PKG 07: RECURSION

FRANK CARRANO:

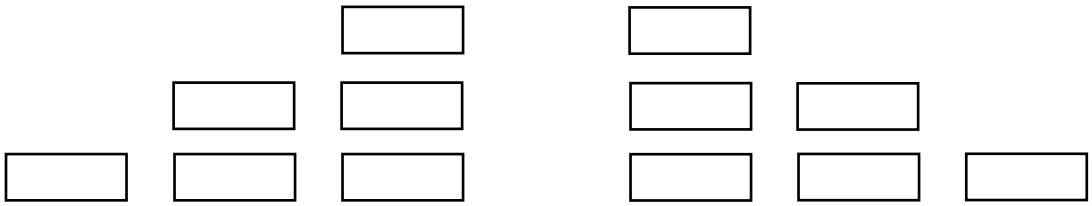
- Recursion is a problem-solving process that breaks a problem into identical but smaller problems.

- Base case: Known solution. Solution stated non-recursively. No recursive call.

- The stack of activation records: Each call to a method generates an activation record that captures the state of the method's execution and that is placed into the program stack. However, these methods need not be distinct. That is, the program stack enables a run-time environment to execute recursive methods. Each invocation of any method produces an activation record that is pushed onto the program stack. The activation record of a recursive method is not special in any way.

Does Java make multiple copies of recursive methods?

No. Java records the current state of the method's execution, including the values of its parameters and local variables as well as the location of the current instructions. Each record is called an activation record and provides a snapshot of a method's state during execution. The records are placed into the program stack. The stack organizes the records chronologically, so that the record of the currently executing method is on top. In this way, Java can suspend the execution of a recursive method and invoked it again with new argument values.



- A recursive method uses more memory than an iterative method, in general, because each recursive call generates an activation record.

```
public class CountDownIterative {
    public static void main(String[] args) {
        int num = 3;
        countDown(num);
    }

    public static void countDown(int x) {
        while (x >= 1) {
            System.out.println("x: " + x);
            x--;
        }
    }
}
```

```
/* OUTPUT
x: 3
x: 2
x: 1
*/
```

- A recursive method that makes many recursive calls will place many activation records in the program stack. Too many recursive calls can use all the memory available for the program stack, making it full. As a result, the error stack overflow occurs. Infinite recursion or large-size problems are the likely causes of this error.

- Tail recursion occurs when the last action of a recursive method is a recursive call. This recursive call performs a repetition that can be done by using iteration.

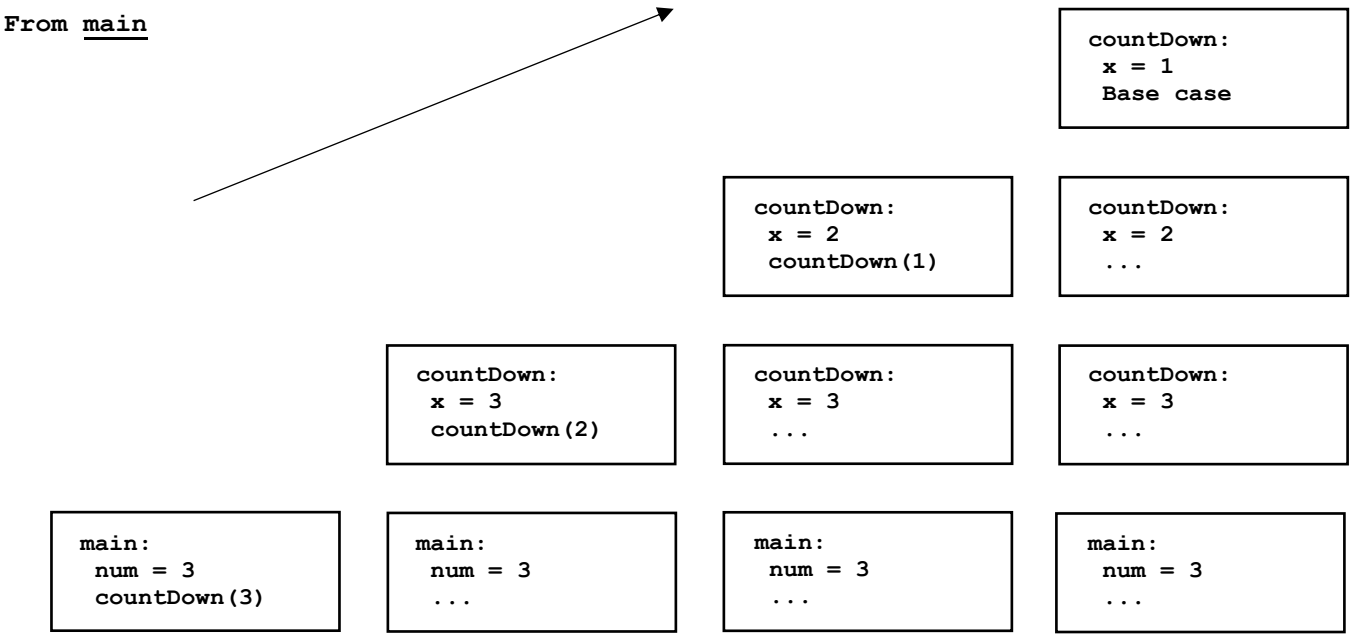
- Indirect recursion results when a method calls a method that calls a method, and so on until the first method is called again.

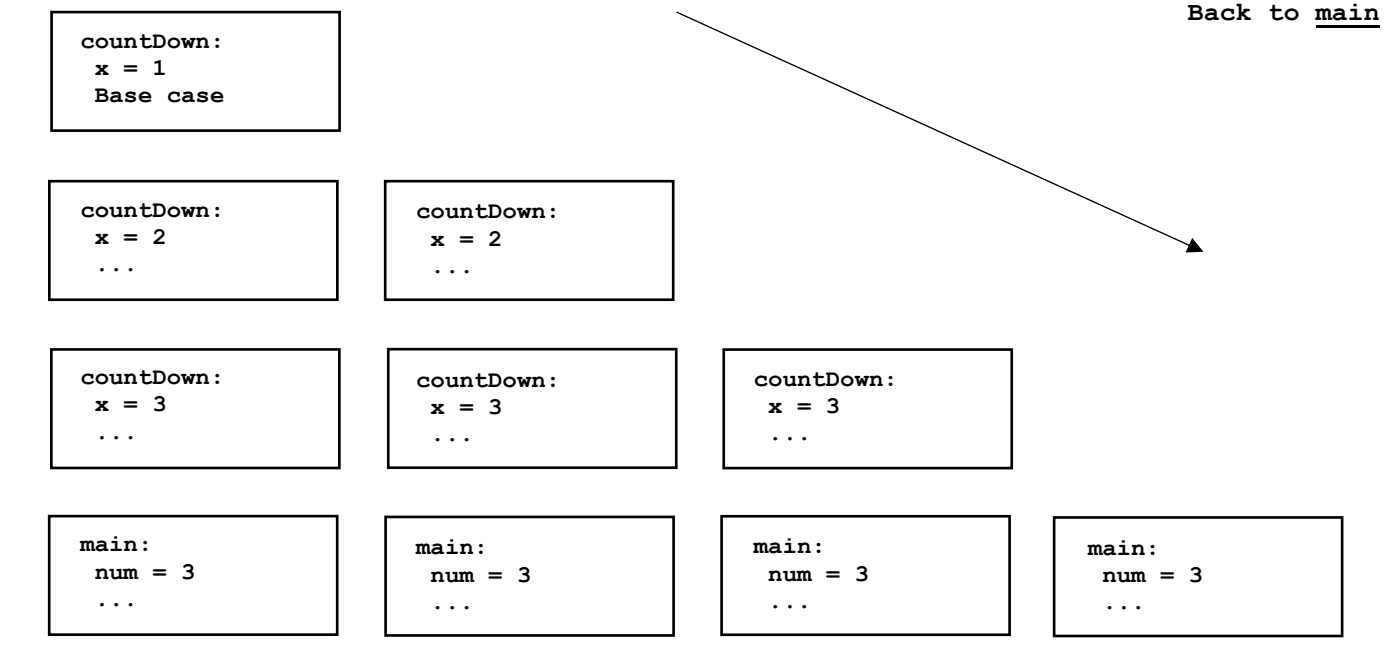
RECURSION

```
public class CountDownRecursive {
    public static void main(String[] args) {
        int num = 3;
        countDown(num);
    }

    public static void countDown(int x) {
        System.out.println("x: " + x);
        if (x > 1) {                          // Base case ?
            countDown(x - 1);                 // Recursive call
        }
    }
}
```

```
/* OUTPUT
x: 3
x: 2
x: 1
*/
```



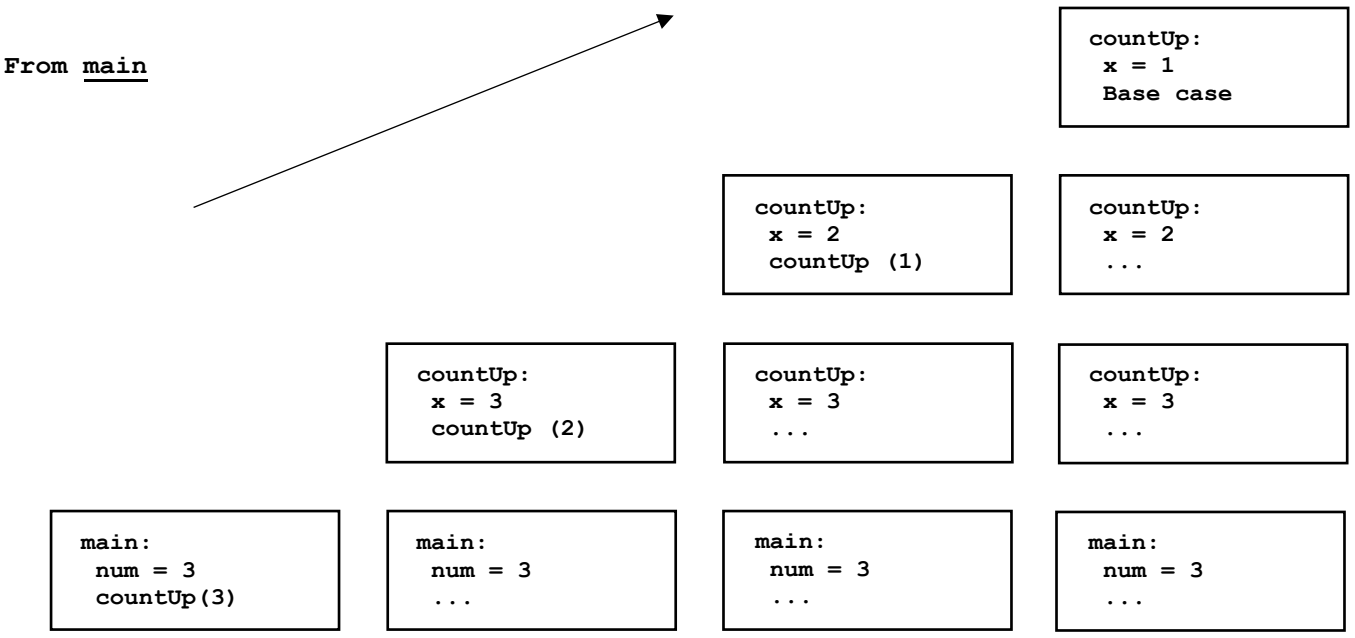


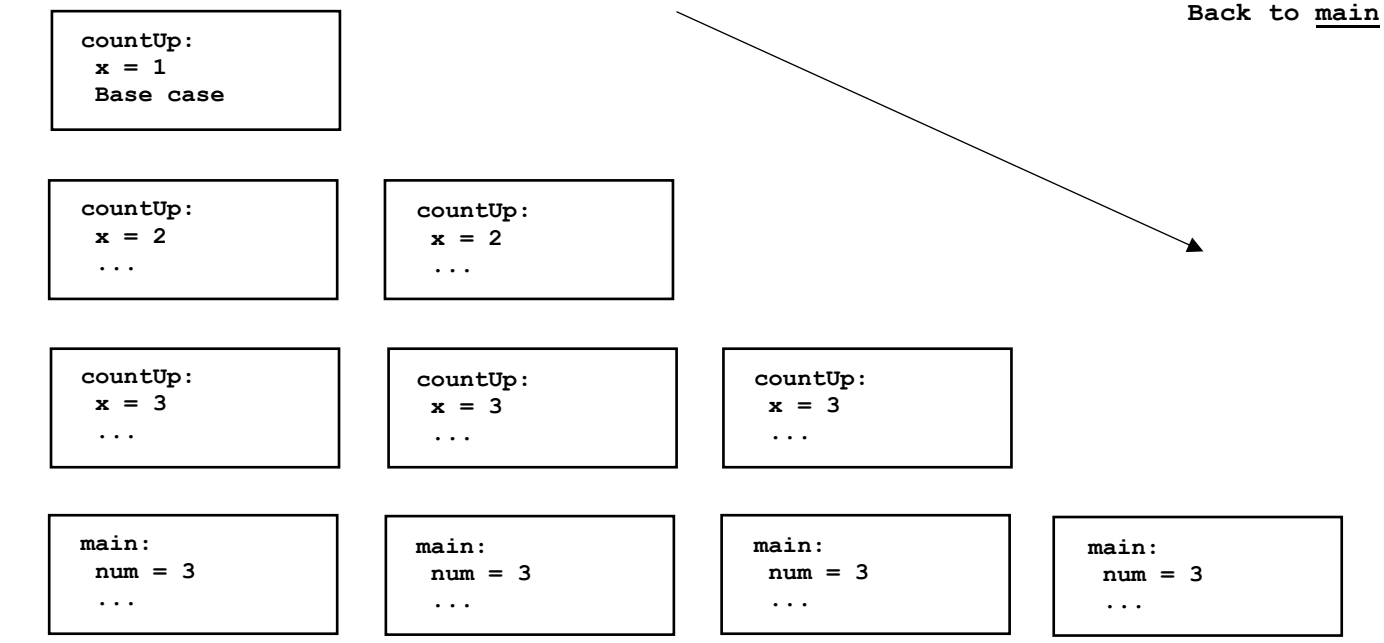
RECURSION

```
public class CountUpRecursive {
    public static void main(String[] args) {
        int num = 3;
        countUp(num);
    }

    public static void countUp(int x) {
        if (x == 1) {                        // Base case ?
            System.out.println("x: " + x);
        } else {
            countUp(x - 1);                  // Recursive call
            System.out.println("x: " + x);
        }
    }
}
```

```
/* OUTPUT
x: 1
x: 2
x: 3
*/
```



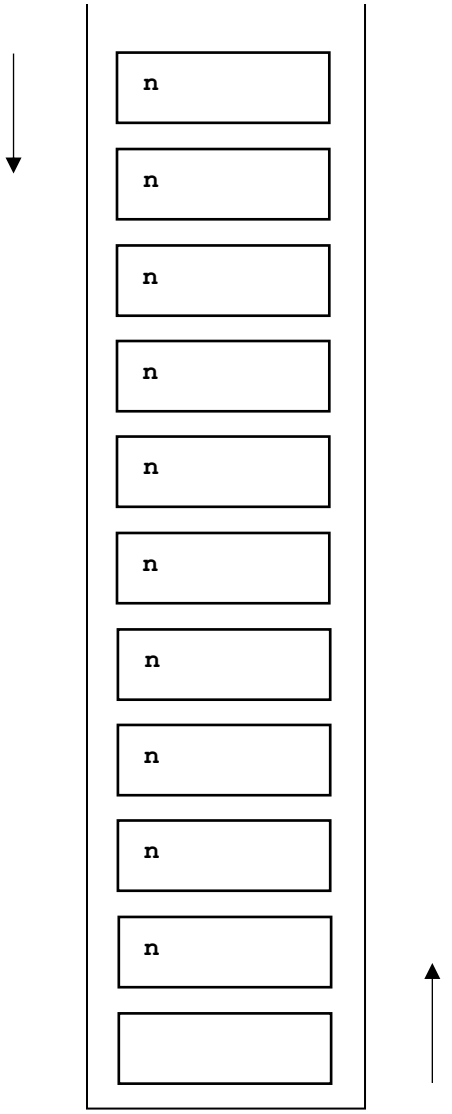


RECURSION, PRACTICE

```
public class SumOf {
    public static void main(String[] args) {
        int x = 30;
        System.out.println(sumOf(x));
    }

    public static int sumOf(int n) {
        int sum;
        if (n == 1) {                       // Base case ?
            sum = 1;
        } else {
            sum = sumOf(n - 1) + n;         // Recursive call
        }
        return sum;
    }
}
```

```
/* OUTPUT
465
*/
```



DESIGN GUIDELINES FOR SUCCESSFUL RECURSION

- The method must be given an input value, usually as an argument
- The method definition must contain logic that involves this input value and leads to different cases. Typically, such logic includes an if statement or a switch statement.
- One or more of these cases should provide a solution that does not require recursion. There are the base cases or stopping cases.
- One or more cases must include a recursive invocation of the method. These recursive invocations should in some sense take a step toward a base case by using "smaller" arguments or solving "smaller" versions of the task performed by the method.

```
public class DisplayArray {

    public static void main(String[] args) {
        int[] nums = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21};
        displayArrayFirst(nums, 0, 5);
        System.out.println("----------------------");
        displayArrayLast(nums, 0, 5);
        System.out.println("----------------------");
        displayArrayHalf(nums, 0, 5);

        System.out.println("----------------------");
        int reviewMid = 3;
        System.out.println("reviewMid: 3/2 = " + reviewMid / 2);
    }

    // The first entry is printed at the first method call.
    public static void displayArrayFirst(int array[], int first, int last) {
        System.out.println("First: " + first + " | Entry: " + array[first] + " ");
        if (first < last) {
            displayArrayFirst(array, first + 1, last);
        }
    }

    // The first entry is printed when the base case is reached.
    // The others are suspended in the stack.
    public static void displayArrayLast(int array[], int first, int last) {
        if (first <= last) {
            displayArrayLast(array, first, last - 1);
            System.out.println("Last: " + last + " | Entry: " + array[last] + " ");
        }
    }

    public static void displayArrayHalf(int array[], int first, int last) {
        if (first == last) { // Base case: array of 1 element
            System.out.println("First|Last: " + first + "|" + last + " \t| Entry: " +
array[first] + " ");
        } else { // If array has more than 1 element, keep dividing
            int mid = (first + last) / 2;
            System.out.print(" \t\t\t\t\t| Mid: " + mid + "\n");
            displayArrayHalf(array, first, mid);
            displayArrayHalf(array, mid + 1, last);
        }
    }
}
```

```
First: 0 | Entry: 1
First: 1 | Entry: 3
First: 2 | Entry: 5
First: 3 | Entry: 7
First: 4 | Entry: 9
First: 5 | Entry: 11
----------------------
Last: 0 | Entry: 1
Last: 1 | Entry: 3
Last: 2 | Entry: 5
Last: 3 | Entry: 7
Last: 4 | Entry: 9
Last: 5 | Entry: 11
----------------------
```

```
                                    | Mid: 2
                                    | Mid: 1
                                    | Mid: 0
First|Last: 0|0   | Entry: 1
First|Last: 1|1   | Entry: 3
First|Last: 2|2   | Entry: 5
                                    | Mid: 4
                                    | Mid: 3
First|Last: 3|3   | Entry: 7
First|Last: 4|4   | Entry: 9
First|Last: 5|5   | Entry: 11
----------------------
reviewMid: 3/2 = 1
```

```
317
318 public class FibonacciTimed {
319
320     public static void main(String args[]) {
321
322         int n = 40;
323
324         long start = System.nanoTime();
325         System.out.println("Recursive:");
326         for (int i = 0; i <= n; i++) {
327             System.out.print(fibonacciRecursive(i) + " ");
328         }
329         long end = System.nanoTime();
330         System.out.format("\nTime: %,d microseconds\n", (end - start) / 1000);
331
332         start = System.nanoTime();
333         System.out.println("\nIterative:");
334         for (int i = 0; i <= n; i++) {
335             System.out.print(fibonacciIterative(i) + " ");
336         }
337         end = System.nanoTime();
338         System.out.format("\nTime: %,d microseconds\n", (end - start) / 1000);
339
340         System.out.println("");
341     }
342
343     public static int fibonacciRecursive(int n) {
344         if (n <= 1) {
345             return n;
346         }
347         return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
348     }
349
350     public static int fibonacciIterative(int n) {
351
352         if (n <= 1) {
353             return n;
354         }
355         int fib = 1;
356         int prevFib = 1;
357
358         for (int i = 2; i < n; i++) {
359             int temp = fib;
360             fib += prevFib;
361             prevFib = temp;
362         }
363         return fib;
364     }
365
366 }
367
368 Recursive:
369 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
370 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
371 5702887 9227465 14930352 24157817 39088169 63245986 102334155
372 Time: 1,198,879 microseconds
373
374 Iterative:
375 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
376 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
377 5702887 9227465 14930352 24157817 39088169 63245986 102334155
378 Time: 786 microseconds
```