# Fundamentals of Computer Science II

*04 - Recursion*

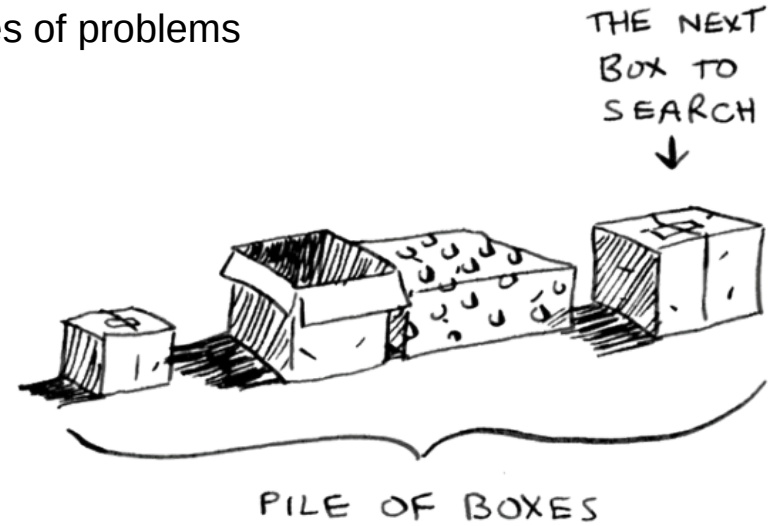
*Instructor: Varik Hoang*

# Objectives

- Recursion
- Why Learn Recursion?
- Recursion Cases
  - Base Case(s)
  - Recursive Case(s)
- Recursion & Iteration
- Infinite Recursion
- Public & Private Paired Methods

# Recursion

- The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.
- Writing methods that call themselves to solve problems recursively.
  - **No loop**: an equally powerful substitute for iteration (loops)
  - Particularly well-suited to solving certain types of problems



# Example - Bee Counting

```
How many bees are
in this line?
```

**function**

**Leader** **Bee A** **Bee B** **Bee C** **Bee D**

# Example - Bee Counting (cont.)

```
How many bees are
behind you?
```

**function**

**Leader**

**recursive**

**case 1 + ?**

**Bee A** **Bee B** **Bee C** **Bee D**

# Example - Bee Counting (cont.)

```
How many bees are
behind you?
```

**function**

**recursive case 1 + ?**

**recursive case 1 + ?**

**Leader**

**Bee A**

**Bee B**

**Bee C**

**Bee D**

# Example - Bee Counting (cont.)

```
How many bees are
behind you?
```

**function**

**recursive case 1 + ?**

**recursive case 1 + ?**

**recursive case 1 + ?**

**Leader**

**Bee A**

**Bee B**

**Bee C**

**Bee D**

# Example - Bee Counting (cont.)

```
How many bees are
behind you?
```

**function**

**Leader**

**recursive case 1 + ?**

**Bee A**

**recursive case 1 + ?**

**Bee B**

**recursive case 1 + ?**

**Bee C**

**recursive case 1 + ?**

**Bee D**

# Example - Bee Counting (cont.)

```
Oops! There is no
bee behind me
```

**function**

**Leader**

**recursive**

**case 1 + ?**

**Bee A**

**recursive**

**case 1 + ?**

**Bee B**

**recursive**

**case 1 + ?**

**Bee C**

**base**

**case 1 + 0**

**Bee D**

# Example - Bee Counting (cont.)

```
We got 1 bee
```

**function**

**recursive case 1 + ?**

**recursive case 1 + ?**

**recursive case 1 + 1**

**Leader**

**Bee A**

**Bee B**

**Bee C**

**Bee D**

# Example - Bee Counting (cont.)

```
We got 2 bees
```

**function**

**recursive case 1 + ?**

**recursive case 1 + 2**

**Leader**

**Bee A**

**Bee B**

**Bee C**

**Bee D**

# Example - Bee Counting (cont.)

```
We got 3 bees
```

**function**

**Leader**

**recursive**

**case 1 + 3**

**Bee A**

**Bee B**

**Bee C**

**Bee D**

# Example - Bee Counting (cont.)

```
We got 4 bees
```

**function**

**Leader** **Bee A** **Bee B** **Bee C** **Bee D**

# Example - Bee Counting (cont.)

```
We got total 4 bees in this line
Thank you all bees.
Please do not count me in this line
```

**function**

**Leader** **Bee A** **Bee B** **Bee C** **Bee D**

# Why Learn Recursion?

— A **different way** of thinking of problems (cultural experience)

— Can solve some kinds of problems better than iteration (**not** always)

— Leads to elegant, simplistic, short code (**when used well**)

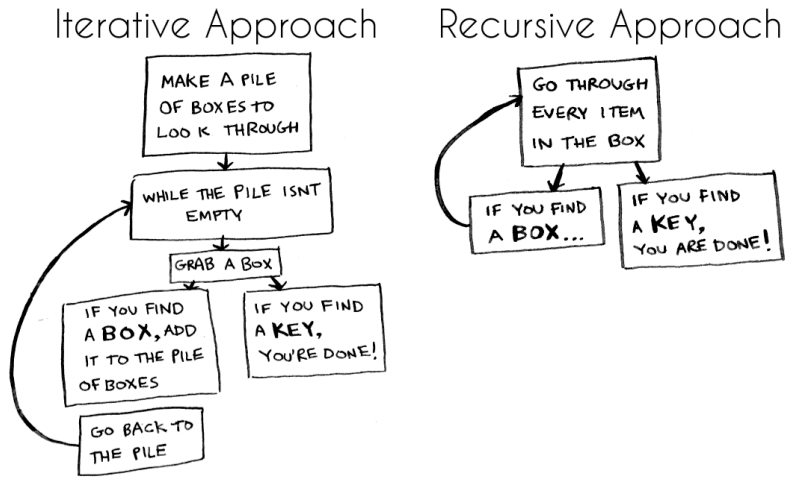
— Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

# Iterative to Recursion (1)

- We create a method called **writeStars**
  - It takes an integer parameter **n**
  - It produces a line of output with exactly **n** stars on it
- We can solve this problem with a simple for loop:

```
public static void writeStars(int n)
{
    for (int i = 1; i <= n; i++)
    {
        System.out.print("*");
    }

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

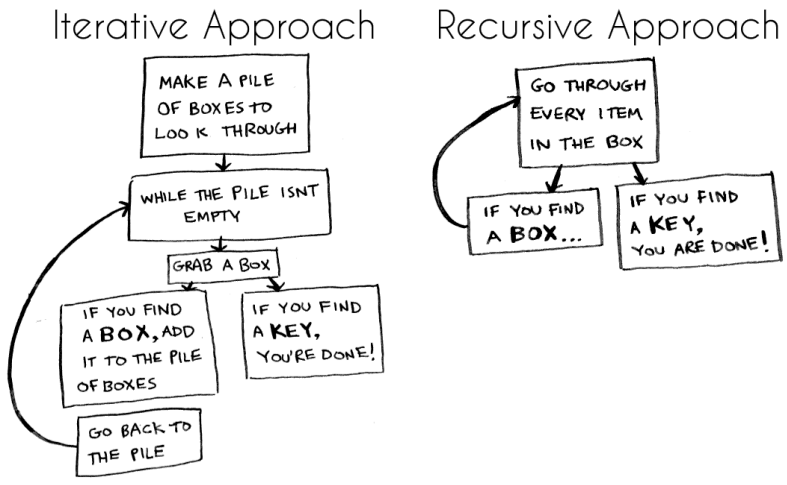


# Iterative to Recursion (2)

— We create recursion based on iteration solution

— Let's consider a case when **n** is **0**

```
public static void writeStars(int n)
{
    if (n == 0)
    {
        System.out.println();
    }
    else
    {
        ...
    }
}
```

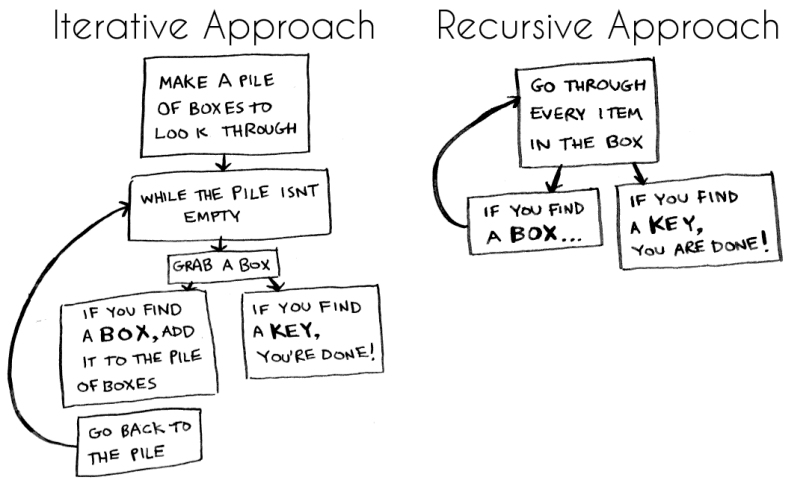



— The code in the **else** part will deal with more than zero stars

# Iterative to Recursion (3)

- We can do a small amount of work to get us closer to the solution
- Let's consider a case when **n** is **1**

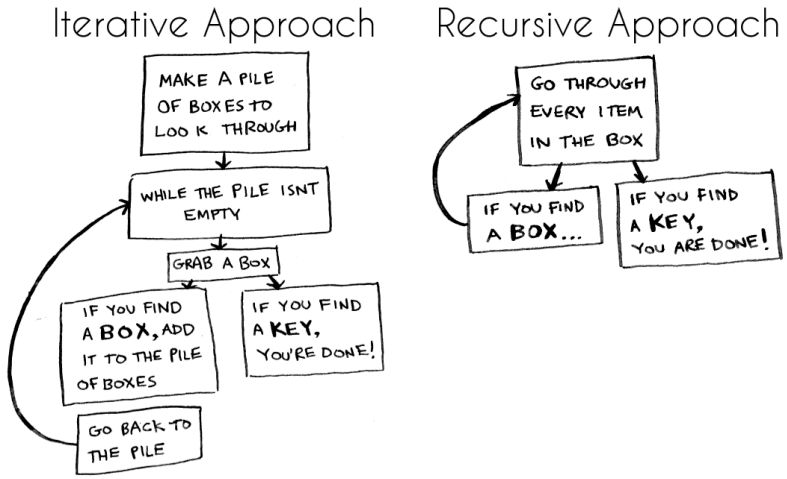
```
public static void writeStars(int n)
{
    if (n == 0)
    {
        System.out.println();
    }
    else
    {
        System.out.print("*");
        // what is left to do?
    }
}
```



# Iterative to Recursion (4)

- We can call the **writeStars** method itself to complete the line of output
- Let's call **writeStars** with the value of (**n** - **1**)

```
public static void writeStars(int n)
{
    if (n == 0)
    {
        System.out.println();
    }
    else
    {
        System.out.print("*");
        writeStars(n - 1);
    }
}
```



# Iterative to Recursion (5)

— What happens when we call the method and request a line of three stars

```
writeStars(3); // call up
```

— Call up from the **writeStars** method

```
System.out.print("*");
writeStars(2); // call up again
```

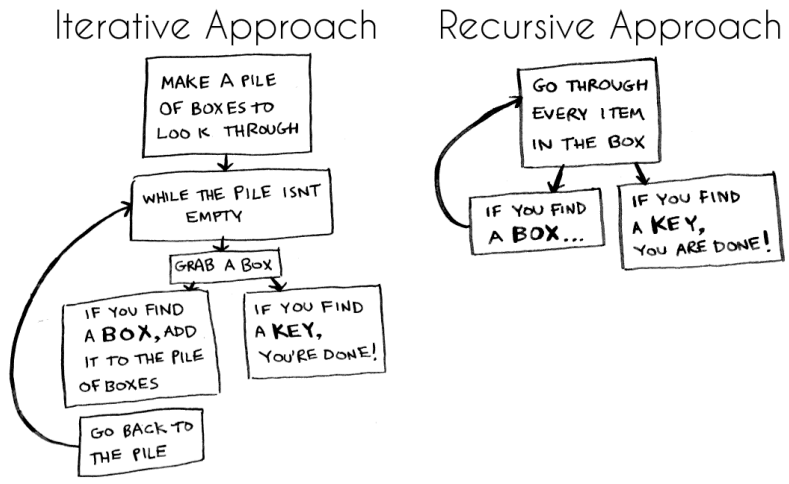
— Call up from the **writeStars** method

```
System.out.print("*");
System.out.print("*");
writeStars(1); // continue until the key found
```

— Call up from the **writeStars** method until **n** is **0** (key found)

```
System.out.print("*");
System.out.print("*");
System.out.print("*");
writeStars(0); // continue until the key found
```

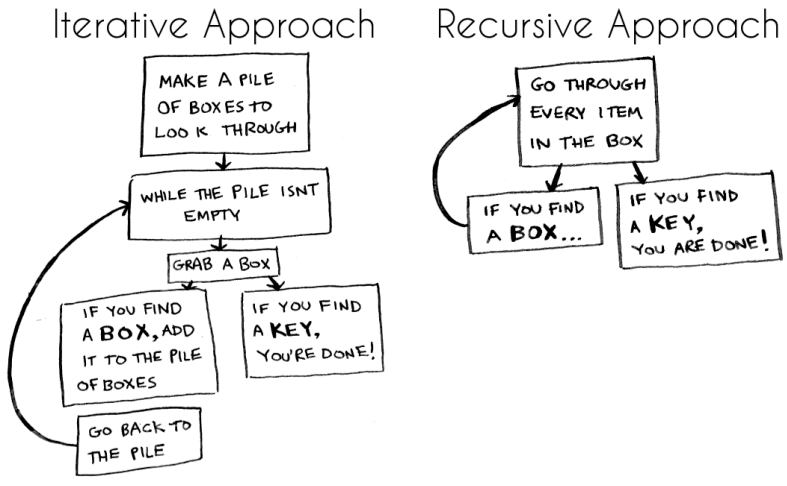
```
System.out.print("*");
System.out.print("*");
System.out.print("*");
System.out.println(); // found the key
```



# Iterative to Recursion (6)

— Here is a trace of the calls that would be made to print the line:

```
writeStars(3); // n > 0, execute else
├─System.out.print("*");
└─writeStars(2); // n > 0, execute else
  ├─System.out.print("*");
  └─writeStars(1); // n > 0, execute else
    ├─System.out.print("*");
    └─writeStars(0); // n == 0, execute if
      └─System.out.println();
```

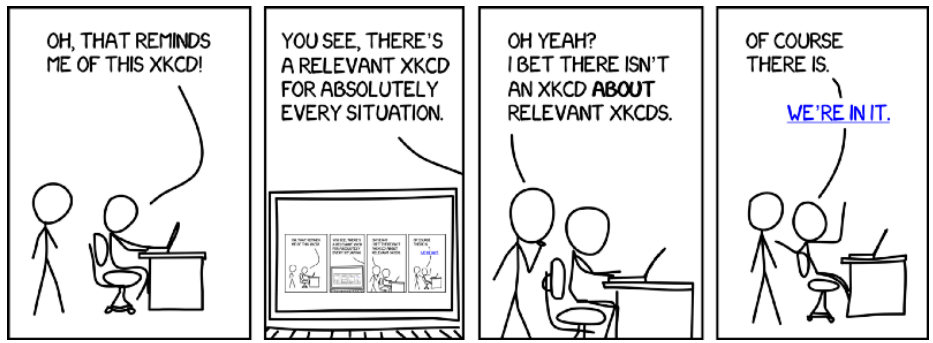


# Structure of Recursion

Every recursive algorithm involves *at least* 2 cases:

- **Base case:** a simple occurrence that can be answered directly
- **Recursive case:** a more complex occurrence cannot be directly answered but smaller occurrences of the same problem

```
public static void writeStars(int n)
{
    if (n == 0)
    {
        // base case
        System.out.println();
    }
    else
    {
        // recursive case
        System.out.print("*");
        writeStars(n - 1);
    }
}
```




https://www.freecodecamp.org/news/recursion-demystified-99a2105cb871/

# Common Mistakes

— A recursive case that did not make any change on **n**:

```
public static void writeStars(int n)
{
    writeStars(n);
}
```

— A recursive case that made changes on n but **no base case**:

```
public static void writeStars(int n)
{
    System.out.print("*");
    writeStars(n - 1);
}
```

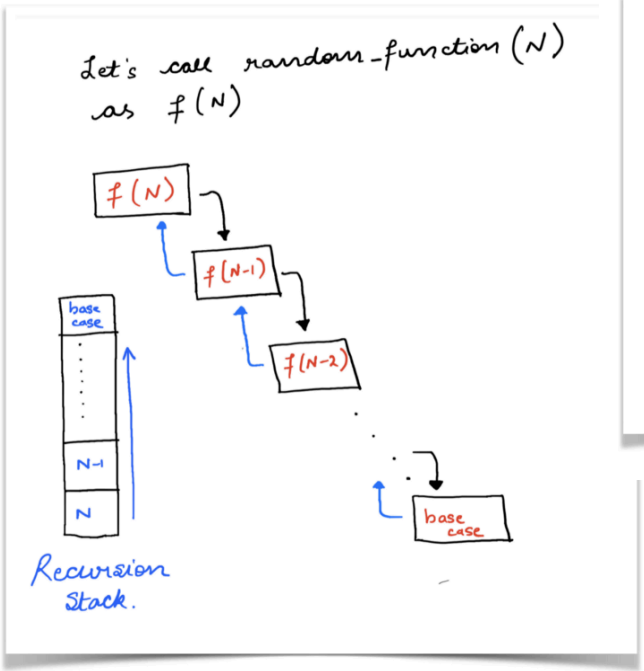
— A recursive solution never finishes executing is called **infinite recursion**

# Recursion & Iteration

- Every recursive solution **has a corresponding iterative solution**
  - For example, N! can be calculated with a loop iteratively or recursively
- Recursion **has the overhead** of multiple method invocations
- However, for some problems recursive solutions are often more simple and elegant than iterative solutions
- How is recursion (recursive) different from iteration (iterative)
  - Iteration: repeat using a loop.
  - Recursion: repeat using a method that calls itself.

# Example - Factorial

We are going to illustrate an example of factorial function.




$f(N) = N * f(N-1)$

$f(N-1) = N-1 * f(N-2)$

$f(N-2) = N-2 * f(N-3)$

$\vdots$

$f(2) = 2 * f(1)$

$f(1) = 1$ Base case of recursion

$\rightarrow f(2) = 2 * f(1)$
$= 2 * 1$

$\rightarrow f(3) = 3 * f(2)$
$= 3 * 2 * 1$

$\rightarrow f(4) = 4 * f(3)$
$= 4 * 3 * 2 * 1$

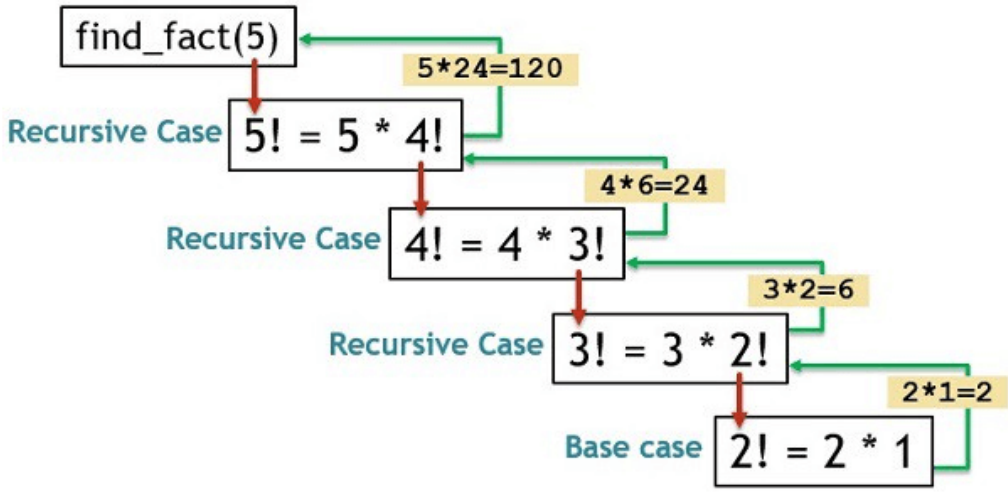
$\rightarrow f(5) = 5 * f(4)$
$= 5 * 4 * 3 * 2 * 1$

Answer returned by smaller subproblem.

# Example - Factorial (cont.)

```
public int find_fact(int n)
{
    // base case to ensure that the value
    // of n will not be less than one
    if (n < 1)
        return 1;

    // recursive case to calculate n * (n-1)!
    return n * find_fact(n - 1);
}
```



# Infinite Recursion

```
public int find_fact(int n)
{
    // base case to ensure that the value
    // of n will not be less than one
    if (n > 1)    // The change keeps the value
        return 1; // decreasing to negative -∞

    // recursive case to calculate n * (n-1)!
    return n * find_fact(n - 1);
}
```

Everyone who uses recursion to write programs eventually accidentally writes a solution that leads to infinite recursion.

# Helper Methods

- Often the parameters we need for our recursion do not match those the client will want to pass.
- In these case, we instead write a pair of methods:
  - A **public**, non-recursive one with the parameters the client wants
  - A **private**, recursive one with the parameters we really need. This will be called from the public, non-recursive method as a helper.

***Note: this is still a recursive solution.***

# Helper Methods - Example

```
// this method is for client use
public int find_fact(int n)
{
    // validate the input from client
    if (n < 0)
        throw new IllegalArgumentException();

    // call the private method
    return find_fact_r(n);
}

// this method is only for internal use
private int find_fact_r(int n)
{
    // base case to ensure that the value
    // of n will not be less than one
    if (n < 1)
        return 1;

    // recursive case to calculate n * (n-1)!
    return n * find_fact_r(n - 1);
}
```

- The **public** method:
  - ... is for the input validation
  - ... is for the client access
- The **private** method:
  - ... is for the actual flow of the algorithm
  - ... is *not* for the client access

# Questions & Answer


Thank You!