# Recursion & Memoization

## Recursion

# Recursively Defined Function

- When a function calls itself in it's definition, it is call recursive
- A recursive function is not circular (infinitely recursive) when:
  - For some values the function does not call itself (base cases)
  - Each time the function calls itself, the arguments must be closer to the base case arguments.
- Some functional programming languages do not define any looping constructs but rely solely on recursion to repeatedly call code.

## Example

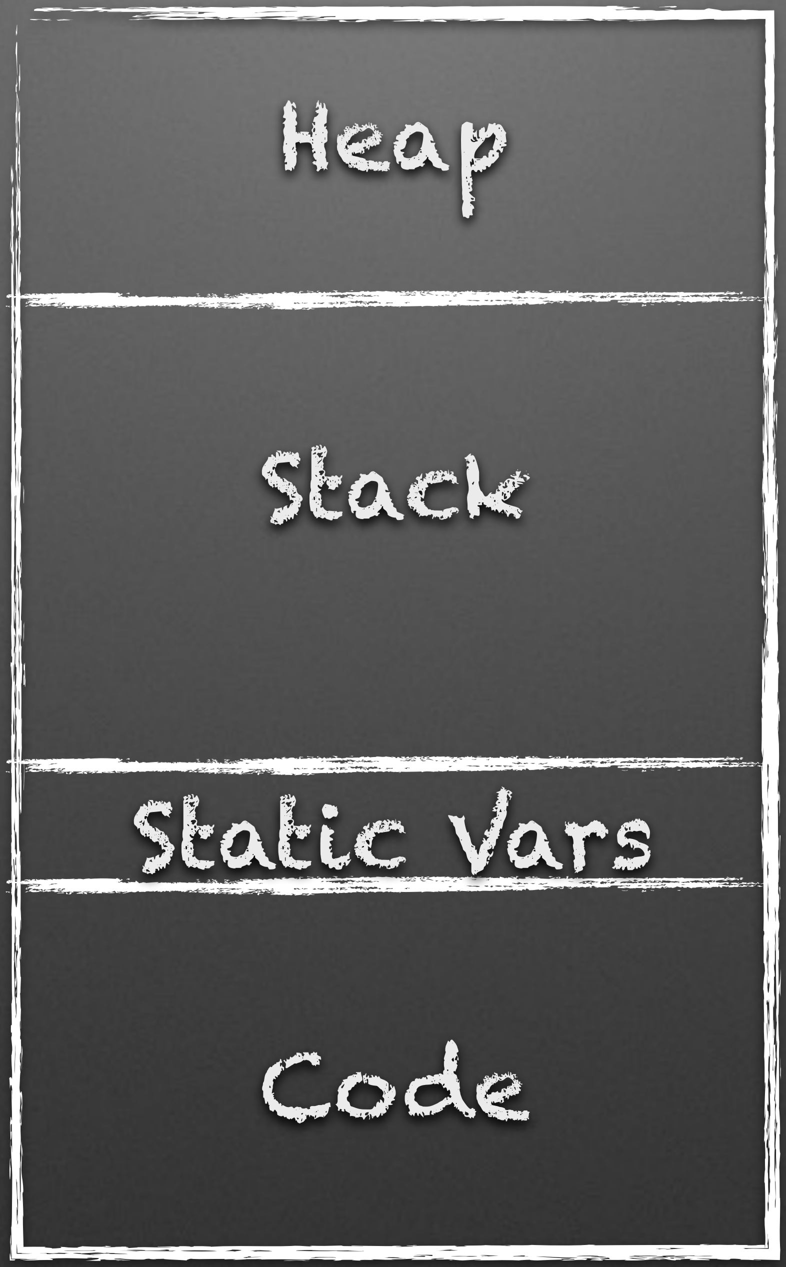
```
var factorial = function(n) {
    if(n == 0) {
        return 1
    } else {
        return n * factorial(n - 1);
    }
}
```

# Stack Overflow Risk

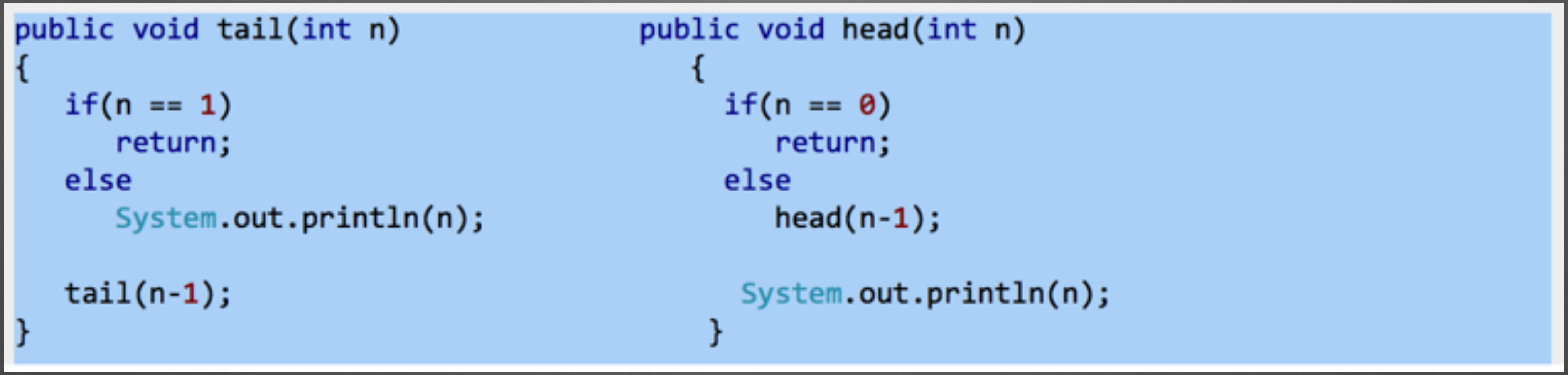
Each Function call creates a record in the stack, called a stack frame or activation record. It is deleted when the function exits.

If the space for number of stack frames required exceeds the space available to the stack, a stack overflow exception will be raised, and the program must exit.

In the case of recursion, when the recursion continues for too long, the call stack gets too deep, the stack space will be exhausted generating a stack overflow exception.



# Head & Tail Recursion



```
public void tail(int n)
{
   if(n == 1)
      return;
   else
      System.out.println(n);

   tail(n-1);
}
```

```
public void head(int n)
 {
   if(n == 0)
      return;
   else
      head(n-1);

    System.out.println(n);
 }
```

- In head recursion, the recursive call, comes before other processing in the function.
- In tail recursion, the function's other processing occurs before the recursive call.

# Tail Recursion Optimization

- Some compilers optimize tail recursive functions and will not create a new stack frame for each recursive call, instead re-using the existing stack frame.

- Note: the compiler/interpreter determines whether or not the recursive calls in a tail recursive function actually use an extra stack frame for each recursive call to the function.

# Recursion Analysis

- The most common strategy is to write the run time as a function of N: T(N). This indicates the time needed to process N items. By tracing carefully through the recursion, we can write down a recurrence relation for the algorithm. For example,

$$T(N) = T(N-1) + 1$$

- Then we repeat the recurrence

$$T(N) = [T(N-2)+1] +1 = T(N-2) + 2$$

$$T(N) = [T(N-3)+1] + 2 = T(N-3) + 3$$

- Look for a pattern:

$$T(N) = T(N-k) + k$$

- By tracing the pattern all the way to the base case T(1), we can determine the running time of the algorithm.

$$T(N) = O(N)$$

# Analyzing the Factorial

- Repeatedly plug in the recurrence

```
T(N) = T(N-1) + 1
     = T(N-2) + 1 + 1 = T(N-2) + 2 = T(N-3) + 1 + 2 = T(N-3) + 3
```

- So the pattern is:

```
T(N) = T(N-k) + k
```

- Letk=N.

```
T(N) = T(0) + N
```

- T(0) is the time to compute the base case factorial(0), which is just O(1).

```
T(N) = 1 + N = O(N)
```

- • The non-recursive version also runs in O(N) time.

# Recursive vs Iterative

- Recursive functions can be space inefficient. If your function recurses to a depth of n, it uses at least O(n) memory.
- ALL recursive algorithms can be implemented iteratively, although iterative code can be much more complex.
- Multiply recursive problems are inherently recursive, because of prior state they need to track. These algorithms can be implemented iteratively with the help of an explicit stack, but the programmer effort involved in managing the stack, and the complexity of the resulting program, arguably outweigh any advantages of the iterative solution.

# Recursive vs Iterative (cont.)

- In imperative programming, iteration is preferred, particularly for simple recursion, as it avoids the overhead of function calls and call stack management, but recursion is generally used for multiple recursion.

- By contrast, in functional languages recursion is preferred, with tail recursion optimization leading to little overhead, and sometimes explicit iteration is not available.

# Algorithm Design

Recursive algorithms build up a solution using sub-problems. Common techniques for breaking up a problem:

- Bottom-up: solve the problem for each of n=1, n=2 and n=3. Look for ways the current answer can be built from earlier answers. I.e. factorial
- Top-down: examine the problem and try to divide it into n sub-problems, taking care that the subsets dont overlap.
- Half and half: can the problem be solved when the data has been divided in half. I.e. binary search

## Memoization

## Definition

- Memoization is an optimization technique used speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
- Although related to caching, memoization refers to a specific case of this optimization. In the context of some logic programming languages, memoization is also known as tabling.
- Memoization is a way to lower a function's time cost in exchange for space cost; that is, memoized functions become optimized for speed in exchange for a higher use of computer memory space.

# Memoization Example

- In the following example, the Fibonacci function is rewritten to include memoization.

```
var fibonacci = (function() {
  var memo = {};

  function f(n) {
    var value;

    if (n in memo) {
      value = memo[n];
    } else {
      if (n === 0 || n === 1)
        value = n;
      else
        value = f(n - 1) + f(n - 2);

      memo[n] = value;
    }
    return value;
  }
  return f;
})();
```

# Resources

- Lecture 10: Recursion Analysis & Binary Search (http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec10.pdf)
- recursion-to-iteration series (http://blog.moertel.com/tags/recursion-to-iteration%20series.html)
- Memoizing recursive JavaScript functions (without mentioning fixed-point combinators or lambda calculus) (http://qntm.org/fib)