# Recursion & Memory Management — Detailed Guide

## 1. Base Case vs Recursive Case

```
A recursive function has two essential parts:

1. Base Case:
   - The condition where recursion stops.
   - Prevents infinite calls.
   Example: if(num == 0) return;

2. Recursive Case:
   - Function calls itself with a smaller input.
   Example: printNum(num - 1);

Together they allow the function to reduce the problem until reaching the base case.
```

## 2. Stack Pointer Movement

```
Stack Pointer (SP) moves DOWN when a new function is called:

   SP → printNum(3)
   SP → printNum(2)
   SP → printNum(1)
   SP → printNum(0)

During unwinding, SP moves UP as frames are removed:

   SP → return to printNum(1)
   SP → return to printNum(2)
   SP → return to printNum(3)
   SP → return to main()

Each movement corresponds to allocation and deallocation of stack frames.
```

## 3. Memory Footprint: Iteration vs Recursion

```
Recursion:
- Uses one stack frame per call.
- Memory usage = O(n).
- Can cause StackOverflowError for deep recursion.

Iteration:
- Uses one function frame.
- Memory usage = O(1).
- No risk of stack overflow.

Recursion trades memory for cleaner, more elegant logic.
```

## 4. Tail Recursion

```
A function is tail-recursive when the recursive call is the LAST statement.

Example (tail recursive):
   return tailFunc(n - 1);

Java does NOT optimize tail recursion, so:
- Tail recursion is NOT memory-efficient in Java.
- Still uses O(n) stack space.
Languages like Scala, Kotlin, Python (limited), and functional languages optimize tail recursion.
```

# 5. Why Stack Overflow Happens

```
StackOverflowError occurs when:
- Too many recursive calls build up.
- Stack memory limit is exceeded.

Example:
   infinite recursion:
      foo() calls foo()

Java reserves a limited-size stack.
Deep recursion consumes stack space quickly.
```