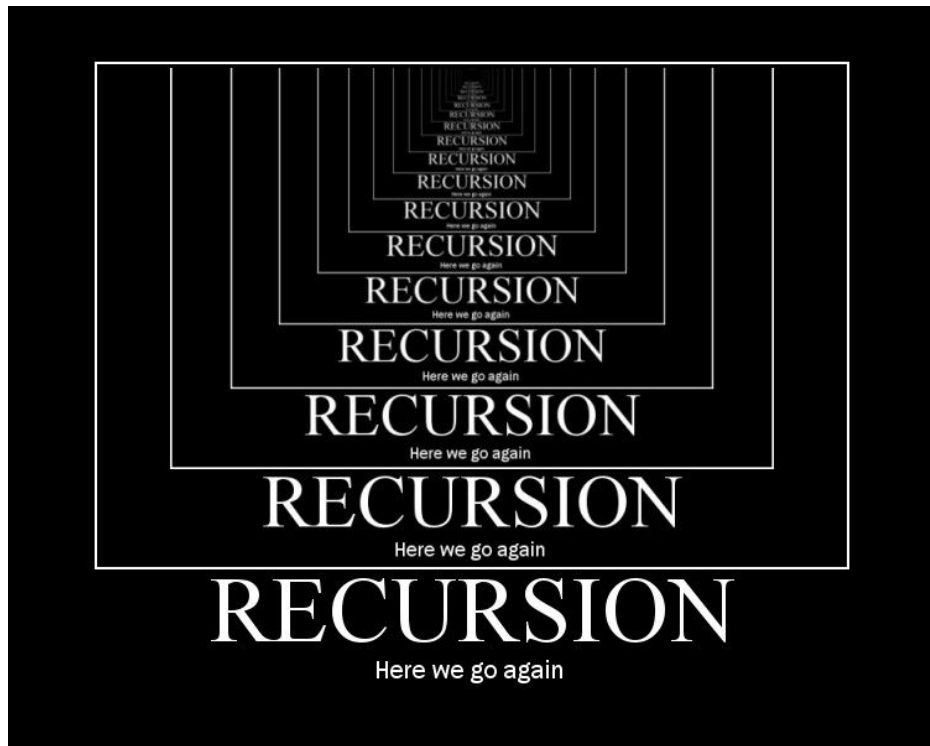# Recursion

Agenda

- Define recursion
- Identify recursive functions
- Explain Call Stack
- Example
- Exercises
- Exit Ticket

Recursion

Recursion is an iterative process where a method calls itself.

Recursion reduces a problem to smaller and similar sub-problems that can be solved more easily than the larger problem.

Google and recursion: Go to google and search recursion. What do you notice?

Example

```
public static void neverEnd()
{
  System.out.println("This is the method that never ends!");
  neverEnd();
}
```

This is an infinite recursion.

Is there a way to stop a recursive process?

Recursive Functions

Every recursive algorithm involves at least two cases:

- **Recursive case:** The function calls itself to solve a smaller problem then use that solution to solve the complete problem.

- **Base case**: The function recognizes the simplest situations and completes the computation without calling itself.

Both cases are important. Since working out the recursive case usually takes a lot of thinking, programmers often forget about the simple base case. When that happens, a recursive function will call itself repeatedly in an infinite recursion and the function will never complete.

The Call Stack

Recursive functions use something called "the call stack."

When a program calls a function, that function goes on top of the call stack (like stack of books)

Things are added one at the time.

To take something off, the top item is removed first.

Factorial Function Example

`fact(3)` is written as 3! and it is defined like this: 3! = 3 * 2 * 1. Here is a recursive function to calculate the factorial of a number:
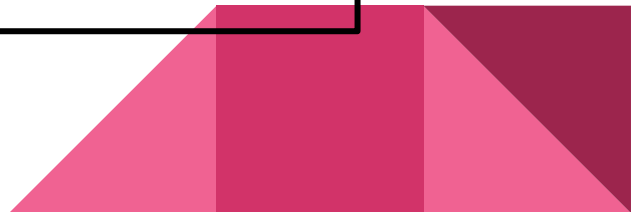
```
public static int fact(int x)
{
    if (x == 1)
        return 1;
    else
        return x * fact(x - 1);
}
```

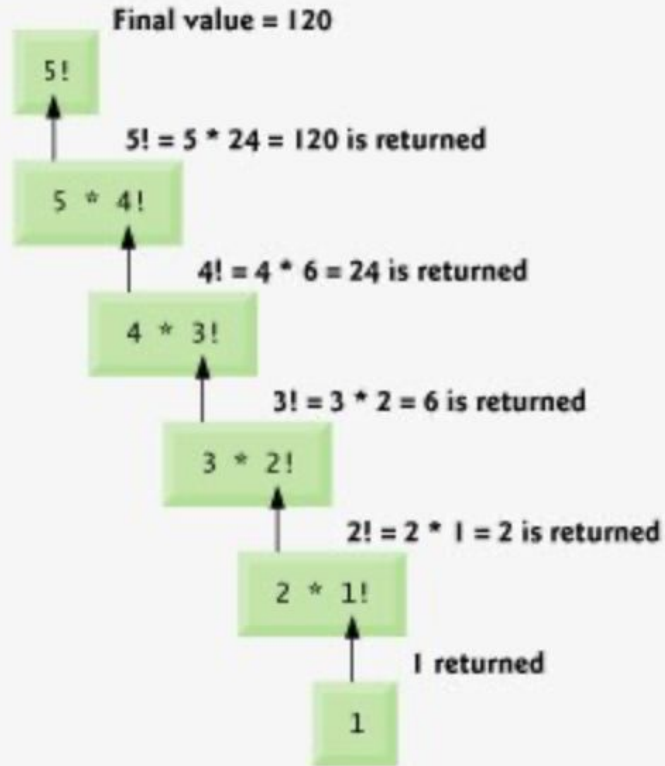Base Case

Recursive Case

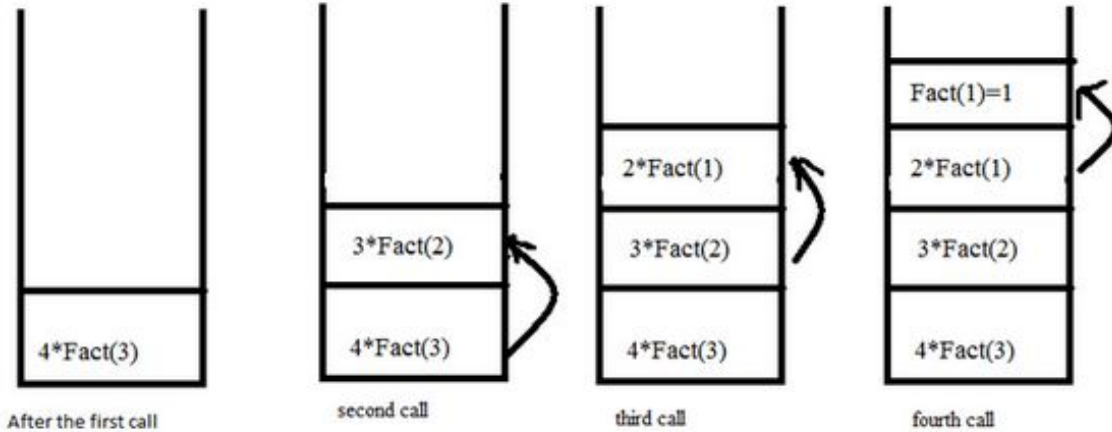Image credit: Adit Bhargava

Recursion - Factorial
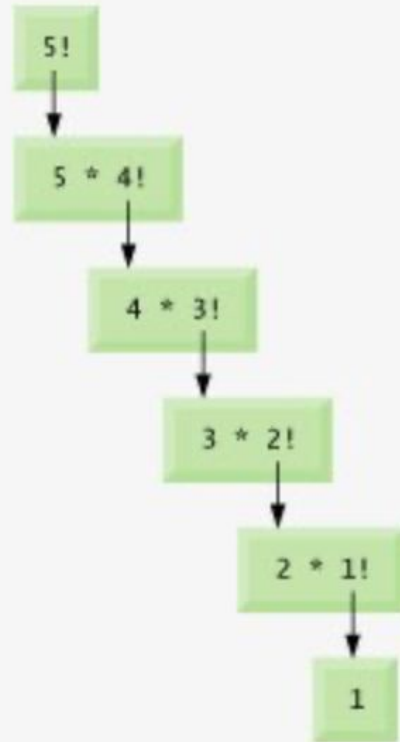
(a) Sequence of recursive calls.

(b) Values returned from each recursive call.

When function call happens previous variables gets stored in stack

			Fact(1)=1
		2*Fact(1)	2*Fact(1)
	3*Fact(2)	3*Fact(2)	3*Fact(2)
4*Fact(3)	4*Fact(3)	4*Fact(3)	4*Fact(3)
After the first call	second call	third call	fourth call

Returning values from base case to caller function

Fact(1)=1			
2*Fact(1)	2*Fact(1)		
3*Fact(2)	3*Fact(2)	3*Fact(2)	
4*Fact(3)	4*Fact(3)	4*Fact(3)	4*6=24

1

2

6

One more diagram fact(4)
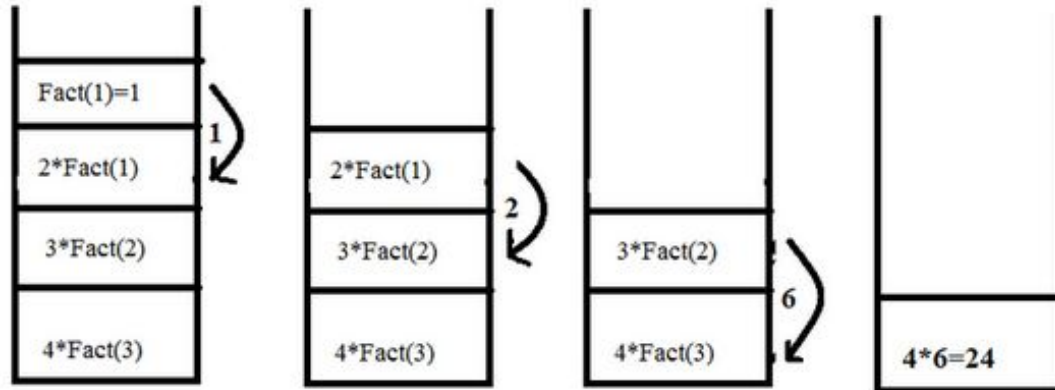
How to solve recursive problems?

Think of how you can break up a problem recursively which means how to split the problem in smaller sub-problems of the original one.

Follow these tips:

1. Your code must have a case for all valid inputs.
2. You must have a base case that makes not recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make forward progress towards the base case.

Exercise 1

Assume n is a positive number. Do not use remainder %. Instead think of the simplest two cases (base cases), and how can all other cases can get closer to the base case (recursive case).

Write the code and draw the call stack (upload an image of your call stack).

Save the code and an image of the call stack in the classwork directory:

.../APCSA_1/apcsa-assignments-YourUsername/classwork/28_recursion/Even.java

```
public static boolean isEven(int n){

}
```

Exercise 2

Given an array of integers. Find the sum of all the elements in the array.

Write the code and draw the call stack (upload an image of your call stack).

Save the code and an image of the call stack in the classwork directory:

.../APCSA_1/apcsa-assignments-YourUsername/classwork/28_recursion/SumArrayRecursion.java

public static int sumArray(int[] nums, int index){

}