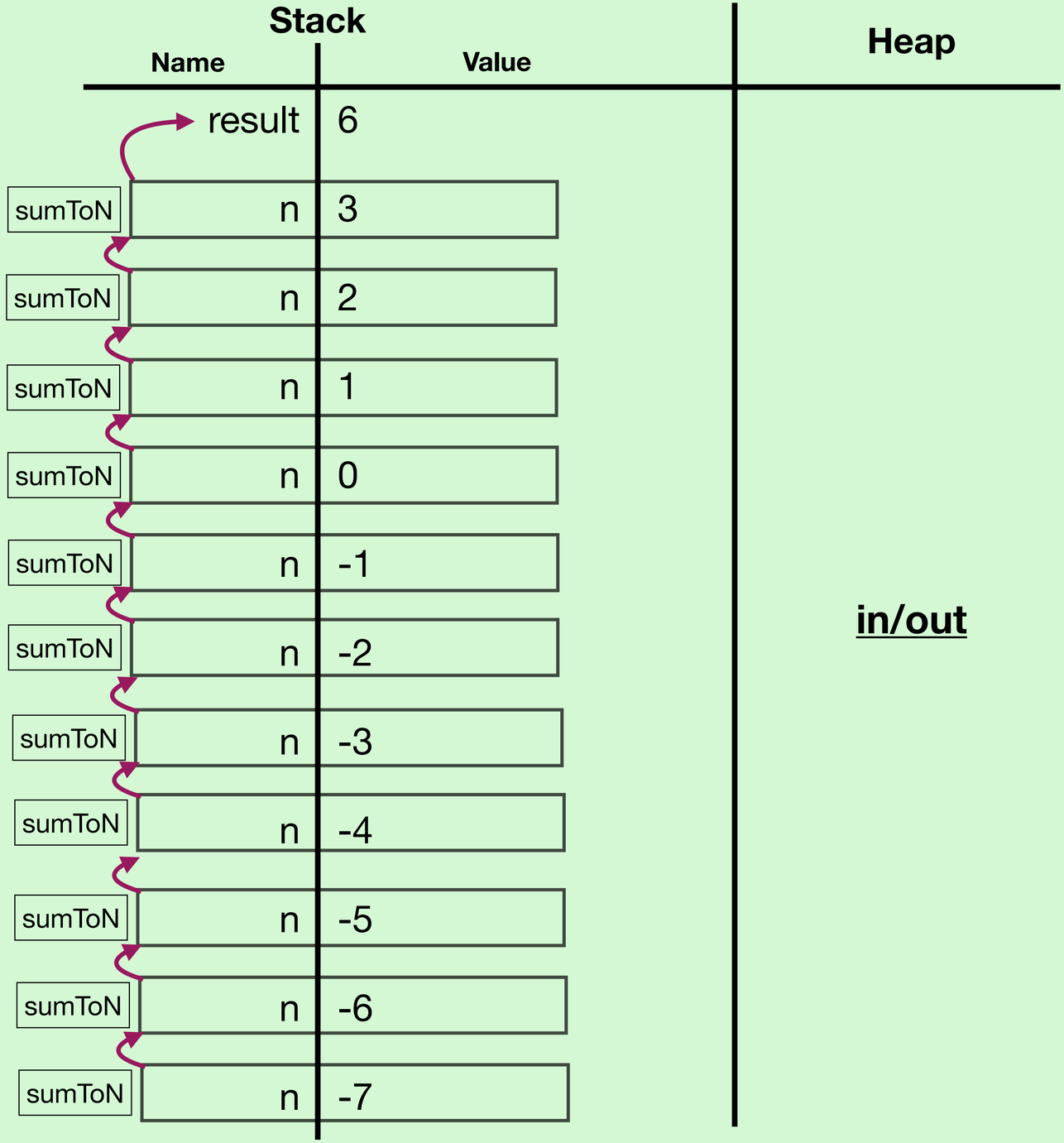
```
def sumToN(n: Int): Int ={
    n + sumToN(n - 1)
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```




- Instead of stopping at n==0
  - This method will continue to add frames on the stack

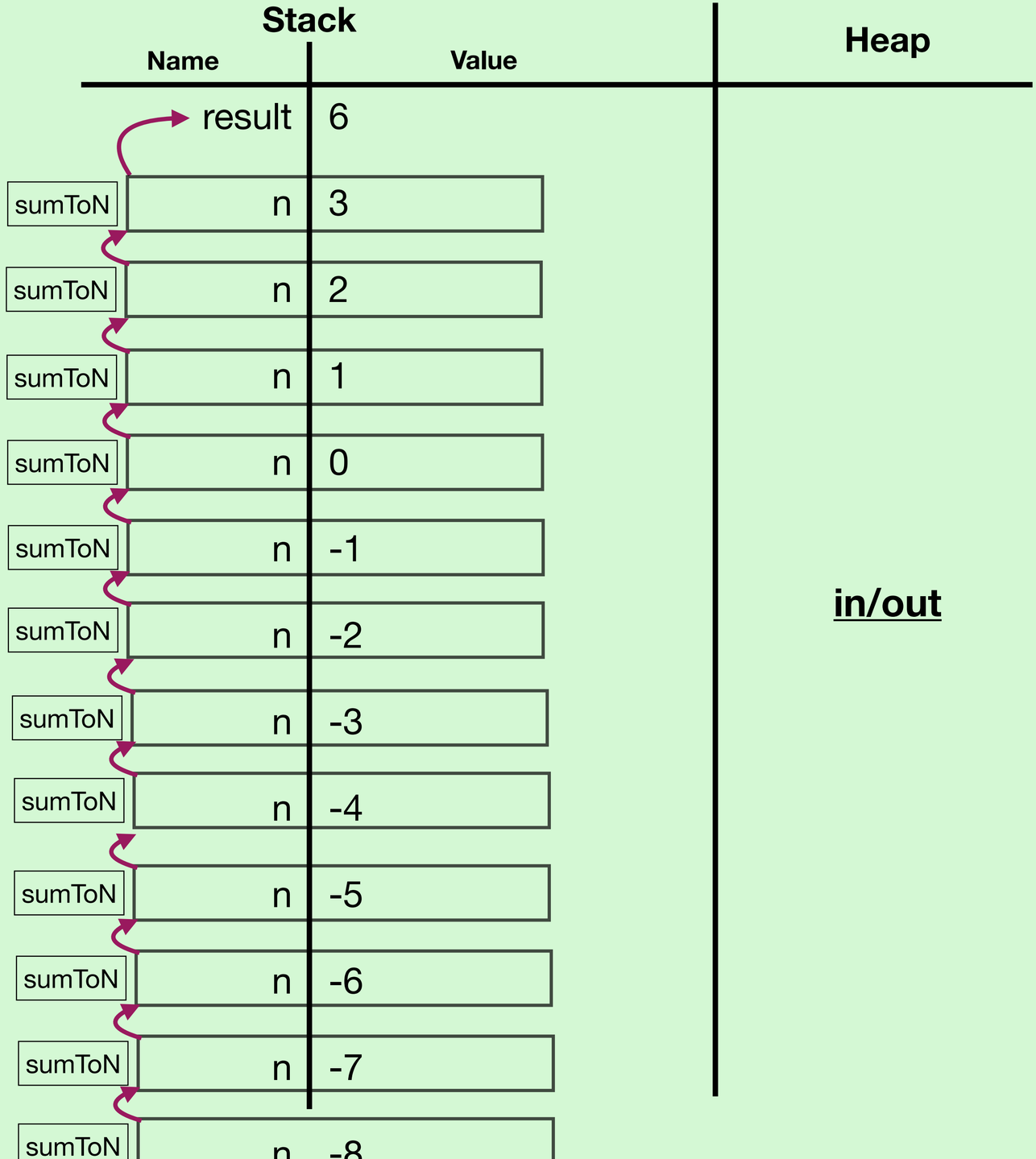
```
def sumToN(n: Int): Int ={
    n + sumToN(n - 1)
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- When will this madness end??



```
def sumToN(n: Int): Int ={
    n + sumToN(n - 1)
}

def main(args: Array[String]): Unit = {
  val result: Int = sumToN(3)
  println(result)
}
```

- Eventually we will run out stack memory
- Result: **Stack Overflow** error
- Program crashes



# Writing Recursive Methods

- **Write a base case(s)** for an input that has a trivial return value
  - A simple input where the return value is trivial
  - Ex. An empty list, an empty String, 0, 1

# Writing Recursive Methods

- Add a conditional to your method to check for the base case(s)
  - If the input is a base case, return the trivial solution
  - Else, run your code that makes the recursive call(s)

# Writing Recursive Methods

- **Assume your recursive calls return the correct values**

  -and-

- **Write your method based on this assumption**

- The primary benefit of writing recursive methods/functions is that we can assume that the recursive calls are correct
- If these calls are not correct, we have work to do elsewhere
  - While writing the top level functionality, assume they are correct and fix the other issues if they are not

# Writing Recursive Methods

- **Ensure your recursive calls always get closer to a base case**
  - Base case is eventually reached and returned
  - Ex. Base case is 0, each recursive call decreases the input
  - Ex. Base case is the empty String and an each recursive call removes a character from the input
- If your recursive calls don't reach a base case
  - Infinite recursion
  - Stack overflow

# Anagrams Example?

Optional. Will only cover in lecture if there is extra time

This is example contains quite a bit of advanced code and you are not expected to understand all of it

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- A method that computes all the anagrams of an input String

- Ex: Input is "cse"
  - output is ("cse", "ces", "sce", "sec", "esc", "ecs")

# Anagrams Example

```scala
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Base Case
  - A String of length 1 is itself its only anagram
  - If the length is 1, return a new list containing only that String

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Base Case Note
  - We will eventually return a list containing all anagrams from the top level call
  - The base case is the only time we create a new List

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Recursive Step
  - For each character in the input String
    - Remove that character and make a recursive call with the remaining characters
    - Append the removed character to all the returned anagrams

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Recursive Step
  - We write this code with the assumption that our recursive calls will return all the anagrams of the new Strings
  - If our logic is sound, this assumption will be true through the power of recursion!

# Anagrams Example

```scala
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Always reach a base case
  - We always make recursive calls on the input String with 1 character removed
    - newString.length == input.length -1
  - This always gets us closer to the base case

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Always reach a base case
  - When the base case is reached and returned, our logic starts working for us
  - If this code does append the removed character to each returned anagram, output is generated starting at the base case and built up as the stack frames return

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Example:
  - input == "at"
  - Makes 2 recursive calls to the base case
    - ["a"] and ["t"] are returned
  - Append "t" to "a" and "a" to "t" (The removed characters)
  - Return ["at", "ta"] to the next recursive call with an input of length 3

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- yield: Creates a data structure by collecting the value of the last expression at each iteration of the loop
- map: Calls a function on every element of a data structure (appending the $i^{th}$ character of input to each String in the List)

# Anagrams Example

```
def anagrams(input: String): List[String] = {
  if (input.length == 1) {
    List(input)
  } else {
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)
      anagrams(newString).map(_ + input.charAt(i))
    }).toList

    output.flatten.distinct
  }
}
```

- Flatten: Creates a single List from a List of Lists containing all the elements from each List
- Distinct: Creates a new List with all duplicate values removed