Java Memory Management

Objectives

- Understand how the memory of a computer is used when executing a program
- Identify the different parts of memory for storing classes, objects, and the execution stack

Memory Allocation in Java

- When a program is being executed, separate areas of memory are allocated for
  - code (classes)
  - objects
  - execution stack

Memory Areas

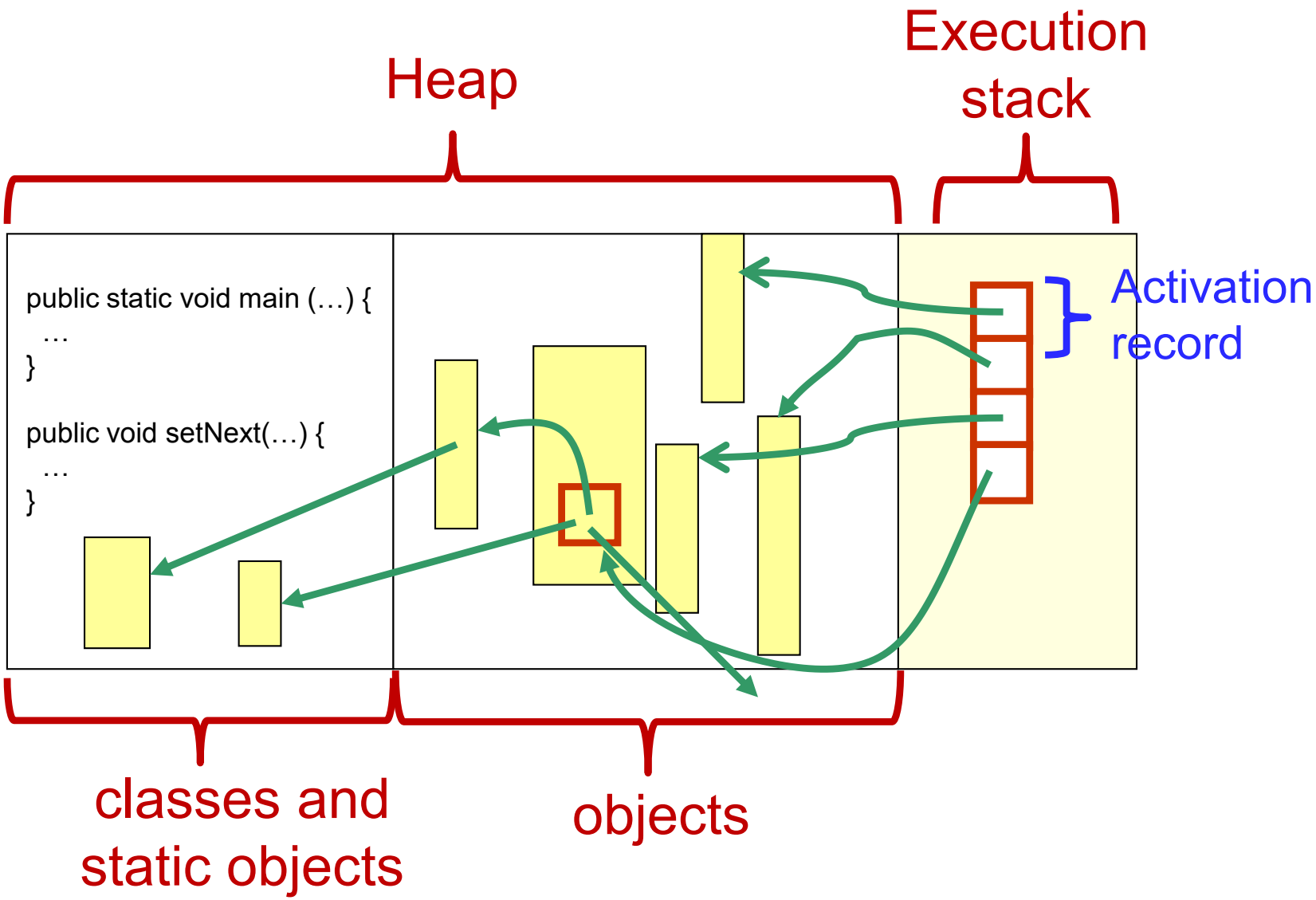
- ***Execution stack (also called runtime stack or call stack)***

  Used to store *method* information needed *while the method is being executed,* like

  - Local variables
  - Formal parameters
  - Return value
  - Return address

- ***Heap***
  - Used to store
    - *Code*
    - *Objects*

Memory Allocated to a Program



Memory Allocation in Java

- What happens when an object is created by **new**, as in

  Person friend = new Person(…);

  Declaring variable => execution stack

  Create the memory => the heap.

  - The reference variable friend has memory allocated to it in the ***execution stack***
  - The object is created using memory in the ***heap***

Execution Stack

- ***Execution stack (runtime stack)*** is the memory space used to store the information needed by a method, *while the method is being executed*
- When a method is invoked, an ***activation record*** (or ***call frame***) for that method is created and pushed onto the execution stack
  - All the information needed during the execution of the method is stored in an activation record

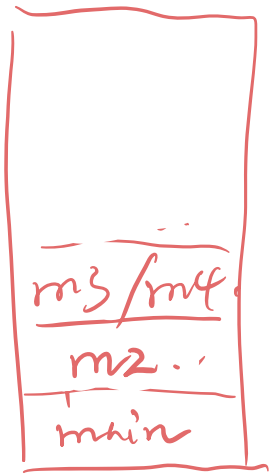
Activation Record

- An ***activation record*** contains:
  - Address to return to after method ends
  - Method's formal parameter variables
  - Method's local variables
  - Return value (if any)

- Note that the values in an activation record are accessible ***only*** while the corresponding method is being executed!

```
public static void m4( ) {
    System.out.println("Starting m4");
    System.out.println("Leaving m4");
    return;
}

public static void main(String args[ ]) {
    System.out.println("Starting main");
    System.out.println("main calling
    m2");
    m2( );
    System.out.println("Leaving main");
}
```

push → pop ← once it's done

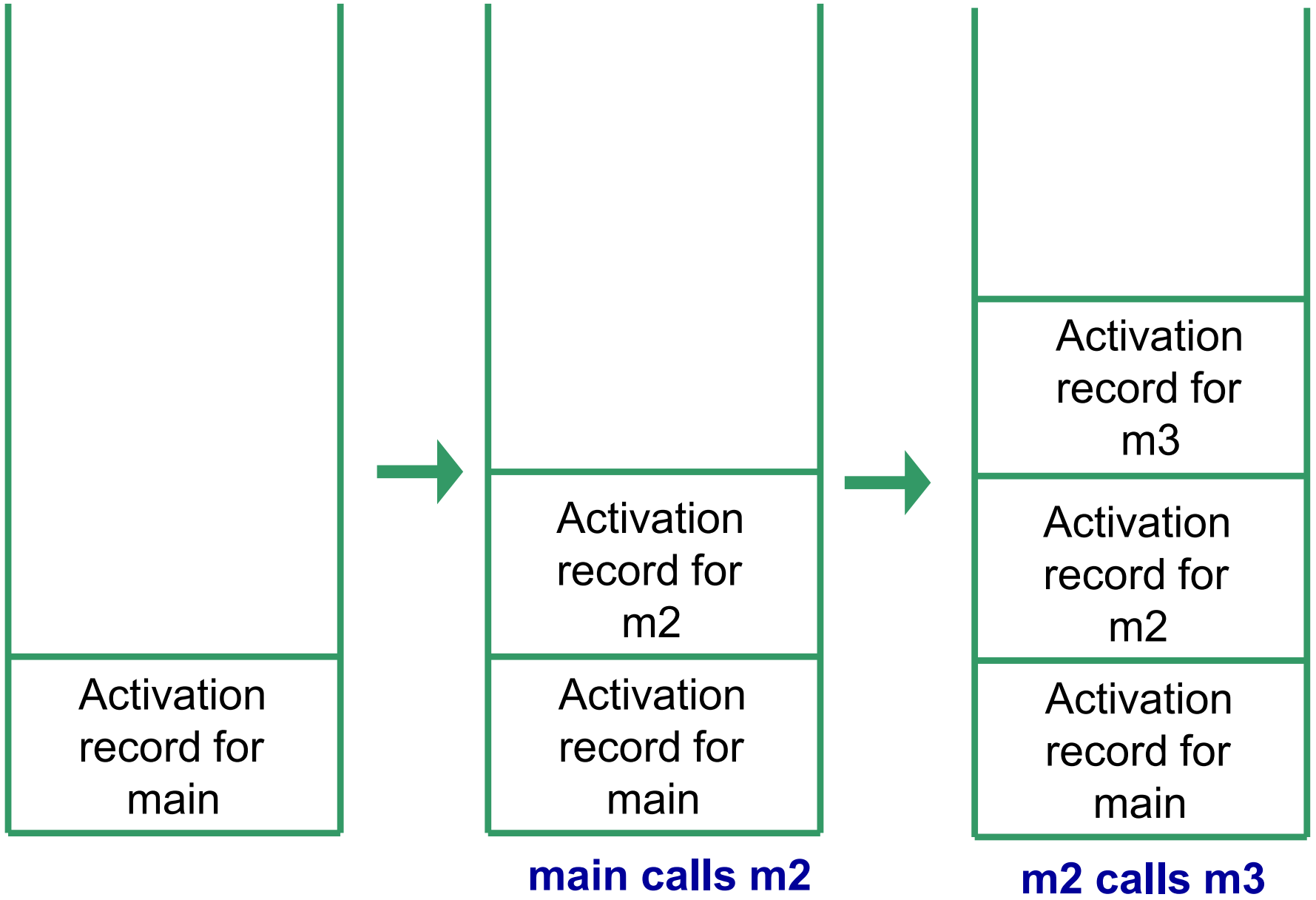


method println is pushed and popped right after main() is pushed into the stack.

```
public static void m2( ) {
    System.out.println("Starting m2");
    System.out.println("m2 calling m3");
    m3();
    System.out.println("m2 calling m4");
    m4();
    System.out.println("Leaving m2");
    return;
}

public static void m3( ) {
    System.out.println("Starting m3");
    System.out.println("Leaving m3");
    return;
}
```

After method is done, it is popped from the stack.

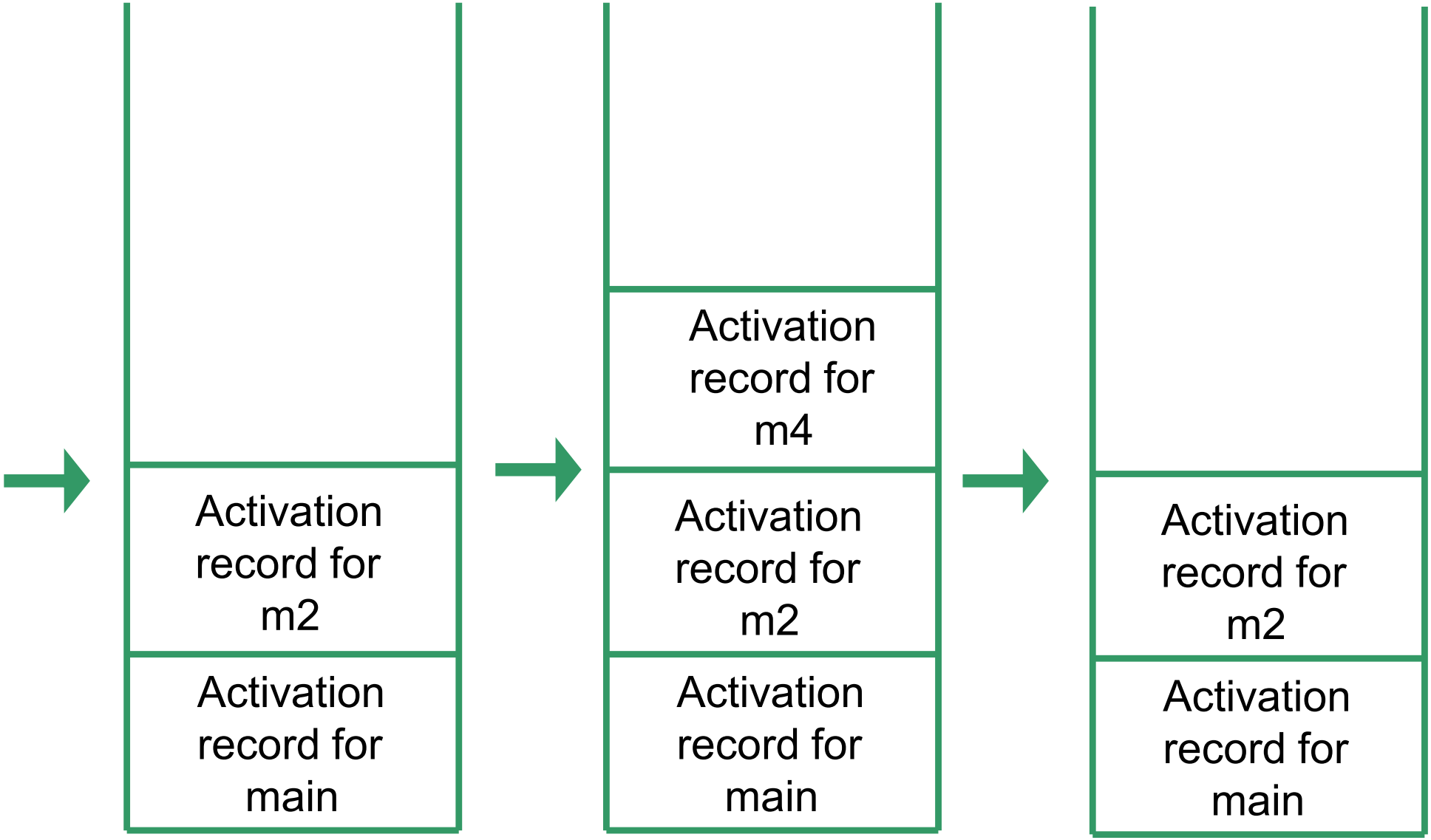
Execution Stack for a Typical Calling Sequence



main calls m2

m2 calls m3

Execution Stack for a Typical Calling Sequence



Execution Stack for a Typical Calling Sequence

- When the **main** method is invoked:
  - An activation record for main is created and pushed onto the execution stack
- When **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 Stack for a Typical Calling Sequence

- When **m2** next calls **m4**:
  - What happens next? push(m4)
  - What happens when **m4** terminates? pop(m4).
- What happens when **m2** terminates? pop(m2).
- What happens when **main** terminates?
  1. Its activation record is popped off and
  2. control returns to the operating system

Activation Records

- We will now look at some examples of what is in the activation record for a method
  - First for simple variables
  - Then for reference variables

Example: Activation Records – Simple Variables

```
public class CallFrameDemo1 {

    public static double square (double n) {
        double temp;
        temp = n * n;
        return temp;
    }

    public static void main (String args[ ]) {
        double x = 4.5;
        double y;
        y = square(x);
        System.out.println("Square of " + x + " is " + y);
    }

}
```

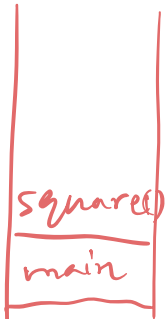
formal variable.

local

Both are static and will be stored in the heap.

address: main.

although the return type is void, it still return by going back to operating system



Activation Records – Example 1

Draw a picture of the activation records on the execution stack:

- What will be in the activation record for the main method?
  - Address to return to operating system
  - Variable **args**
  - Variable **x**
  - Variable **y**
- What will be in the activation record for the method square?
  - Address to return to main
  - Variable **n**
  - Variable **temp**
  - Return value

Discussion

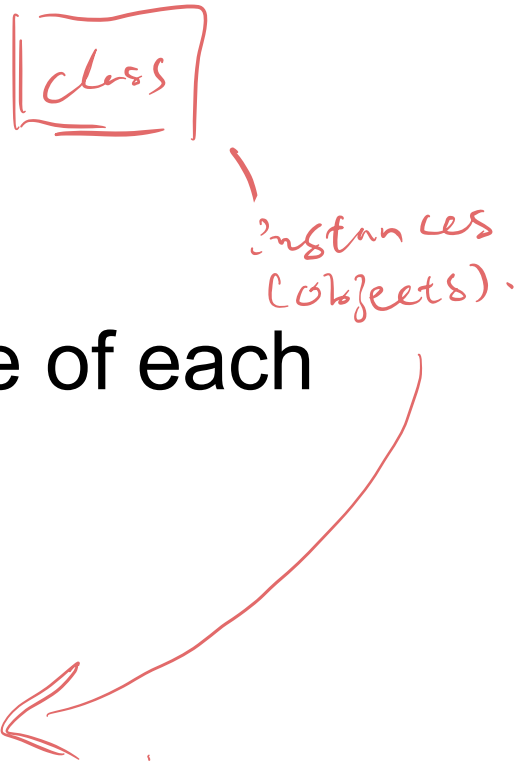
- There will be an activation record on the execution stack for *each* method called. So what other activation record(s) will be pushed onto the execution stack for our example?
- Which activation records will be on the execution stack at the same time?

println.

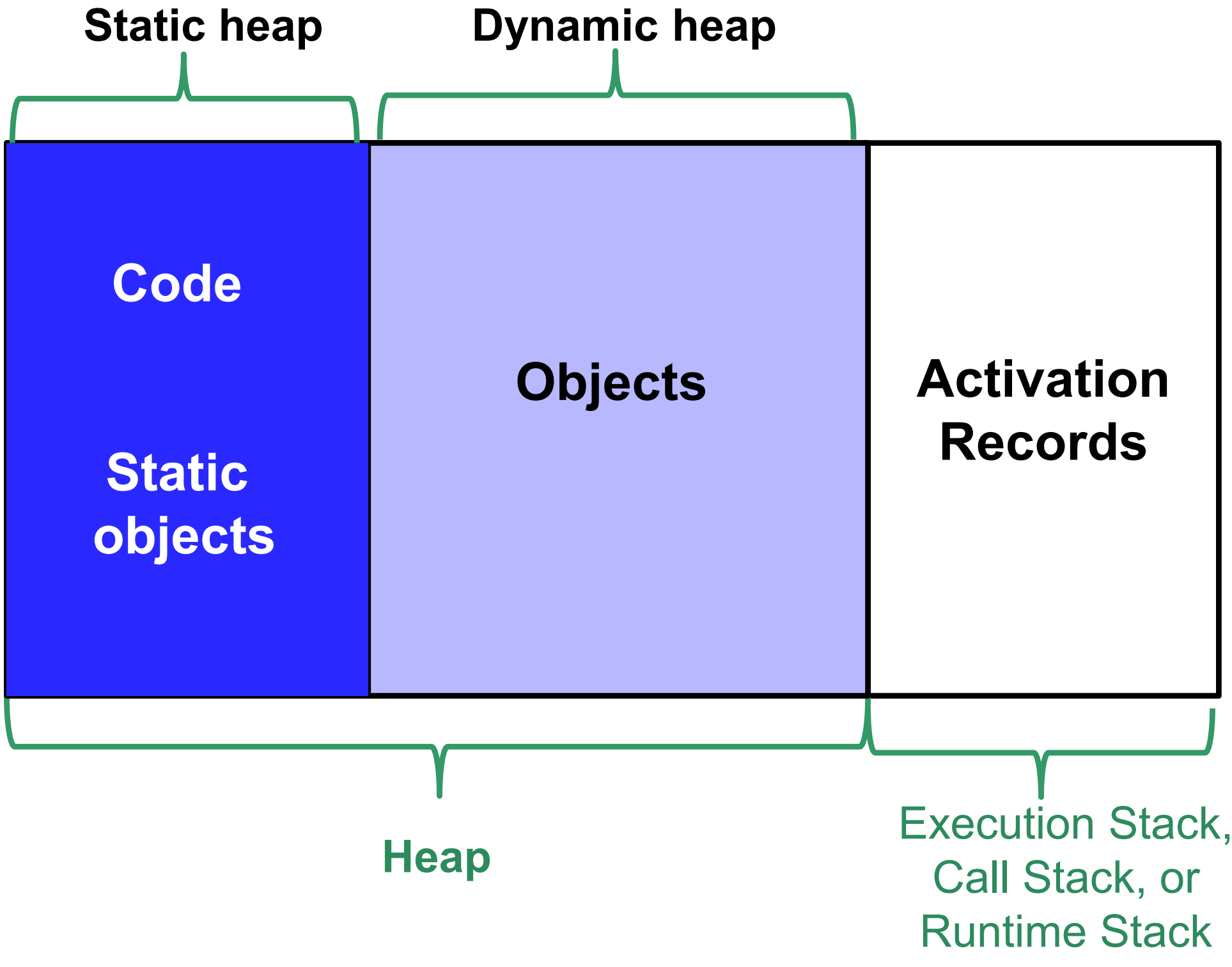
eg-1 main & square

eg-2. main & println.

Heap

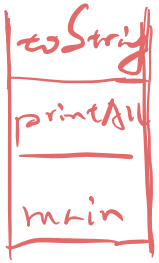


- ***Static space***:
  - contains ***one*** copy of the code of each class used in the program
  - also contains static objects
- ***Dynamic or Object space***:
  - Information that is stored for ***each*** object:
    - values of its instance variables
    - reference to its code


Static heap
Dynamic heap
Code
Static objects
Objects
Activation Records
Heap
Execution Stack, Call Stack, or Runtime Stack

```
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);
    }

}
```

return address : main



Activation Records – Example 2

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

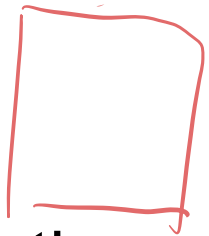
- Activation record for main
- Activation record for String constructor for str1 – then popped off
- Activation record for String constructor for str2 – then popped off
- Activation record for String constructor for str3 – then popped off
- Activation record for printAll
- Activation record for toString for str1 – then popped off
- Activation record for System.out.println – then popped off
- 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? => null
    - Value after return from String constructor?
  - Variable **str2**
  - Variable **str3**

heap object stored. =>

string "Sorlyl".



- What will be in the activation record for **printAll**?

System.out.

return to main

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 there is no longer a variable referencing it!
  - So, Java has automatic ***garbage collection***
    - It regularly identifies objects which no longer have a variable referencing them, and ***deallocates*** that memory