Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer
Science II

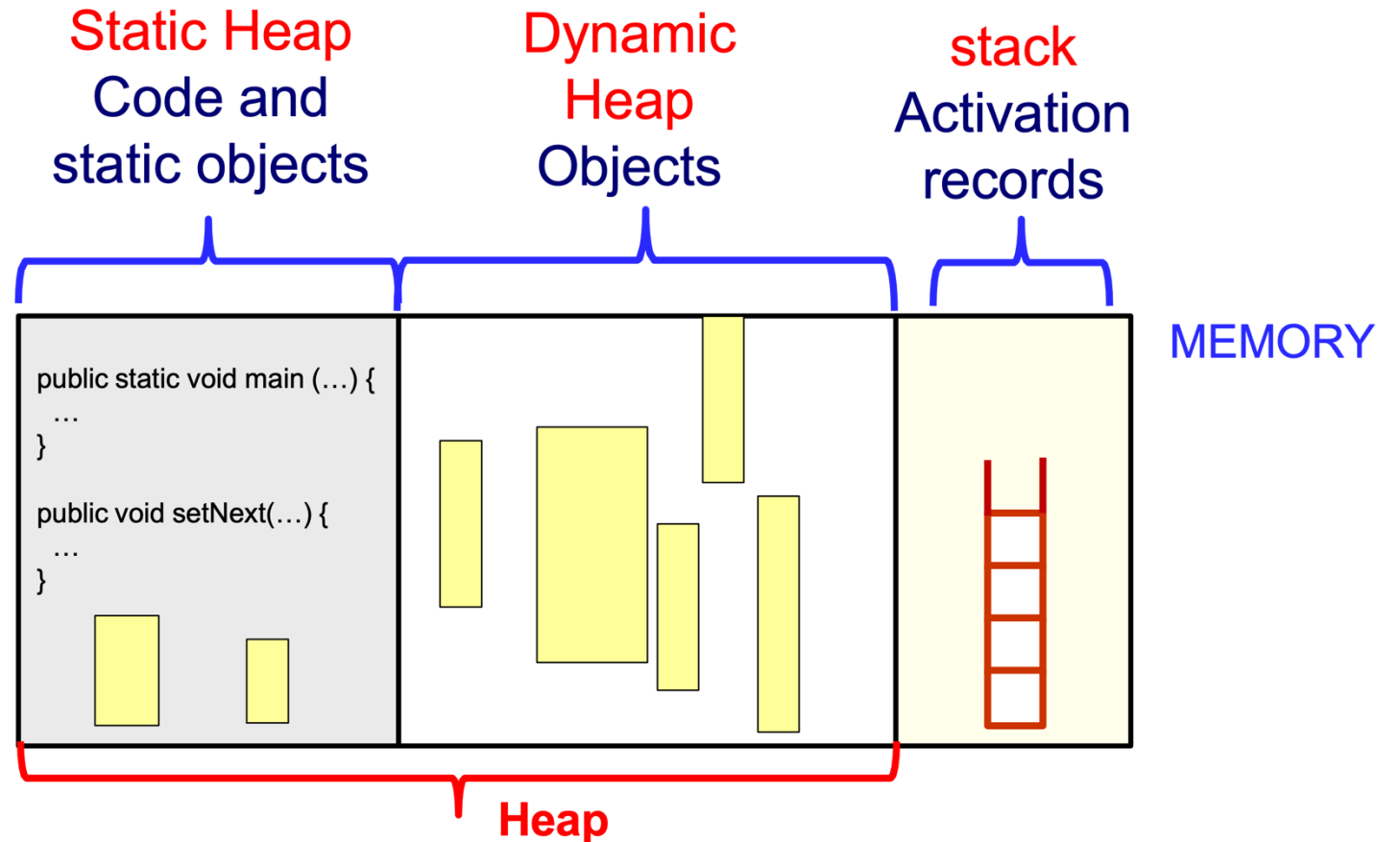# Memory Allocation in Java (cont.)

Ahmed Ibrahim

Memory Allocation in Java

- When a program is being executed, separate areas of memory are allocated:

Static Heap
Code and static objects

Dynamic Heap Objects

stack
Activation records

MEMORY

```
public static void main (…) {
…
}

public void setNext(…) {
…
}
```

Heap

Recall: Execution Stack

- The execution stack (also called runtime stack or call stack) is used to store information needed while a method is being executed, like

  - Local variables

  - Formal parameters

  - Return value

  - Return address

Method information

| Local variables |
| Parameters |
| Return value |
| Return address |
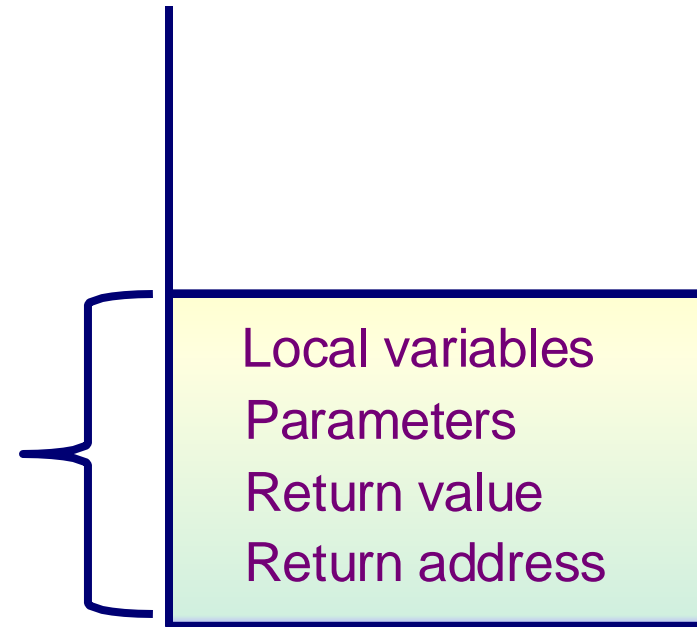
Execution Stack

Recall: Execution of the Program

- When the **main** method is invoked:
  - An activation record for the main is created and pushed onto the execution stack
- When the **main** calls the method **m2**:
  - An activation record for m2 is created and pushed onto the execution stack
- When **m2** calls **m3**:
  - An activation record for m3 is created and pushed onto the execution stack
- When **m3** terminates, its activation record is popped off and control returns to **m2**

Execution of the Program

- When **m2** next calls **m4**:
  - What happens next?
  - What happens when **m4** terminates?
- What happens when **m2** terminates?
- What happens when the **main** function terminates?
  - Its activation record is popped off, and control returns to the operating system

Activation Records

- Example of what is in the activation record for a method with primitive type variables, and non-primitive variables.

```
public static int m2 (int param2)
{int local2 = 1;
 Integer i = new Integer(3);
 m3(4);
 return local2 + param2 + m4(3);}
public static void m3 (int param3)
{int[] arr = new int[param3];}
public static int m4 (int param4) {return param4 * 2;}
public static void main (String[] args) {int local1 = m2(5);}
```

To execute this program, it is first compiled and translated to Java bytecode and stored in the static heap.

```
01001100011101001011010111
01001010110101010001111011
10101001010101010011100101
11101011101011101001110000
11010101010000100011100011
10111010011000111011000001
11010100100010101000101011
```

Static heap

Activation Records

- Example of what is in the activation record for a method with primitive type variables, and non-primitive variables.

```
public static int m2 (int param2)
{int local2 = 1;
 Integer i = new Integer(3);
 m3(4);
 return local2 + param2 + m4(3);}
 public static void m3 (int param3)
 {int[] arr = new int[param3];}
 public static int m4 (int param4) {return param4 * 2;}
 public static void main (String[] args) {
 int local1 = m2(5);}
```

addr2
addr3

addr1

Each instruction is assigned an address. We indicated the addresses of method invocations in the program.
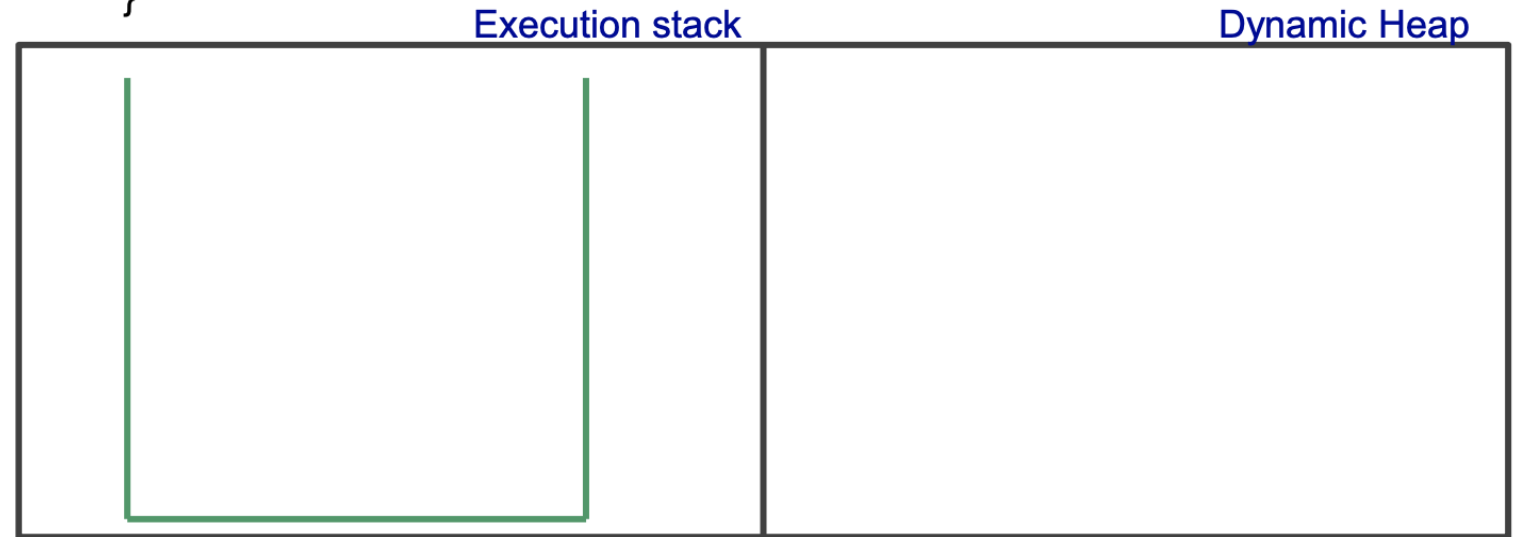
0100110001110100101101011
01001010101101010100011110111
101010010101010100011100101
1110101110101110100110000
1101010101000001**00011100011**
101110100110001110110000001
11010100**100010101**000101011

Static heap

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
}

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5); addr1
}
```

Execution stack

Dynamic Heap

Before Execution

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
}

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```

Execution stack

Dynamic Heap

top of stack

Activation record for main

```
args = null  ret addr = OS
local1 =
```

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
}

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```
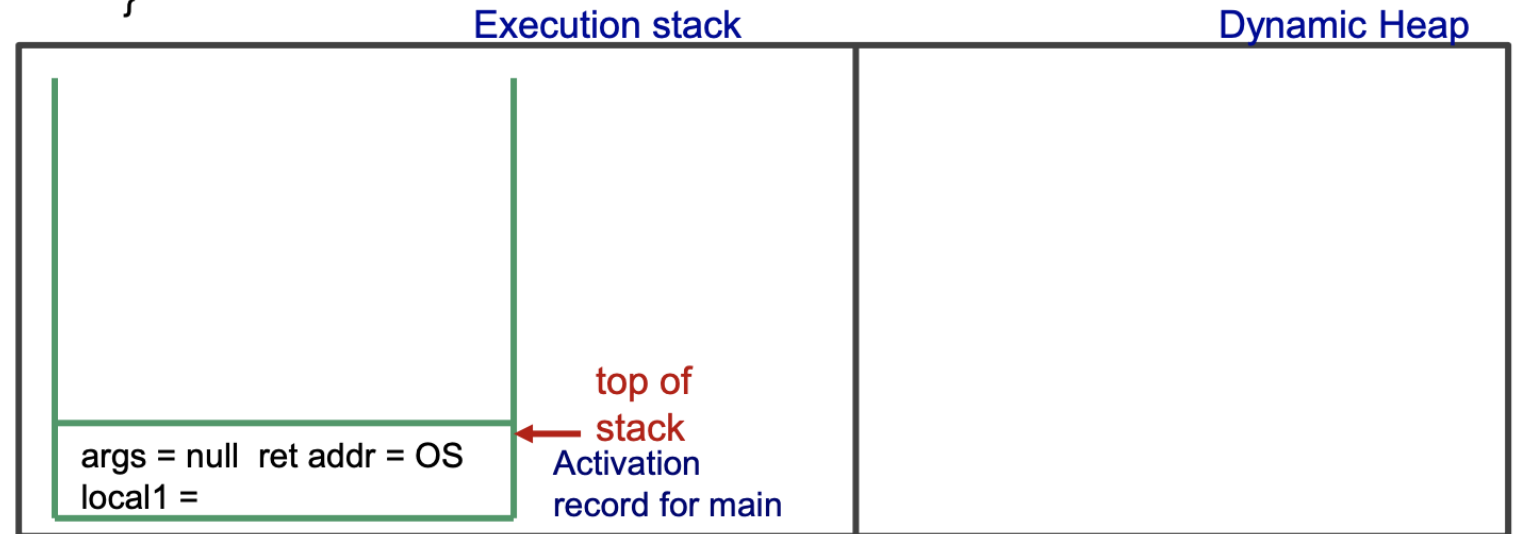
Execution stack   Dynamic Heap

```
                                    ← top of
                                       stack
param2 = 5  ret addr = addr1       Activation
local2 = 1    ret value =          record for m2
i =
args = null  ret addr = OS
local1 =
```

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
}

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```
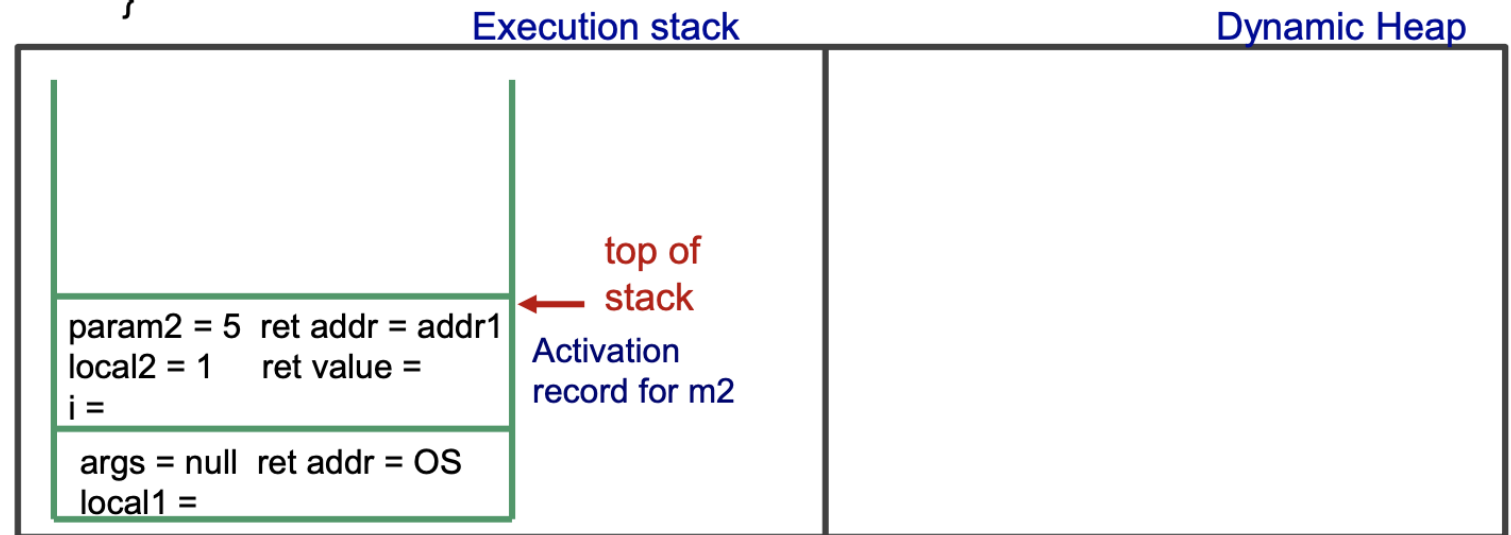
```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);   addr1
}
```

An object is created

Execution stack

Dynamic Heap

top of stack

```
param2 = 5  ret addr = addr1
local2 = 1    ret value =
i = addr4
args = null  ret addr = OS
local1 =
```

addr4

```
3
```

Integer object

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2  m3(4);
addr3  return local2 + param2 + m4(3);
}

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```
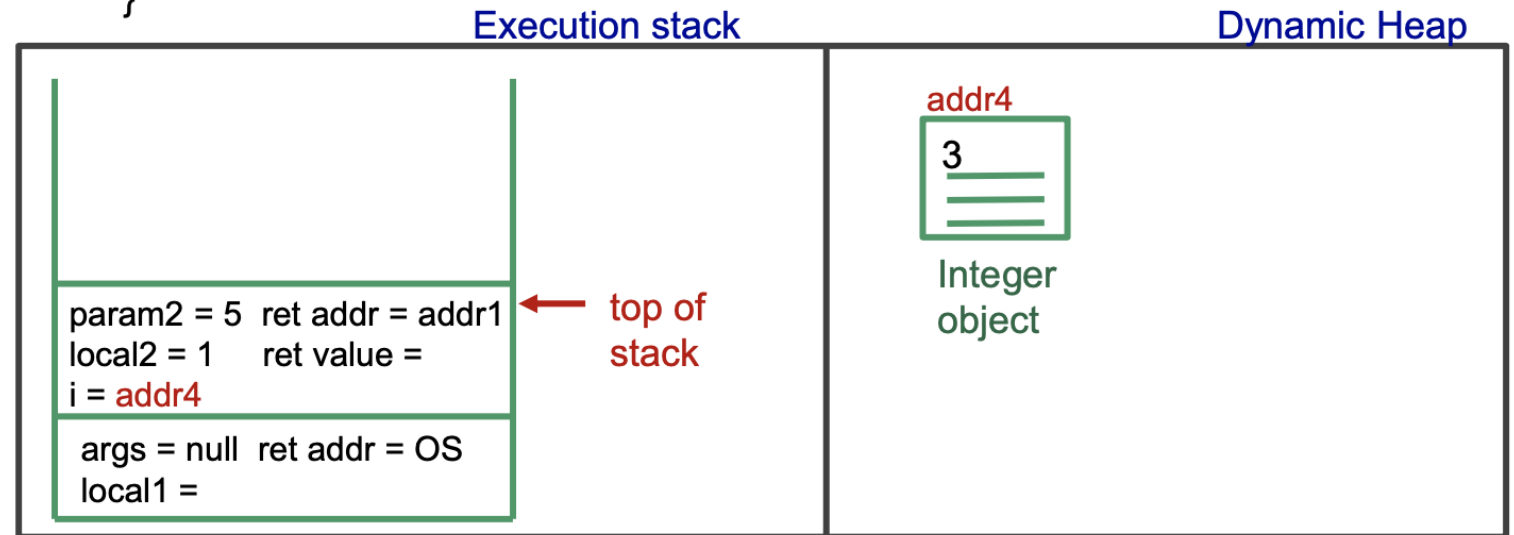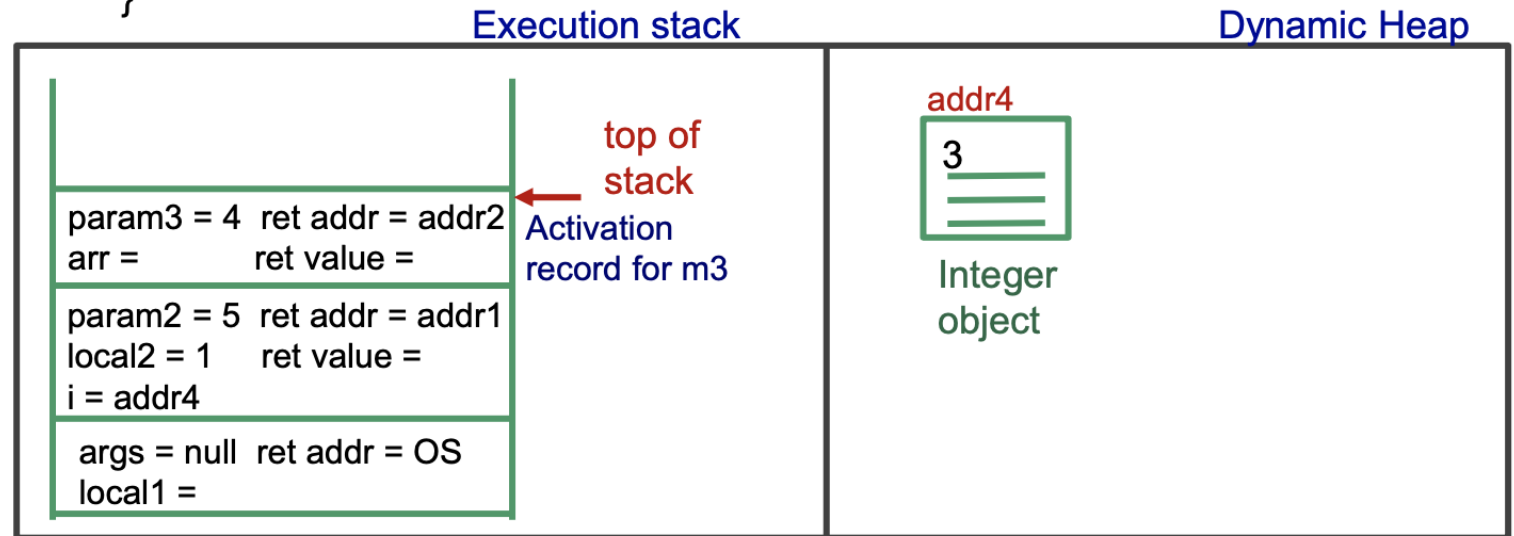
```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```

Execution stack   **Dynamic Heap**

top of
stack

param3 = 4 ret addr = addr2   Activation
arr =   ret value =   record for m3

param2 = 5 ret addr = addr1
local2 = 1   ret value =
i = addr4

args = null ret addr = OS
local1 =

addr4
3

Integer
object

Activation Records

```
public static int m2 (int param2) {            public static int m4 (int param4) {
    int local2 = 1;                                return param4 * 2;
    Integer i = new Integer(3);                }
addr2 m3(4);
addr3 return local2 + param2 + m4(3);          public static void main (String[] args) {
    }                                              int local1 = m2(5);  addr1
                                               }
    public static void m3 (int param3) {
```

int[] arr = new int[param3];   ◄─── An object is created

```
    }
```

Execution stack   **Dynamic Heap**

	addr4
	3
param3 = 4 ret addr = addr2 ◄── top of stack	
arr = addr5	Integer object
param2 = 5 ret addr = addr1	
local2 = 1 ret value =	addr5
i = addr4	
args = null ret addr = OS	array of int
local1 =	

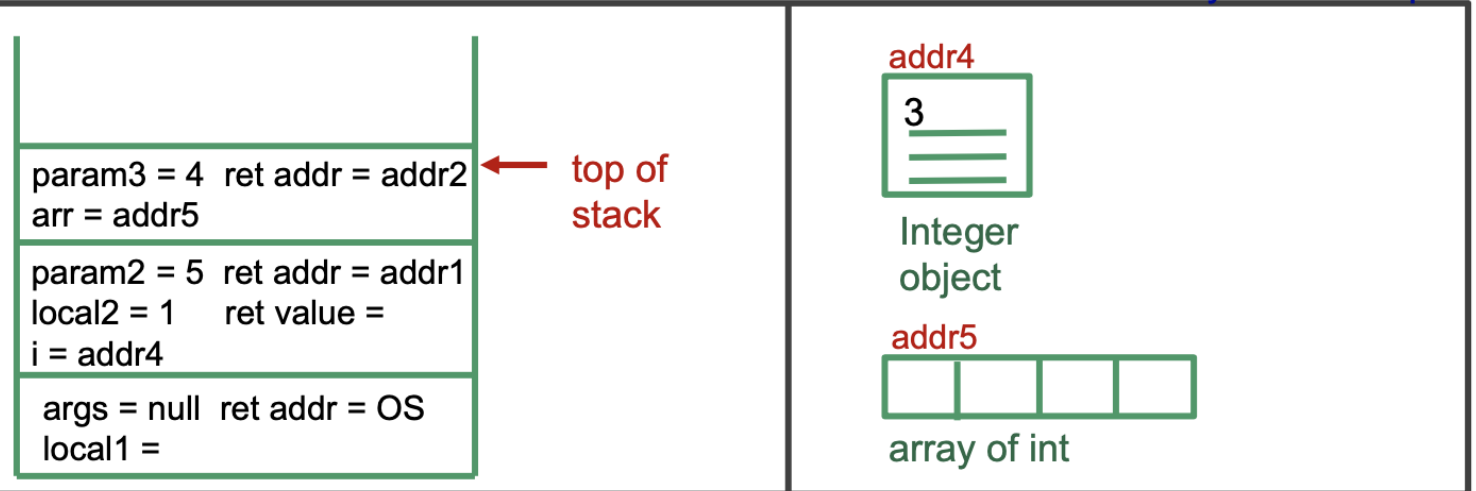# Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
    }

    public static void m3 (int param3) {
        int[] arr = new int[param3];
    }
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```
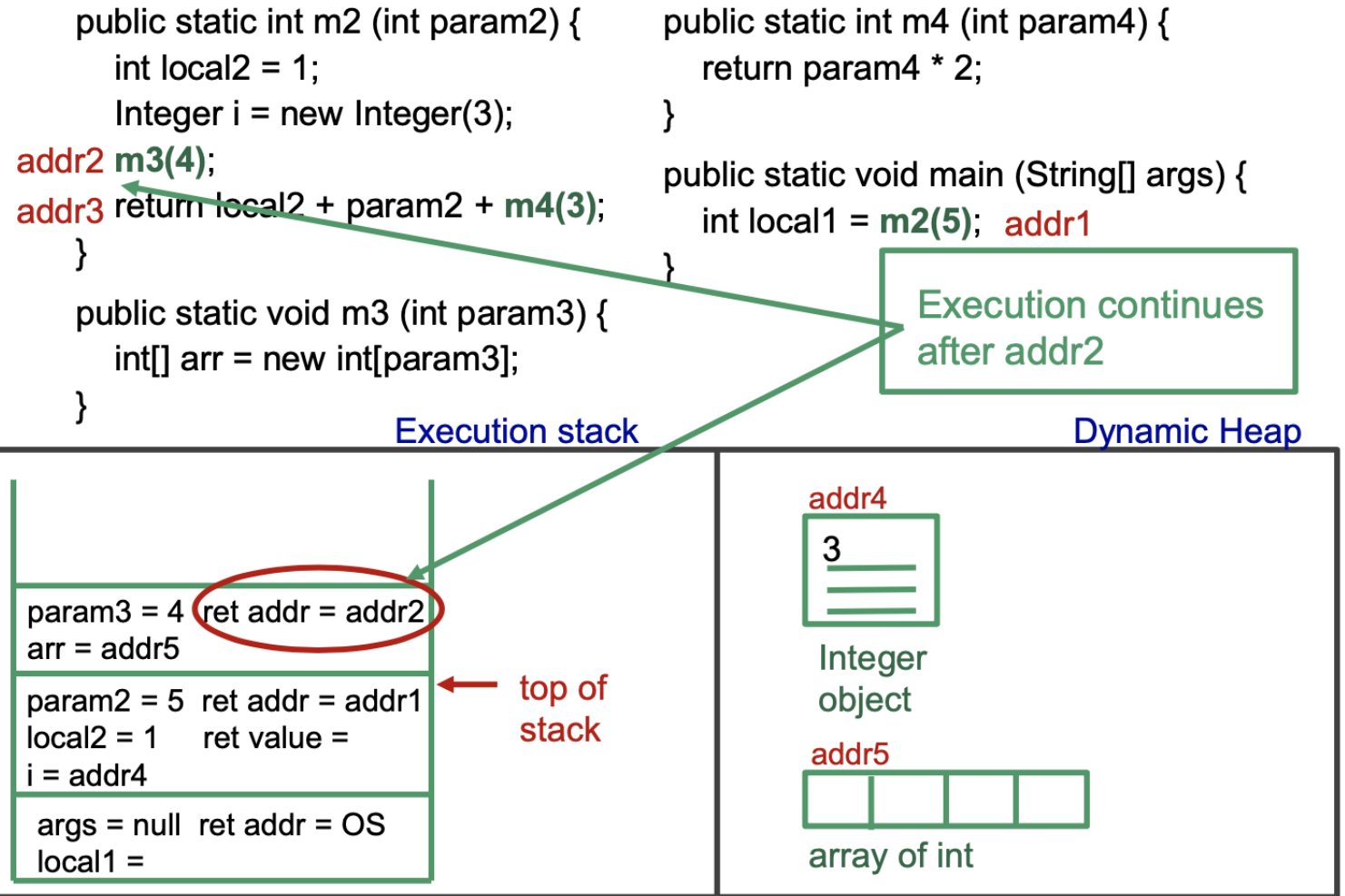
Execution continues after addr2

Execution stack

Dynamic Heap

param3 = 4 ret addr = addr2
arr = addr5

top of stack

param2 = 5 ret addr = addr1
local2 = 1   ret value =
i = addr4

args = null ret addr = OS
local1 =

addr4

3

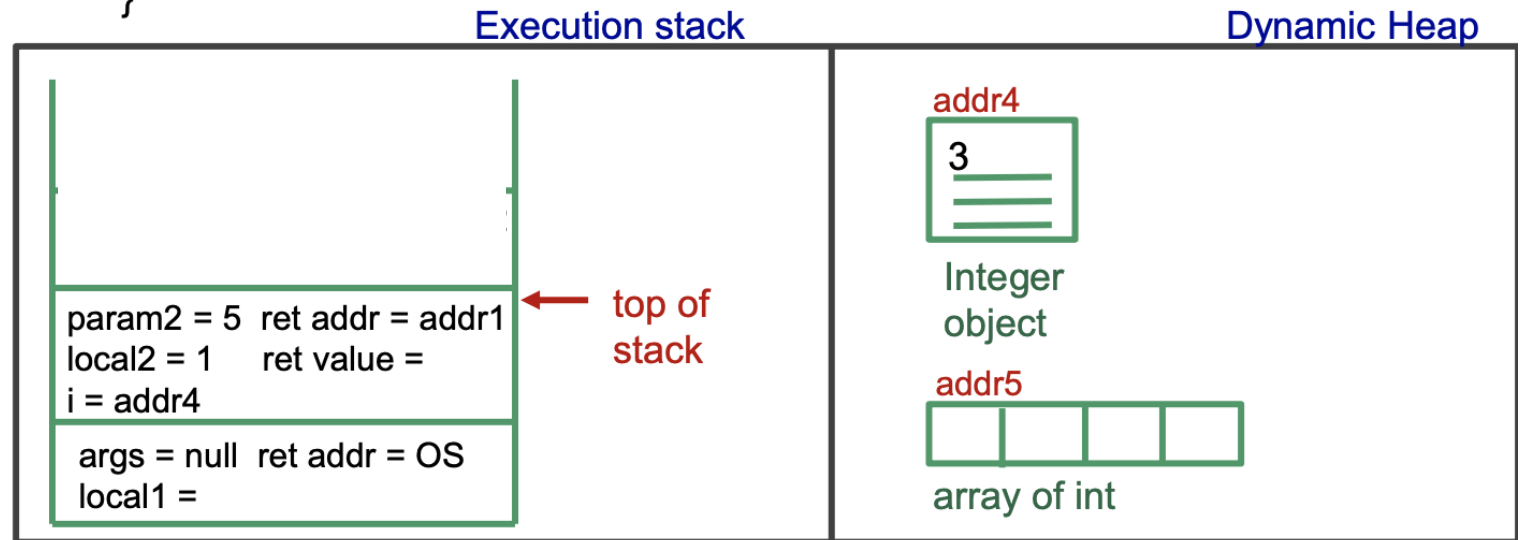Integer object

addr5

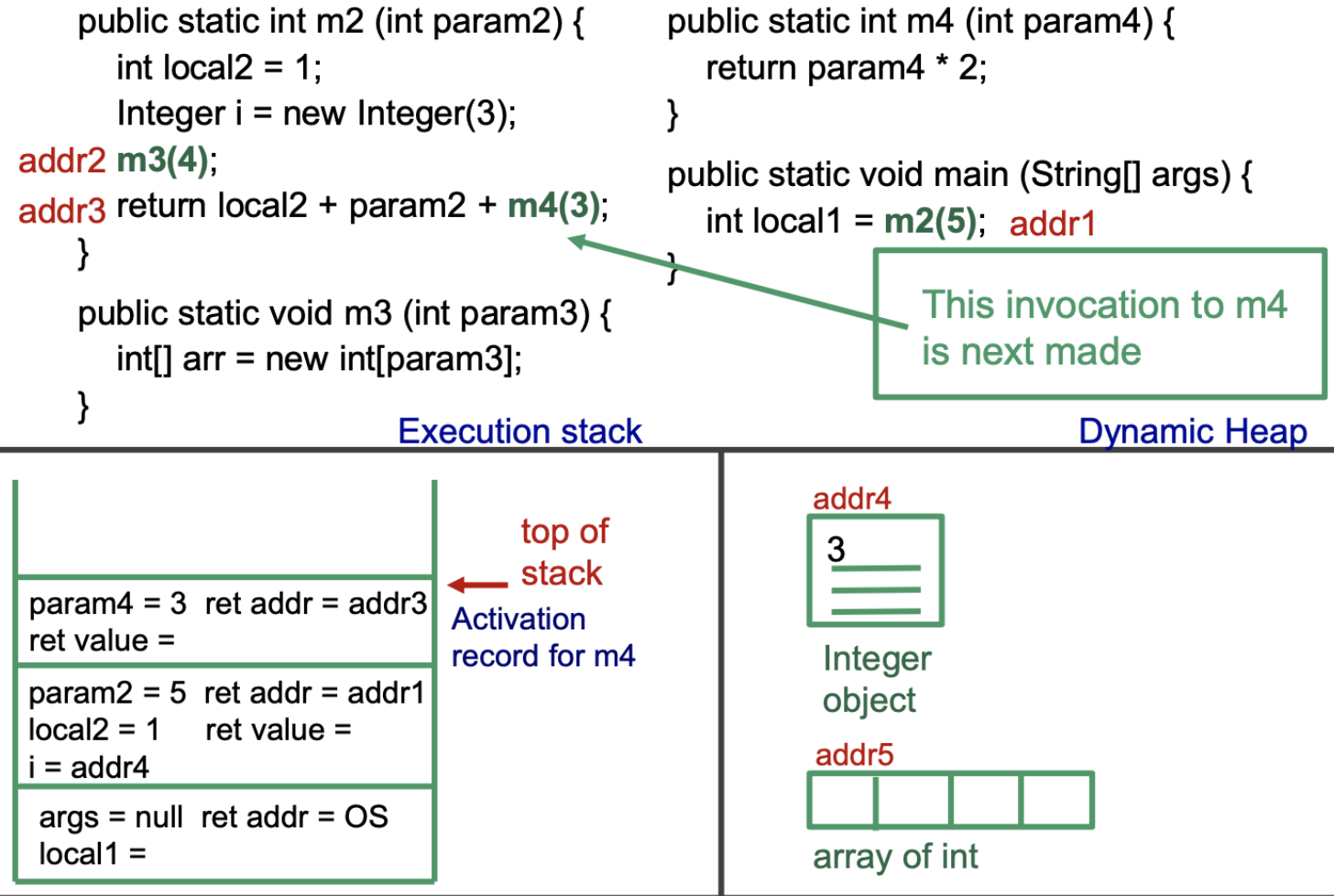array of int

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
    addr2  m3(4);
    addr3  return local2 + param2 + m4(3);
}

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```

This invocation to m4 is next made

Execution stack

Dynamic Heap

param2 = 5 ret addr = addr1 local2 = 1 ret value = i = addr4	← top of stack
args = null ret addr = OS local1 =	

addr4

3
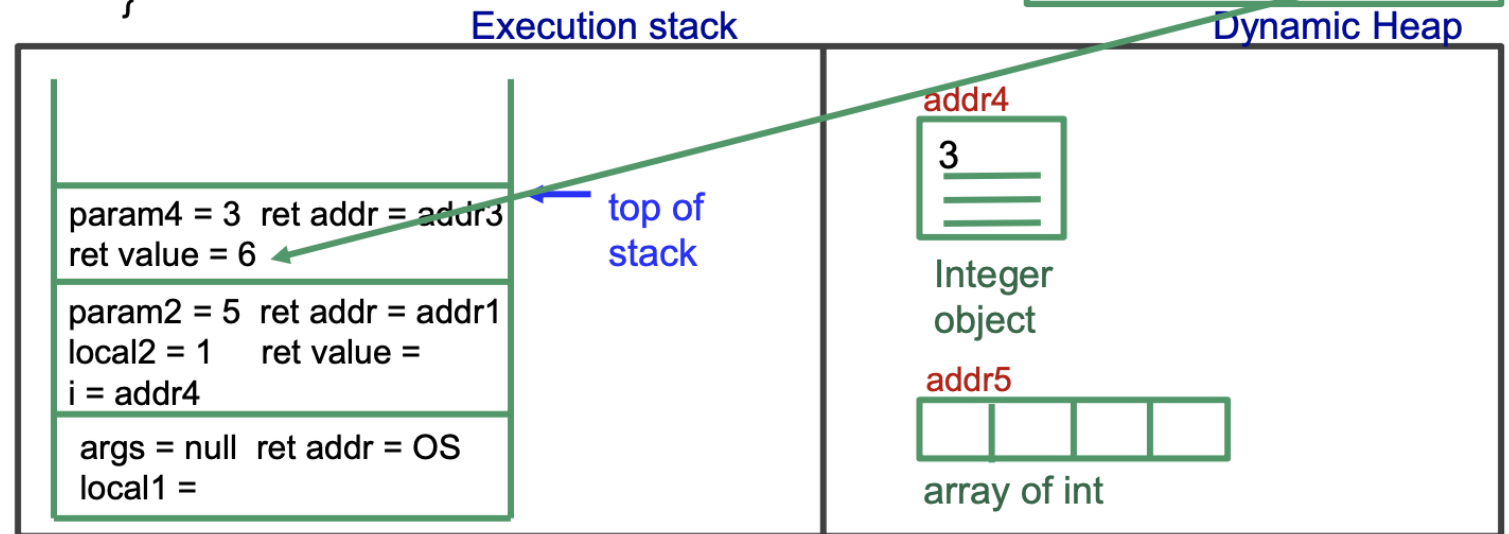
Integer object

addr5

array of int

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
    }

    public static void m3 (int param3) {
        int[] arr = new int[param3];
    }
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```

This invocation to m4 is next made

Execution stack

Dynamic Heap

top of stack

Activation record for m4

| param4 = 3 ret addr = addr3 |
| ret value = |
| param2 = 5 ret addr = addr1 |
| local2 = 1   ret value = |
| i = addr4 |
| args = null ret addr = OS |
| local1 = |

addr4

3

Integer object

addr5

array of int

Activation Records

```
public static int m2 (int param2) {          public static int m4 (int param4) {
    int local2 = 1;                               return param4 * 2;
    Integer i = new Integer(3);               }
addr2  m3(4);
addr3  return local2 + param2 + m4(3);        public static void main (String[] args) {
    }                                             int local1 = m2(5);  addr1

    public static void m3 (int param3) {      }
        int[] arr = new int[param3];
    }
```

addr1

Return value is
computed and stored
in activation record

Execution stack

Dynamic Heap

addr4

3

← top of
stack

```
param4 = 3  ret addr = addr3
ret value = 6
```

Integer
object

```
param2 = 5  ret addr = addr1
local2 = 1     ret value =
i = addr4
```

addr5

```
args = null  ret addr = OS
local1 =
```

array of int

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
    }

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```
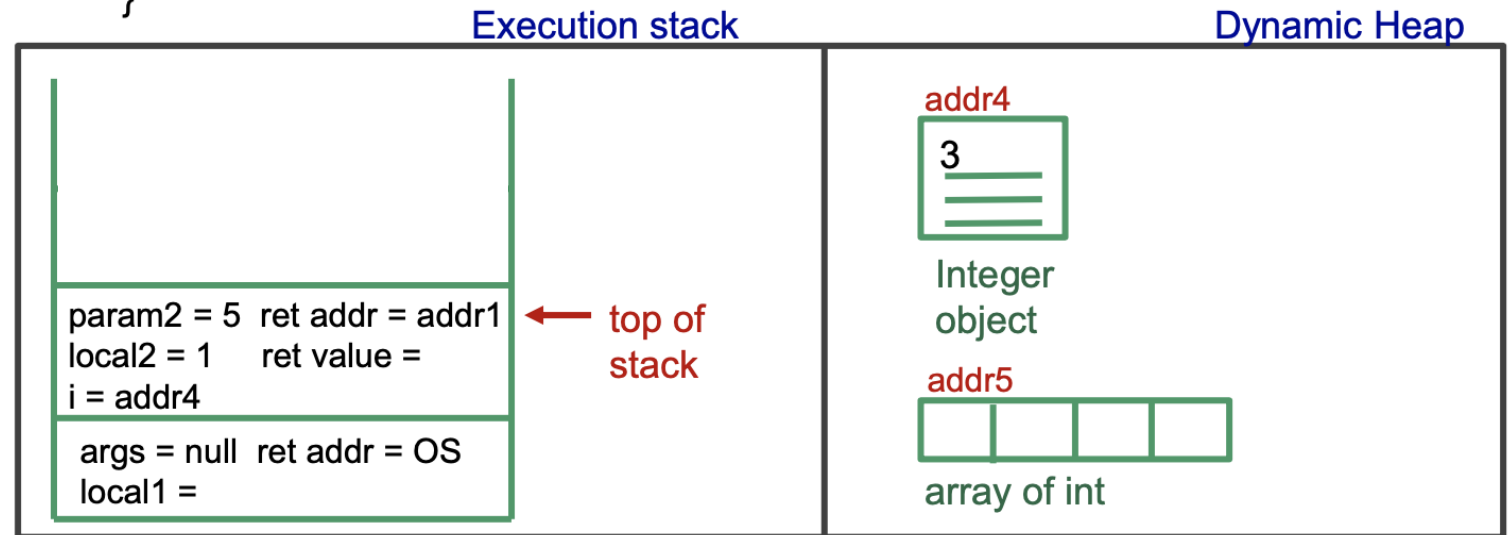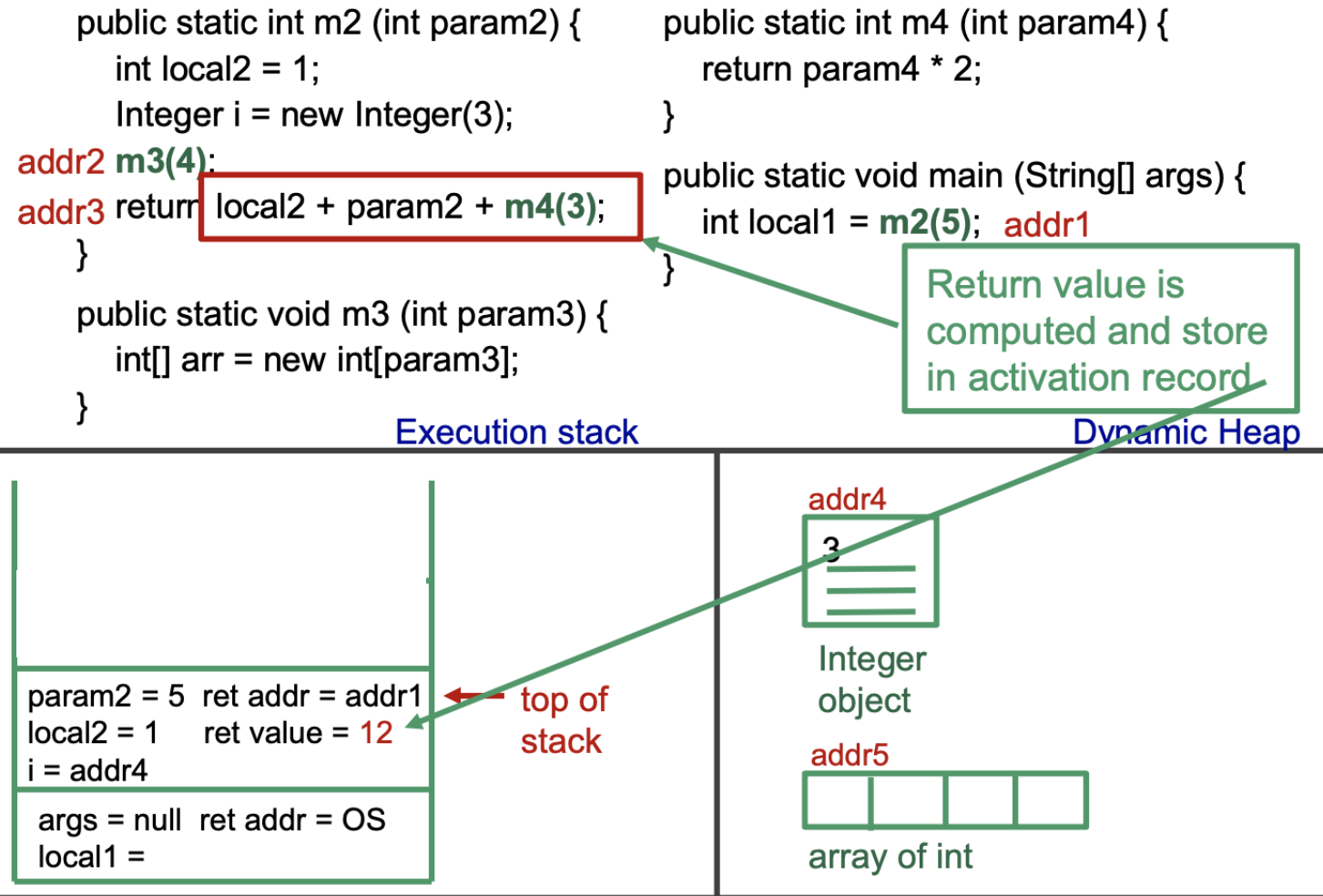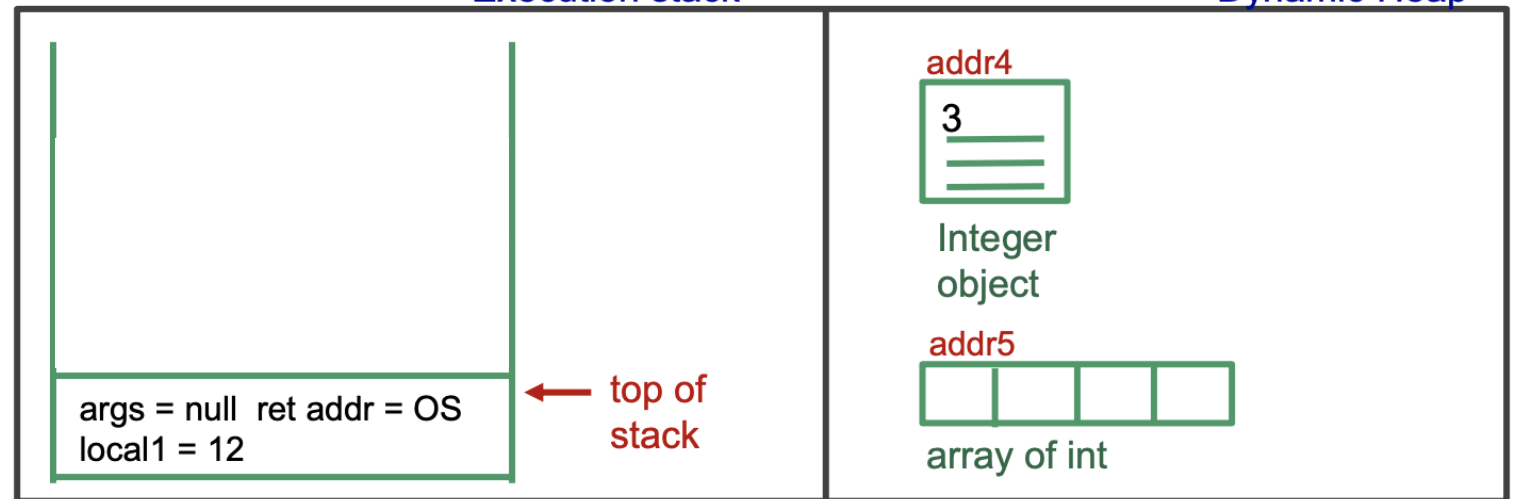
Execution continues after addr3

Execution stack

Dynamic Heap

```
param2 = 5  ret addr = addr1
local2 = 1     ret value =
i = addr4
```
← top of stack

```
args = null  ret addr = OS
local1 =
```

addr4

3

Integer object

addr5

array of int

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2  m3(4);
addr3  return  local2 + param2 + m4(3);
    }

public static void m3 (int param3) {
    int[] arr = new int[param3];
}
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```

Return value is computed and store in activation record

Execution stack

Dynamic Heap

param2 = 5 ret addr = addr1 local2 = 1 ret value = 12 i = addr4	← top of stack
args = null ret addr = OS local1 =	

addr4

3

Integer object

addr5

array of int

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
    }

    public static void m3 (int param3) {
        int[] arr = new int[param3];
    }
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);  addr1
}
```

Execution continues after addr1

Execution stack

Dynamic Heap

addr4

3

Integer object

addr5

array of int

args = null ret addr = OS
local1 = 12

← top of stack

Activation Records

```
public static int m2 (int param2) {
    int local2 = 1;
    Integer i = new Integer(3);
addr2 m3(4);
addr3 return local2 + param2 + m4(3);
    }

    public static void m3 (int param3) {
        int[] arr = new int[param3];
    }
```

```
public static int m4 (int param4) {
    return param4 * 2;
}

public static void main (String[] args) {
    int local1 = m2(5);   addr1
}
```

Program ends and control goes back to Operating System

Execution stack

Dynamic Heap

addr4

3

Integer object

addr5

array of int

top of stack
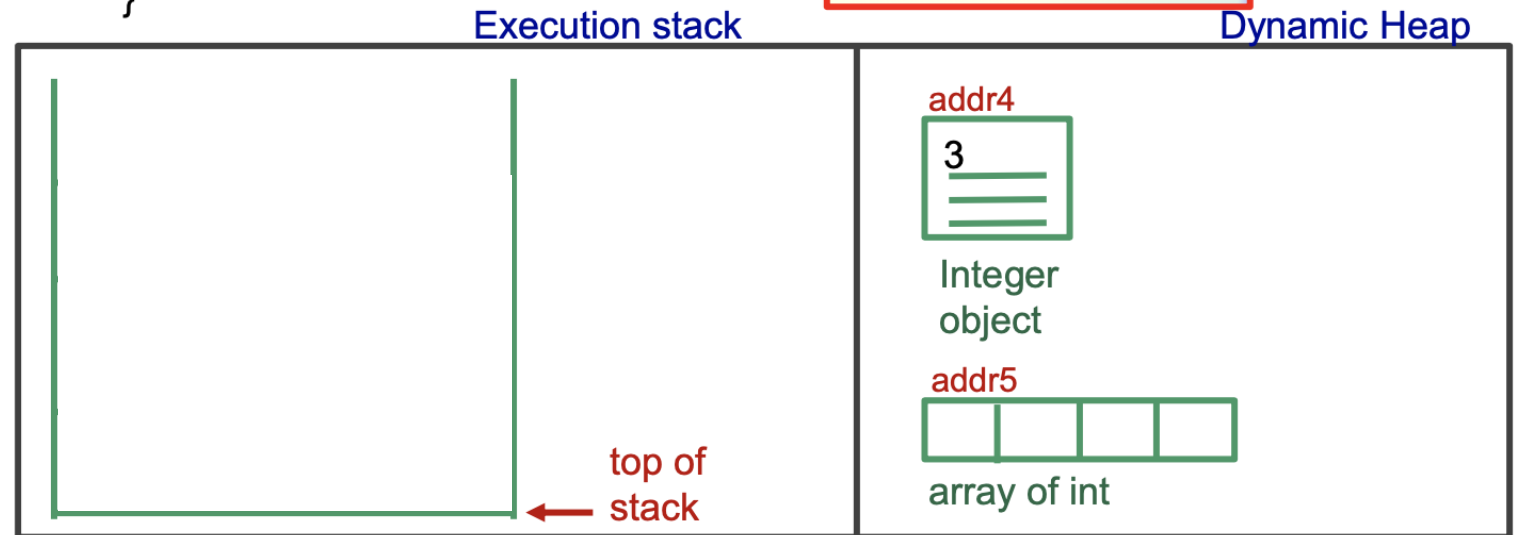
Activation Records – Example 2

```java
public class CallFrameDemo2 {
private static void printAll (String s1, String
s2, String s3) {
System.out.println(s1.toString( ));
System.out.println(s2.toString( ));
System.out.println(s3.toString( ));
}

public static void main (String args[ ]) {
String str1, str2, str3;
str1 = new String(" string 1 ");
str2 = new String(" string 2 ");
str3 = new String(" string 3 ");
printAll(str1, str2, str3);
  }
}
```

Activation Records – Example 2

Draw a picture of the execution stack and of the heap as the above program executes:

1. Activation record for main

2. Activation record for String constructor for str1 – then popped off

3. Activation record for String constructor for str2 – then popped off

4. Activation record for String constructor for str3 – then

5.   popped off

6. Activation record for printAll

7. Activation record for toString for str1 – then popped off

8. Activation record for System.out.println   – then popped off

9. etc.

Activation Records – Example 2

- What will be stored in the activation record for **main**?
  - Address to return to operating system
  - Variable **args**
  - Variable **str1**
    - Initial value?
    - Value after return from String constructor?
  - Variable **str2**
  - Variable **str3**
- What will be in the activation record for **printAll**?

Memory Deallocation

- What happens when a method returns?
  - On the execution stack:
    - The activation record is popped off when the method returns
    - So, that memory is deallocated

Memory Deallocation

- What happens to objects on the heap?
  - An object stays in the heap even if no variable is referencing it!
  - So, Java has automatic garbage collection
    - When memory runs low, objects that no longer have a variable referencing them are identified, and their memory is deallocated.

Recursive Definitions

Recursive Definition

- Defining something in terms of a smaller or simpler version of itself.

- A recursive definition consists of two parts:

  - The *base case*: this defines the *simplest* case or starting point

  - The *recursive part* is the general case that describes all the other cases in terms of smaller

    versions of itself.

  - Example:

```java
// Recursive method to calculate factorial
public static int factorial(int n) {
    // Base case: if n is 1, return 1 (factorial of 1 is 1)
    if (n == 1) {return 1;}
    // Recursive part: n * factorial of (n - 1)
    else {return n * factorial(n - 1);}
}
```

base

recursive

Why is a base case needed in any recursive algorithm?

Recursion
vs. Iteration

- What is iteration? Repetition, as in a loop
- What is recursion? Defining something in terms of a smaller or simpler version of itself (why smaller/simpler?)
- Recursion is a very powerful problem-solving technique.
- Many complex problems would be very difficult to solve without the use of recursion.

Example of Recursive Problem

- Consider the problem of computing the sum of all the numbers between 1 and n:

$$1 + 2 + 3 + 4 + \ldots + \text{n-1} + \text{n}$$

- Here is a simple iterative algorithm for this problem:

```
Algorithm sum (n)
total = 0
for i = 1 to n do
   total = total + i
return total
```

Example of Recursive Problem

- Consider the problem of computing the sum of all the numbers between 1 and n:

$$1 + 2 + 3 + 4 + \dots + n\text{-}1 + n$$

- Recursive definition:

```
sum of 1 to 1 => 1 (base case)
sum of 1 to n => n + the sum of 1 to n-1, for n > 1
```

$$\sum_{k=1}^{n} k = n + \sum_{k=1}^{n-1} k \quad \text{(recursive case)}$$

Recursive Algorithm

- Recursive definition:

  `sum of 1 to 1 => 1 (`*base case*`)`

  `sum of 1 to n => n + `**`the sum of 1 to n-1`**`, for n > 1`

- Recursive algorithm for this problem:

  `Algorithm sum(n)`

  `In: Positive value n`

  `Out: 1 + … + n`

  `if n = 1 then return 1 // base case`

  `else return n + sum(n-1) // recursive case`
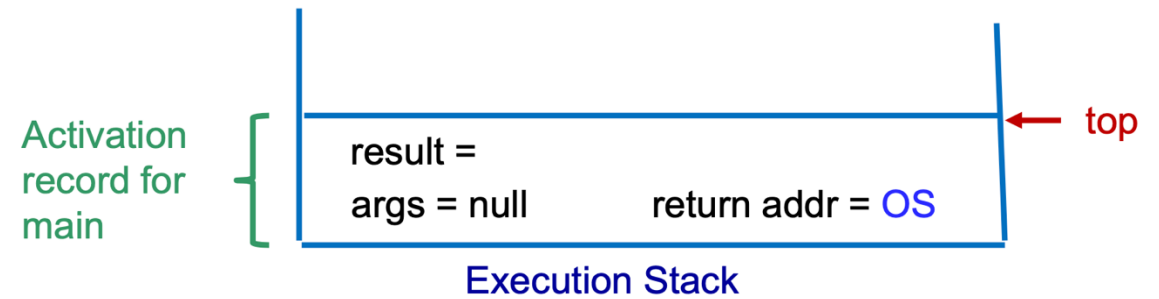
How Recursion Works

- Consider the following program

```
public static void main (String[] args) {int result = sum(4);} // addr 1
public static int sum (int n) {if (n == 1) return 1;
                                else return n + sum(n-1);} // addr2
```
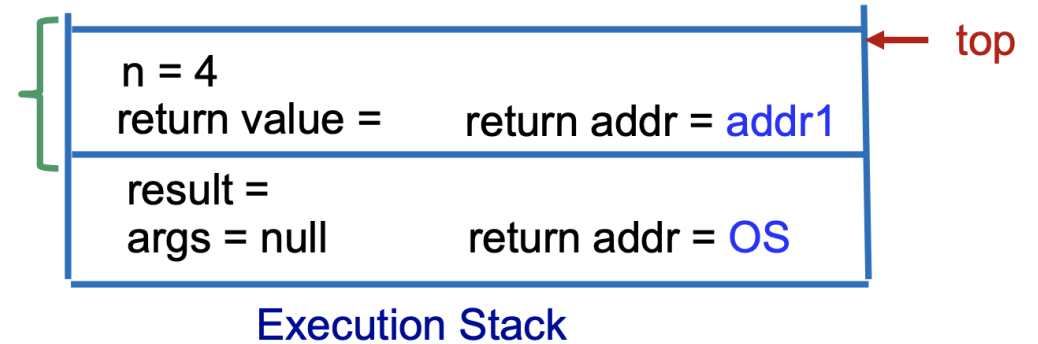
- An **activation record** is created for the method main when the program is executed. This activation record stores:
  - The return address **addr1**
  - The variable **result**
  - The parameter **args**

How Recursion Works

```
public static void main (String[] args)
{int result = sum(4);} // addr 1
public static int sum (int n) {
if (n == 1) return 1;
else return n + sum(n-1);} // addr2
```


Activation record for main

result =
args = null   return addr = OS

top

Execution Stack

- At this point, the execution stack looks like the following figure. We assume no parameter is passed to the main function, so args is null.

- The result variable has no value assigned to it yet, so we left its value blank. OS denotes the address of the virtual machine's instruction where the main method was invoked.

How Recursion Works

```
public static void main (String[] args)

{int result = sum(4);} // addr 1

public static int sum (int n) {

if (n == 1) return 1;

else return n + sum(n-1);} // addr2
```
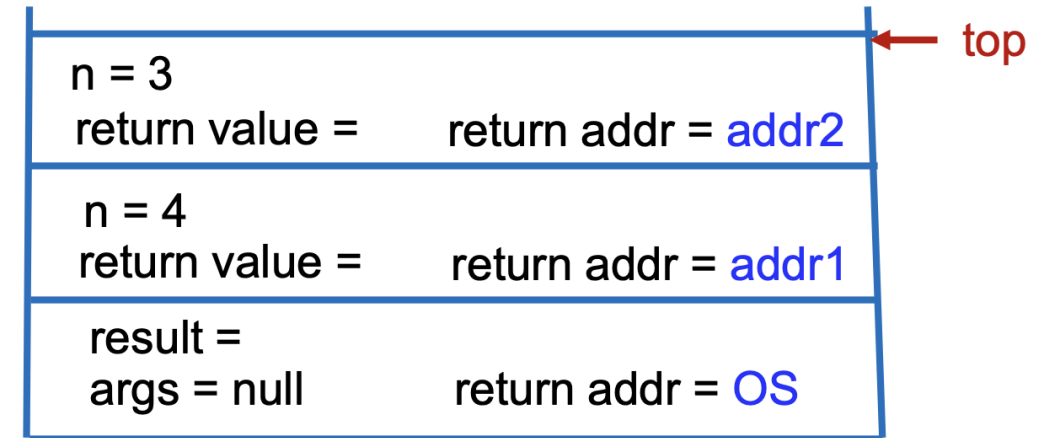
Activation record for sum

n = 4	
return value =	return addr = addr1
result =	
args = null	return addr = OS

← top

Execution Stack

- Once the activation record for the main function has been created and the values of the parameters and return address have been stored, the execution of the method main starts.
- The first and only statement of the main function invokes method **sum**(4). This creates another activation record, which is pushed into the execution stack, as shown above.

How Recursion Works (cont.)

```
public static void main (String[] args)

{int result = sum(4);} // addr 1

public static int sum (int n) {

if (n == 1) return 1;

else return n + sum(n-1);} // addr2
```
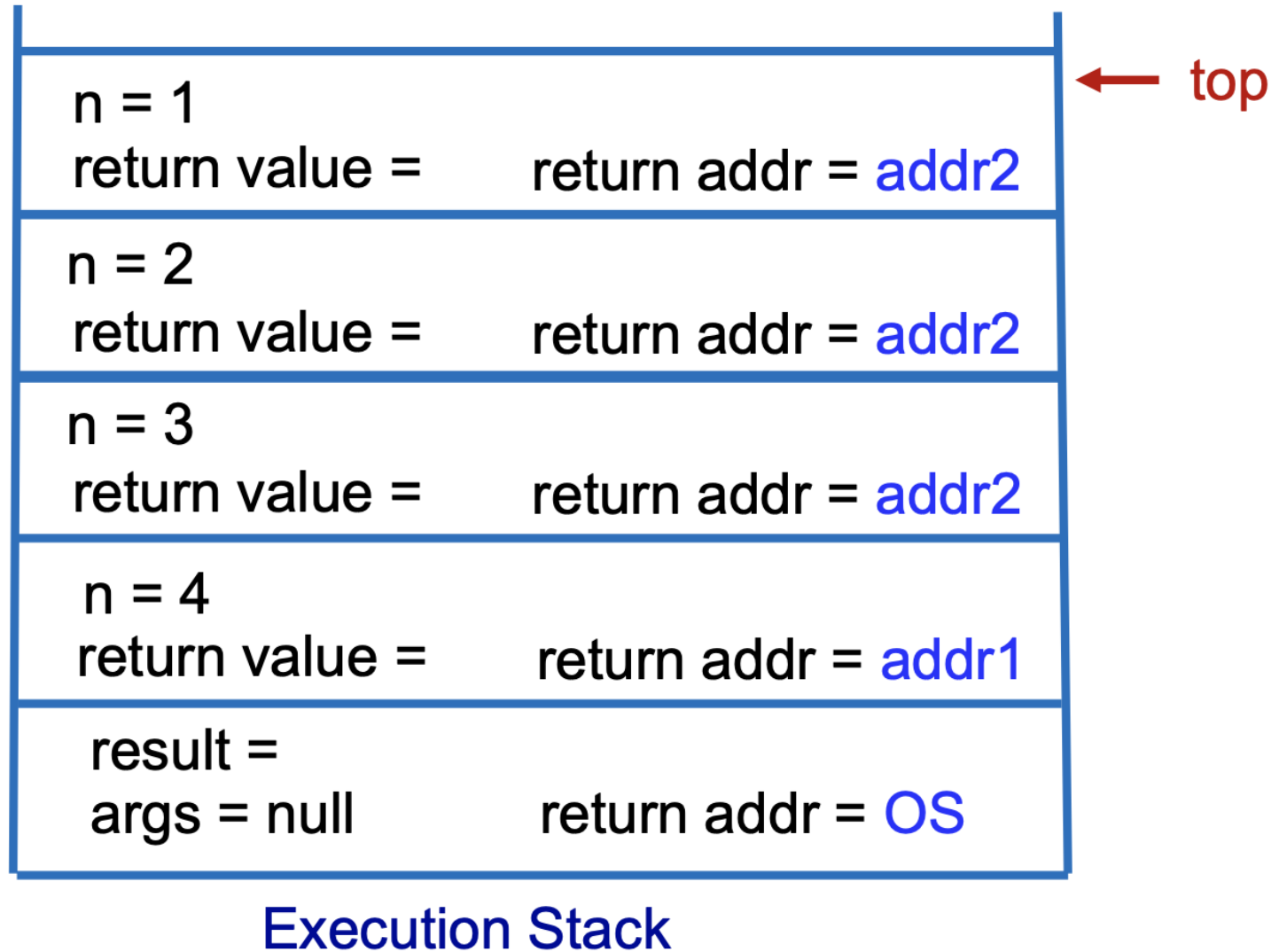
n = 3	← top
return value =	return addr = addr2
n = 4	
return value =	return addr = addr1
result =	
args = null	return addr = OS

Execution Stack

- Once the activation record has been created, the execution of the method sum starts.

- Since n > 1, method sum (n-1) is invoked.

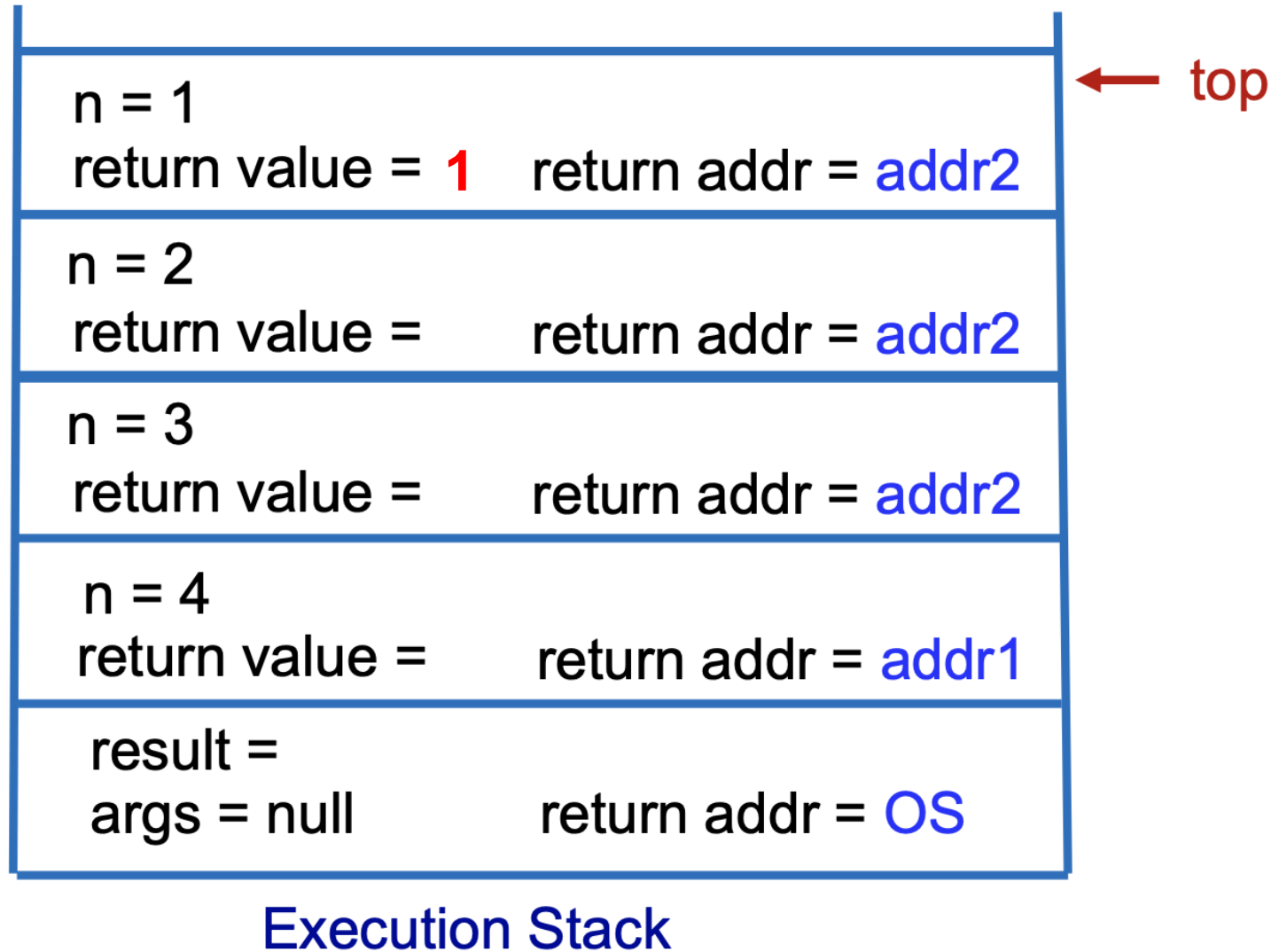- A new activation record is created and pushed into the stack

How Recursion Works (cont.)

- Then, two more invocations to the method sum with parameters 2 and 1 are made.
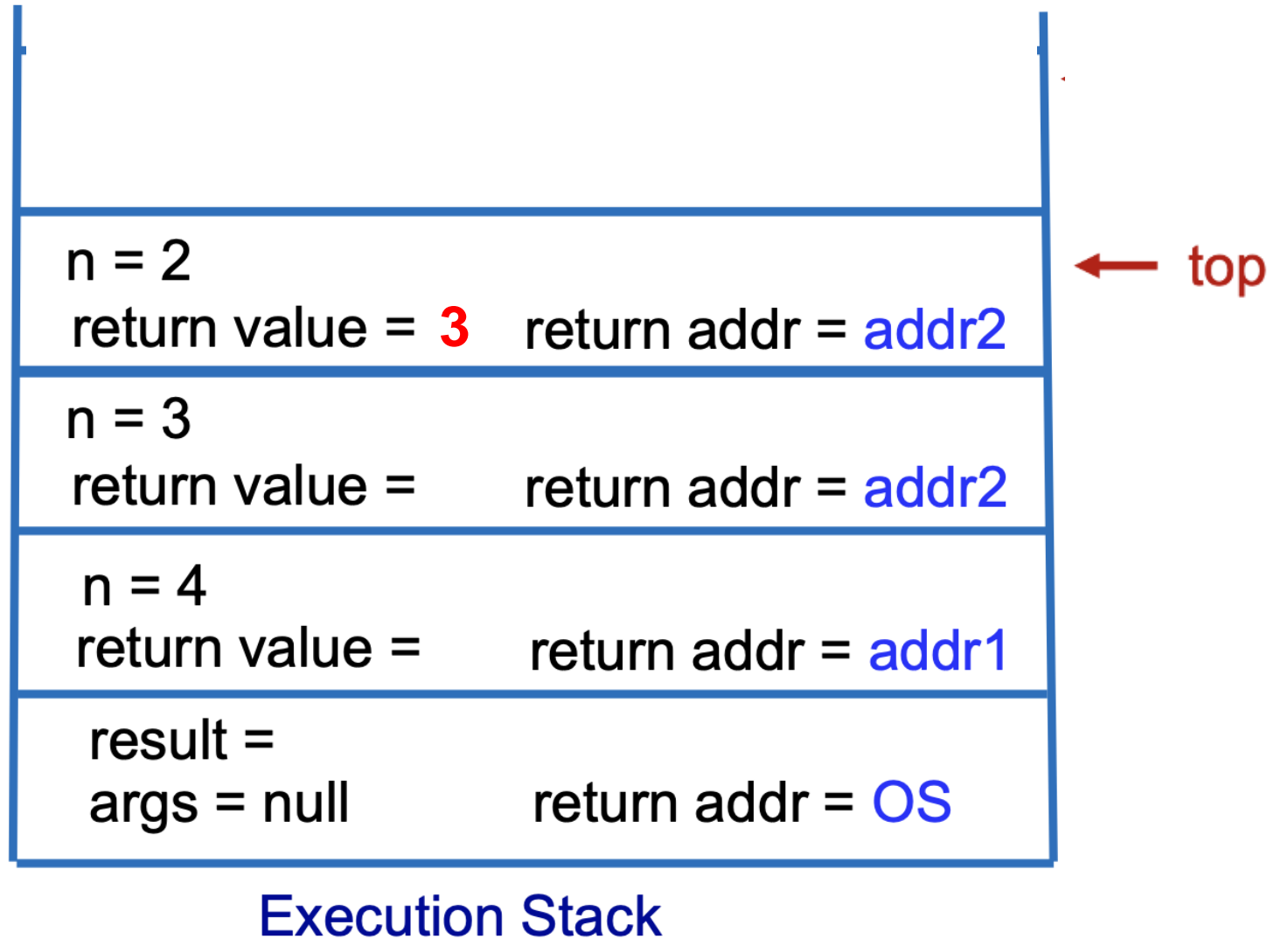- After the last invocation, the execution stack looks like the figure given.

n = 1
return value =   return addr = addr2

n = 2
return value =   return addr = addr2

n = 3
return value =   return addr = addr2

n = 4
return value =   return addr = addr1

result =
args = null   return addr = OS

← top

Execution Stack

How Recursion Works (cont.)

- Since the value of n is 1 in the last invocation of the method sum, the statement return 1 (base case) is executed.
- The value 1 is stored in the **return value**.

n = 1
return value = **1**   return addr = addr2

n = 2
return value =   return addr = addr2

n = 3
return value =   return addr = addr2

n = 4
return value =   return addr = addr1

result =
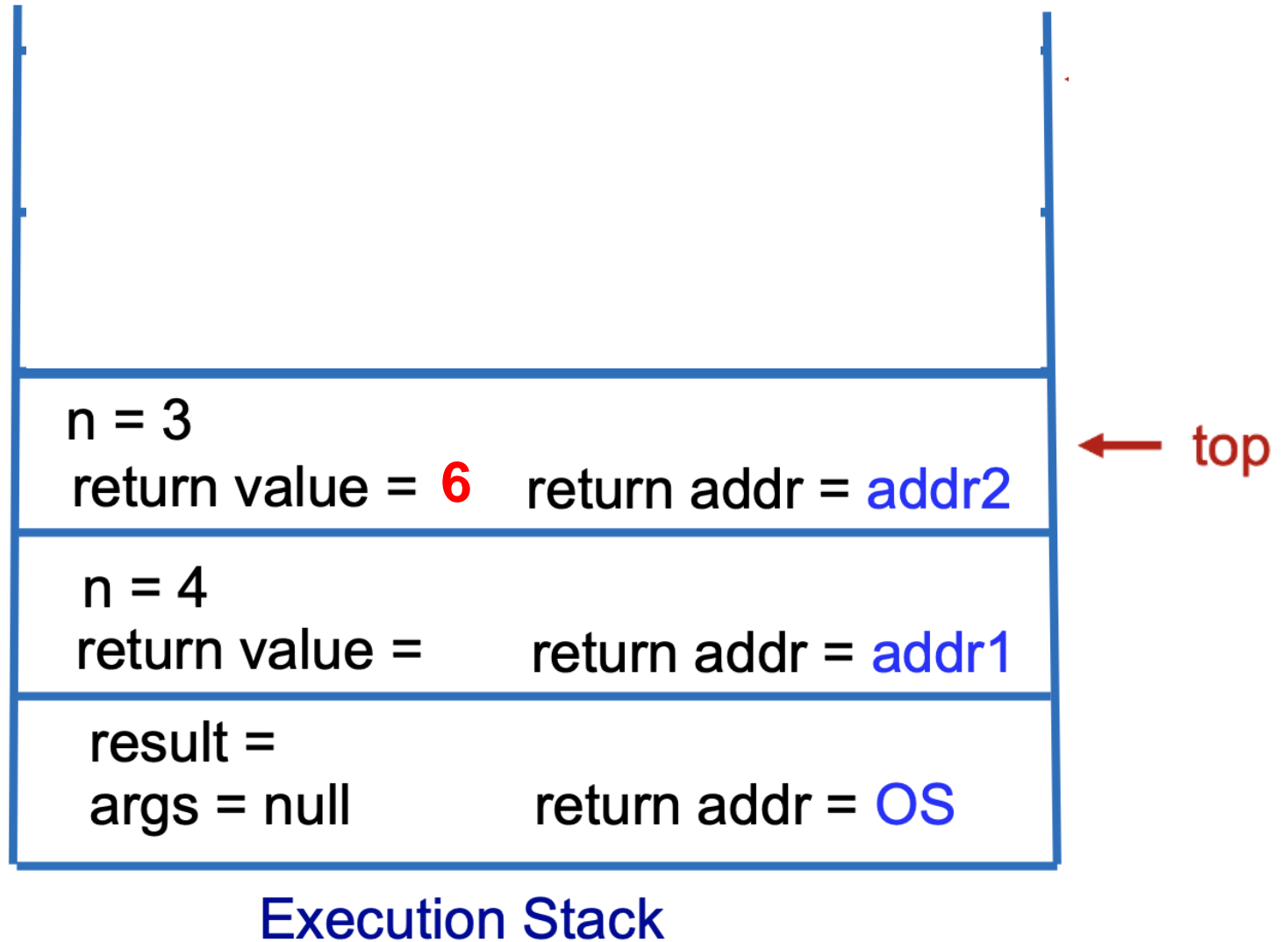args = null   return addr = OS

← top

Execution Stack

How Recursion Works (cont.)

- The method sum ends, and hence, an activation record is popped off the execution stack.
- The return address addr2 is recovered, and execution continues at the statement in that address: This call just finished, and it returned the value 1.
- Hence, n + sum(n-1) = 2 + 1 = 3 will be returned.

n = 2
return value = **3**   return addr = addr2

n = 3
return value =   return addr = addr2

n = 4
return value =   return addr = addr1

result =
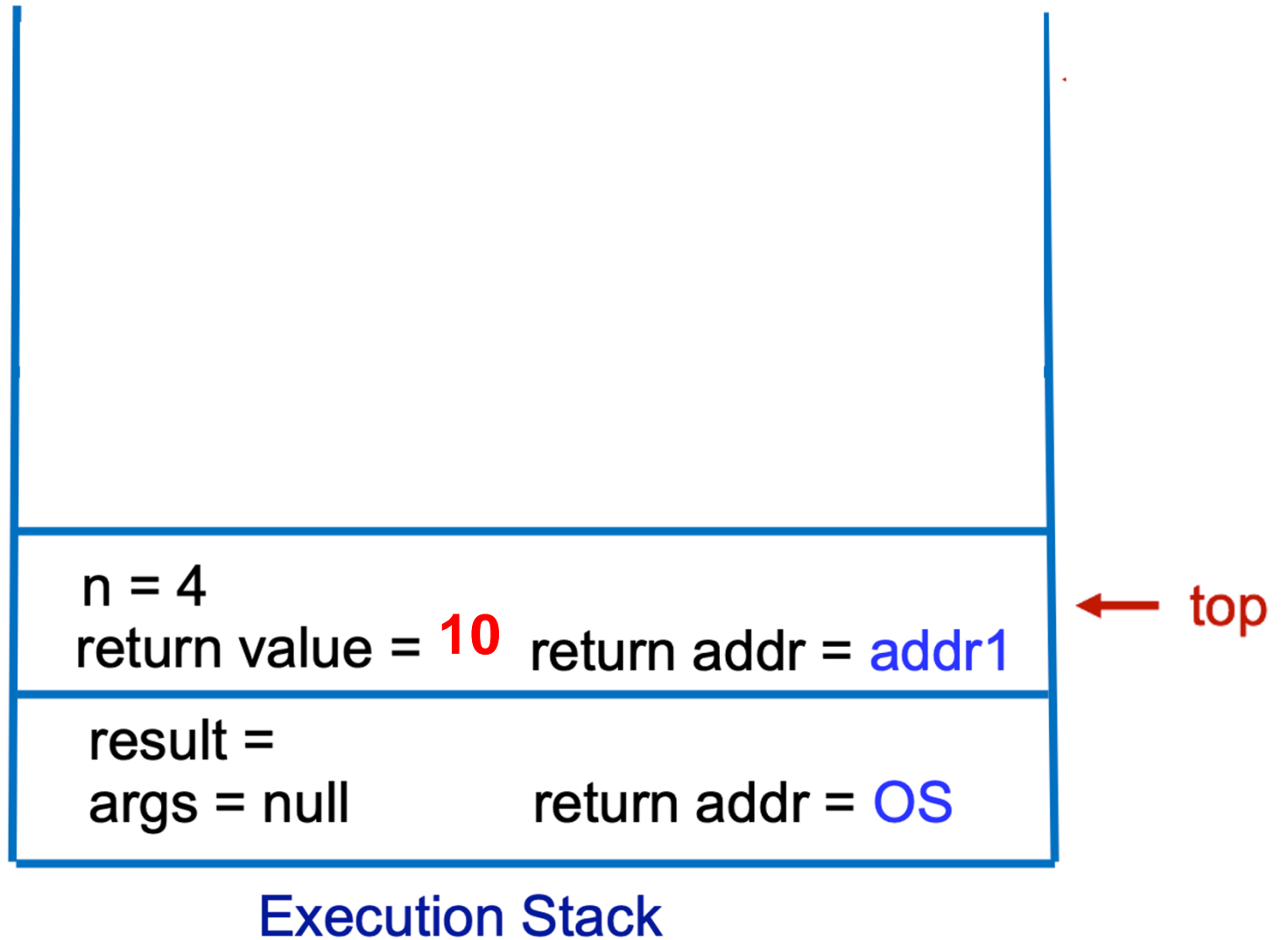args = null   return addr = OS

← top

Execution Stack

How Recursion Works (cont.)

- The next call returns the value 3, and an activation record is popped off the execution stack. The return address addr2 is recovered, and execution continues with the statement in that address.
- The value n + sum(n-1) = 3 + 3 = 6 will be returned.

n = 3
return value = **6**   return addr = addr2   ← top

n = 4
return value =   return addr = addr1

result =
args = null   return addr = OS

Execution Stack

How Recursion Works (cont.)

n = 4
return value = **10**   return addr = addr1   ← top

result =
args = null   return addr = OS

Execution Stack

How Recursion Works (cont.)

```
public static void main
(String[] args) {int result =
sum(4);} // addr 1
```

result = **10**
args = null   return addr = OS

← top
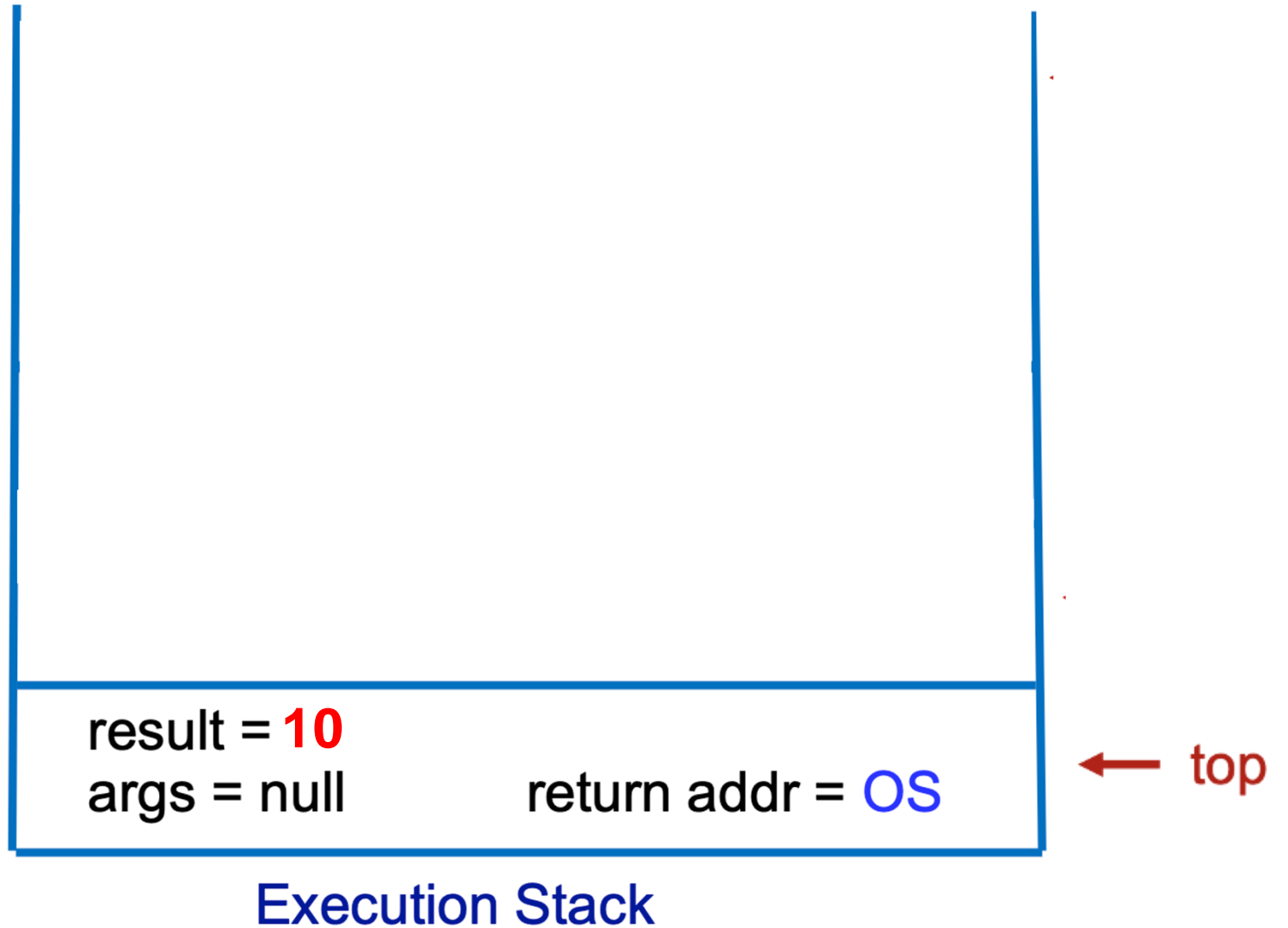
Execution Stack

Recursion vs. Iteration

- Every recursive algorithm can also be written as an **iterative** algorithm. However, the algorithm could be much more complex and require the use of an auxiliary stack or other data structures to simulate the execution stack.
- Thus, just because we can use recursion to solve a problem does not mean we should!
- Would you use iteration or recursion to compute the sum of 1 to n? Why?

Recursion vs. Iteration

- Recursion often uses more memory and can lead to <u>stack overflow errors</u> if the recursive depth is too large. Sometimes, an iterative (loop-based) solution is faster and more memory-efficient.