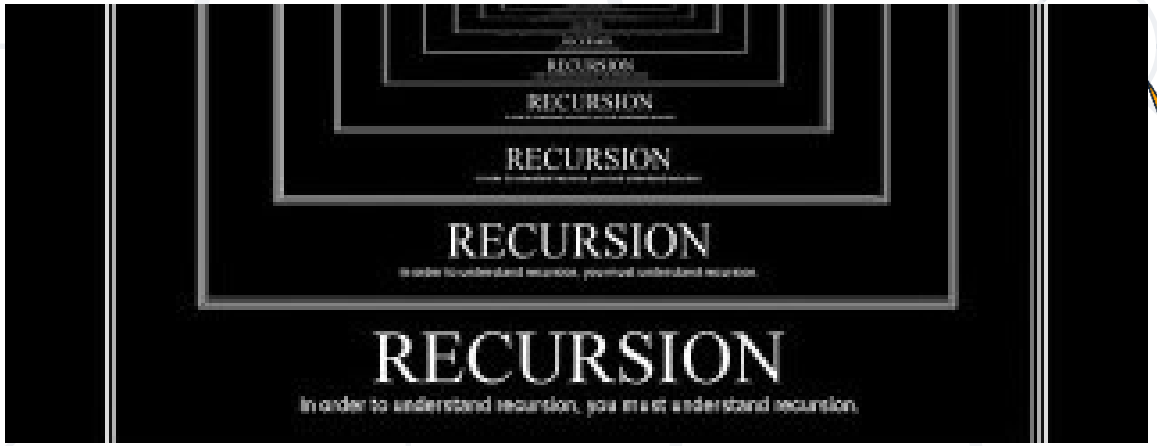
Recursion

Using Recursion, Recursion vs Iteration



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://about.softuni.bg/

Table of Contents

What is Recursion?

What is Recursion?

- A function or a method that **calls itself** one or more times until a specified **condition** is **met**
  - After the recursive call the rest code is processed **from** the **last** one called **to** the **first**

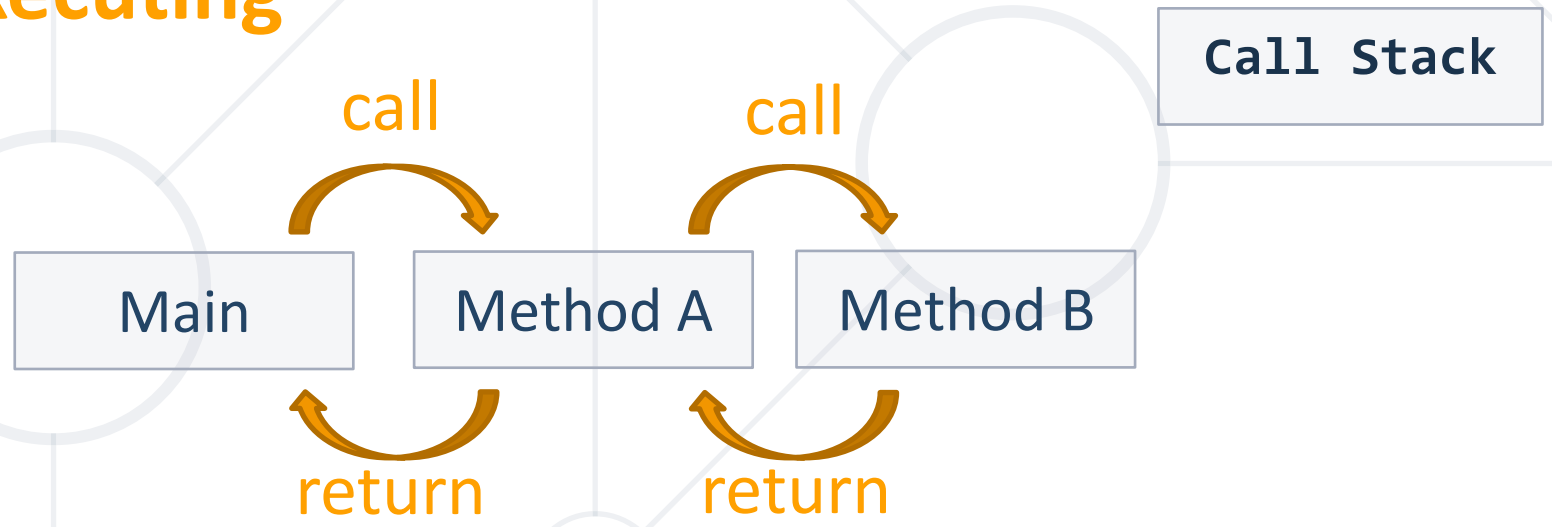
```
int f(int n)
{
  if (n > 1)
    return n * f(n-1);
}
```

What is Recursion?

- **Recursion** == method solving problems
  - Where the solution depends on the solutions of smaller instances of the same problem
- A common **computer programing approach** is to:
  - **Divide** a problem into **sub-problems** of the same type as the original
  - **Solve** those sub-problems
  - **Combine** the **results**

Call Stack

- "The stack" is a small **fixed-size** chunk of memory (e. g. 1MB)
- Keeps track of **the point** to which each active subroutine should **return control** when it **finishes executing**



Recursion: Other Definition

- Problem solving technique (in CS)
  - Involves a **function calling itself**
  - The function should have a **base case**
  - **Each step** of the recursion should **move towards** the **base case**

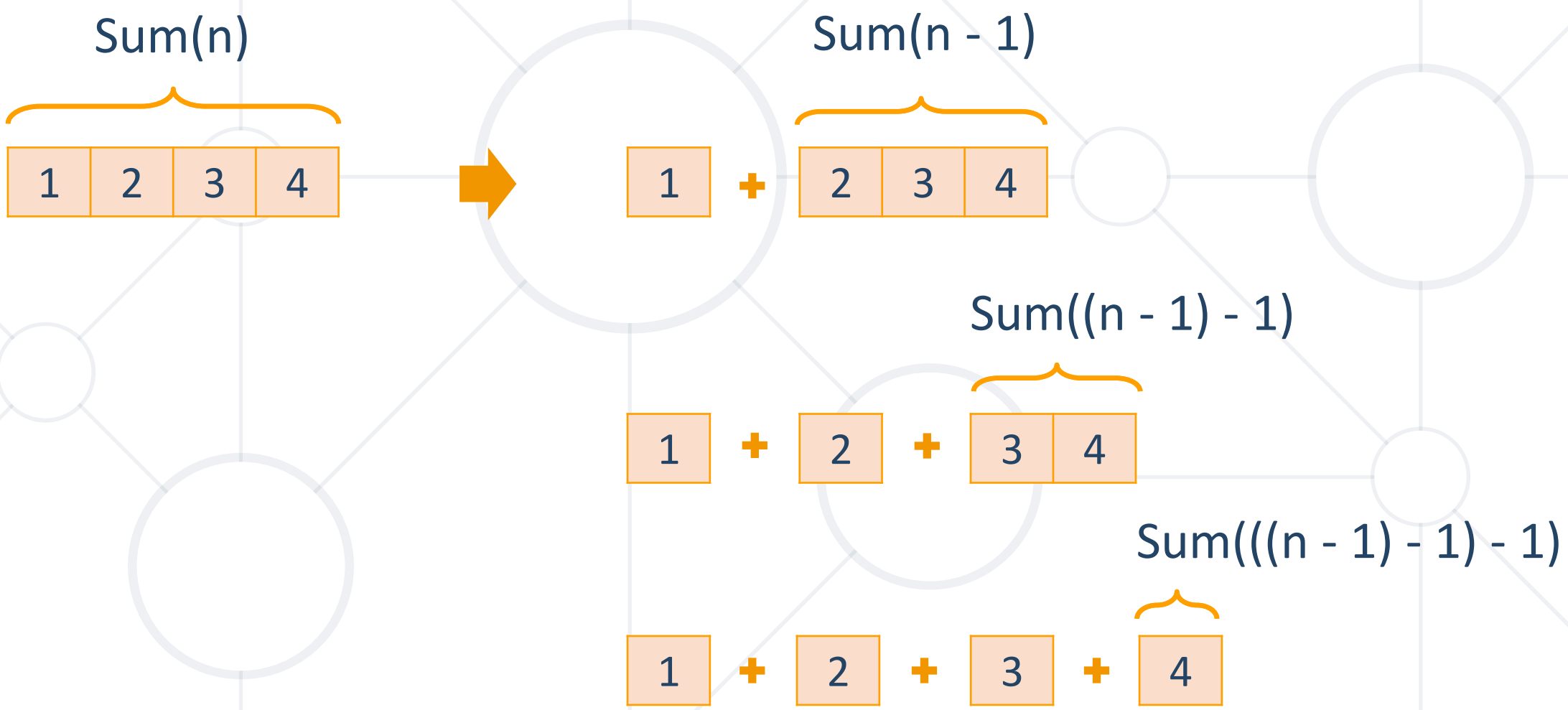
Sum(array)

1	2	3	4	5

array[0] + Sum(sub-array)

1	+	2	3	4	5

Array Sum – Example



Live Exercises

Problem: Array Sum

- Create a **recursive method** that
  - Finds the sum of all numbers stored in an `int[] array`
  - Read the numbers from the console

`1 2 3 4`	→	`10`
`-1 0 1`	→	`0`

Solution: Array Sum

```
static int Sum(int[] array, int index)
{
    if (index == array.Length - 1)
    {
        return array[index];
    }

    return array[index] + Sum(array, index + 1);
}
```

Base case

Recursive call

Check your solution here: https://judge.softuni.org/Contests/Practice/Index/3185#0

Problem: Recursive Factorial

- Create a **recursive method** that calculates **n!**
  - Read n from the console

5	→	120
10	→	3628800

Recursive Factorial – Example

- Recursive definition of **n!** (n factorial):

```
n! = n * (n-1)! for n > 0
0! = 1
```

3!	=	3 * 2!
2!	=	2 * 1!
1!	=	1 * 0!
0!	=	1

Solution: Recursive Factorial

```
static long GetFactorial(int num)
{
    if (num == 0)
     return 1;

  return num * GetFactorial(num - 1);
 }
```

Base case

Recursive call

Check your solution here: https://judge.softuni.org/Contests/Practice/Index/3185#2

Direct and Indirect Recursion

- **Direct** recursion
  - A method directly calls itself
- **Indirect** recursion
  - Method **A** calls **B**, method **B** calls **A**
  - Or even **A** → **B** → **C** → **A**

Recursion Pre-Actions and Post-Actions

- Recursive methods have **three** parts:
  - **Pre-actions** (before calling the recursion)
  - **Recursive calls** (step-in)
  - **Post-actions** (after returning from recursion)

```
static void Recursion()
{
    // Pre-actions
  Recursion();
  // Post-actions
  }
```

Problem: Recursive Drawing

- Create a **recursive method** that draws the following figure

5

```
*****
****
***
**
*
#
##
###
####
#####
```

Check your solution here: https://judge.softuni.org/Contests/Practice/Index/3185#3

Pre-Actions and Post-Actions – Example

```
static void PrintFigure(int n)
{
    if (n == 0)
      return;

    // TODO: Pre-action: print n asterisks
    PrintFigure(n - 1);
    // TODO: Post-action: print n hashtags
}
```

R || I

When to Use and When to Avoid Recursion?

Performance: Recursion vs. Iteration

- Recursive calls are **slower**
- Parameters and return values **travel** through the stack
- Good for **branching** problems

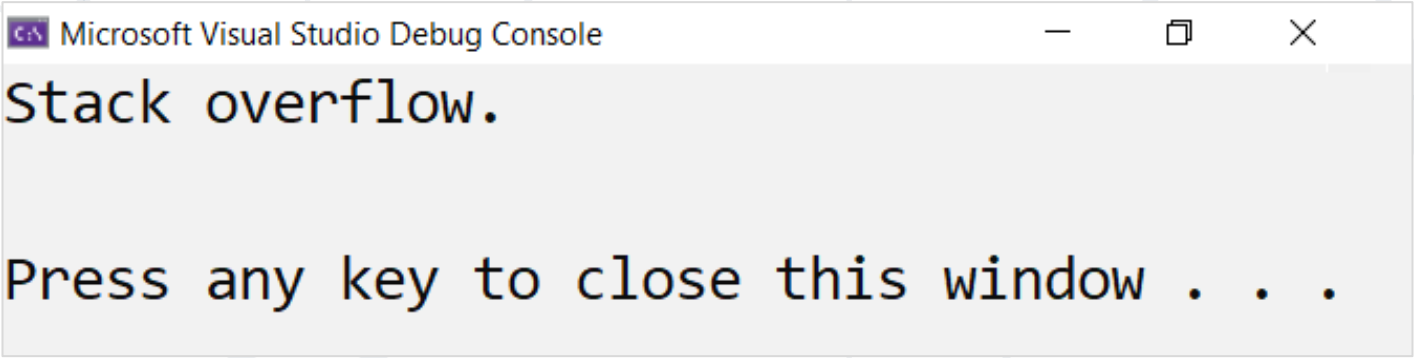
```
static long Fact(int n)
{
  if (n == 0) {
    return 1;
  }
  return n * Fact(n - 1);
}
```

- No function call **cost**
- Creates **local** variables
- Good for **linear** problems (no branching)

```
static long Fact(int n)
{
  long result = 1;
  for (int i = 1; i <= n; i++)
    result *= i;
  return result;
}
```

Infinite Recursion

- **Infinite recursion** == a method calls itself infinitely
  - Typically, infinite recursion == bug in the program
  - The bottom of the recursion is missing or wrong
  - In C# / Java / C++ causes "stack overflow" error



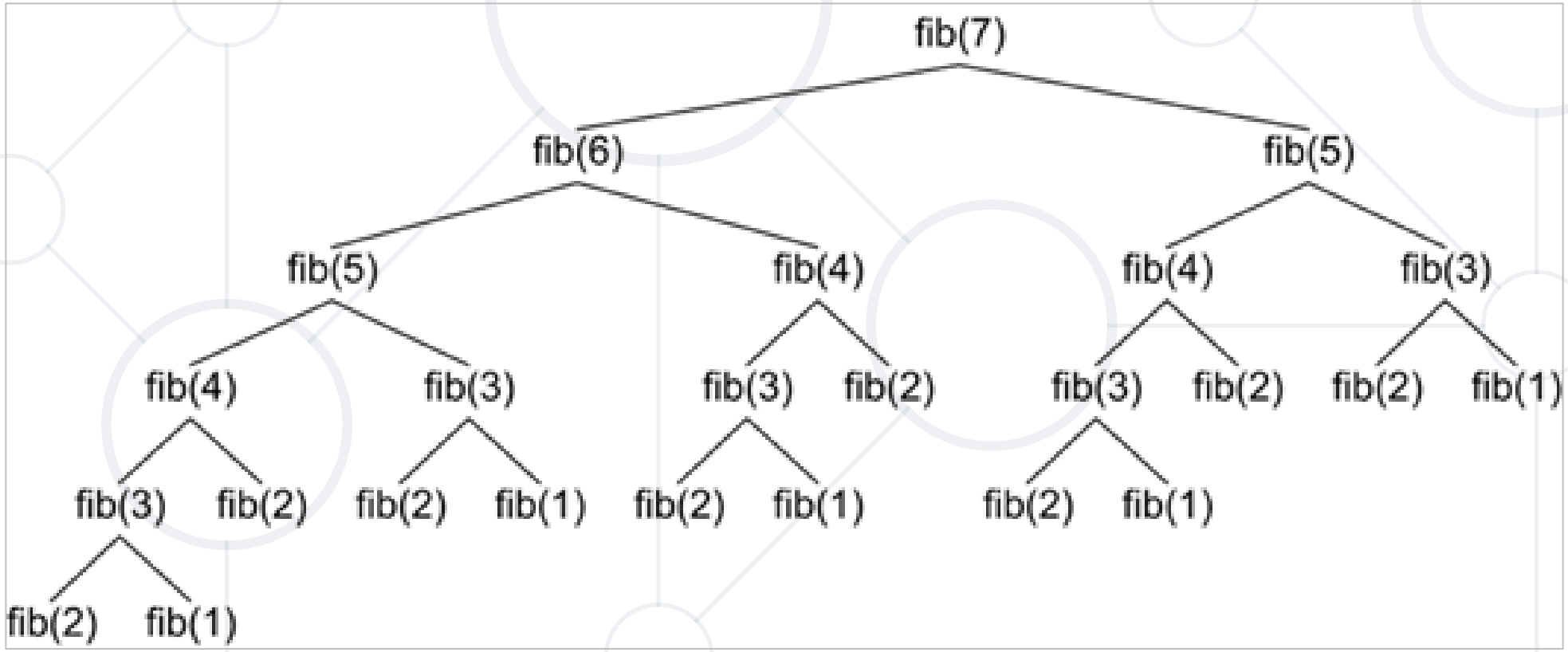
Recursion Can Be Harmful!

- When used incorrectly recursion could take too much **memory** and **computing power**

```
static long CalcFib(int number)
{
  if (number <= 1)
  {
    return 1;
  }
  return CalcFib(number - 1) + CalcFib(number - 2);
}
Console.WriteLine(CalcFib(10)); // 89
Console.WriteLine(CalcFib(50)); // This will hang!
```

How the Recursive Fibonacci Calculation Works?

- `fib(n)` makes about `fib(n)` recursive calls
- The same value is calculated many, many times!



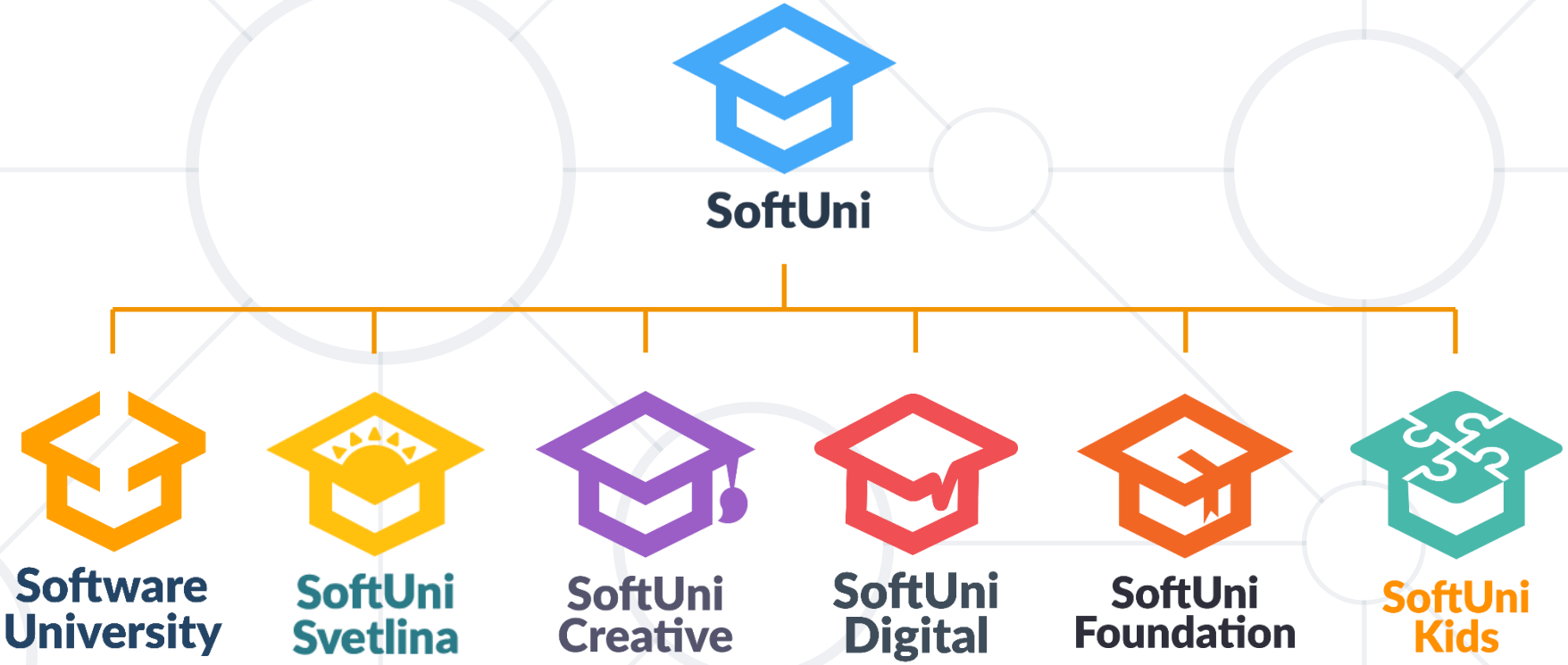
When to Use Recursion?

- Avoid recursion when an **obvious** iterative algorithm **exists**
  - Examples: **factorial**, **fibonacci** numbers
- Use recursion for **combinatorial** algorithms where:
  - At each step you need to **recursively** explore more than one possible continuation, i.e. **branched** recursive algorithms

Summary

- **Recursion**: a method calls itself with different input
  - Pre-actions → recursion → post-actions
- When **to use recursion**?
  - Branched recursive process
- When **to use iteration**?
  - Linear recursive process

Questions?



License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg