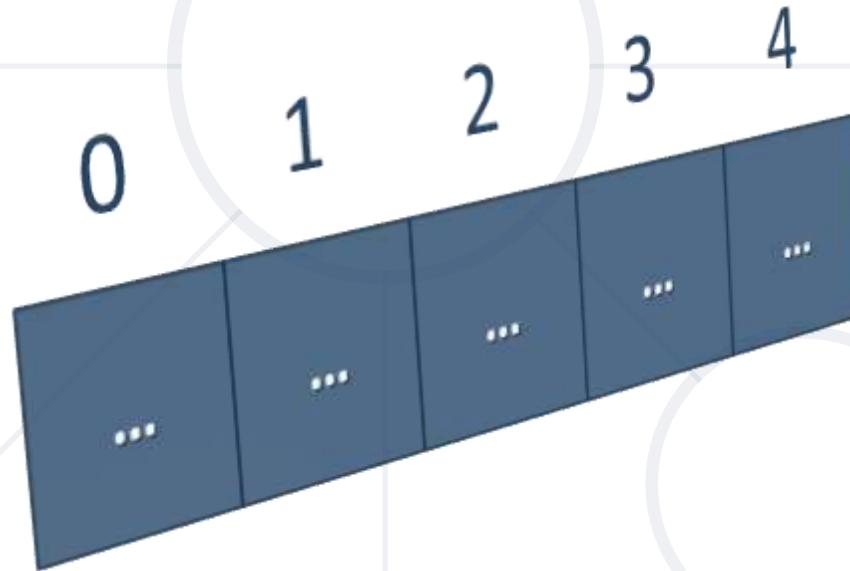


# Stacks and Queues

Processing Sequences of Elements



SoftUni Team

Technical Trainers

 Software University



SoftUni

Software University

<https://about.softuni.bg/>

# Table of Content

## 1. The "Stack" Data Structure (**LIFO** - last in, first out)

- The Class **Stack<T>**
- Push(), Pop(), Peek(),  
ToArray(), Contains() and Count

## 2. The "Queue" Data Structure (**FIFO** - first in, first out)

- The Class **Queue<T>**
- Enqueue(), Dequeue(), Peek(),  
ToArray(), Contains() and Count

Have a Question?



sli.do

**#csharp-advanced**



# The "Stack" Data Structure

Using the **Stack<T>** Class

# Stack – Abstract Data Type

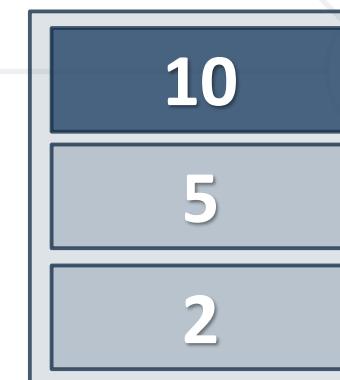
- **Stack** implements a **LIFO** (last in, first out) collection
  - **Push**: insert an element at the top of the stack
  - **Pop**: take the element from the top of the stack
  - **Peek**: retrieve the topmost element without removing it



Push

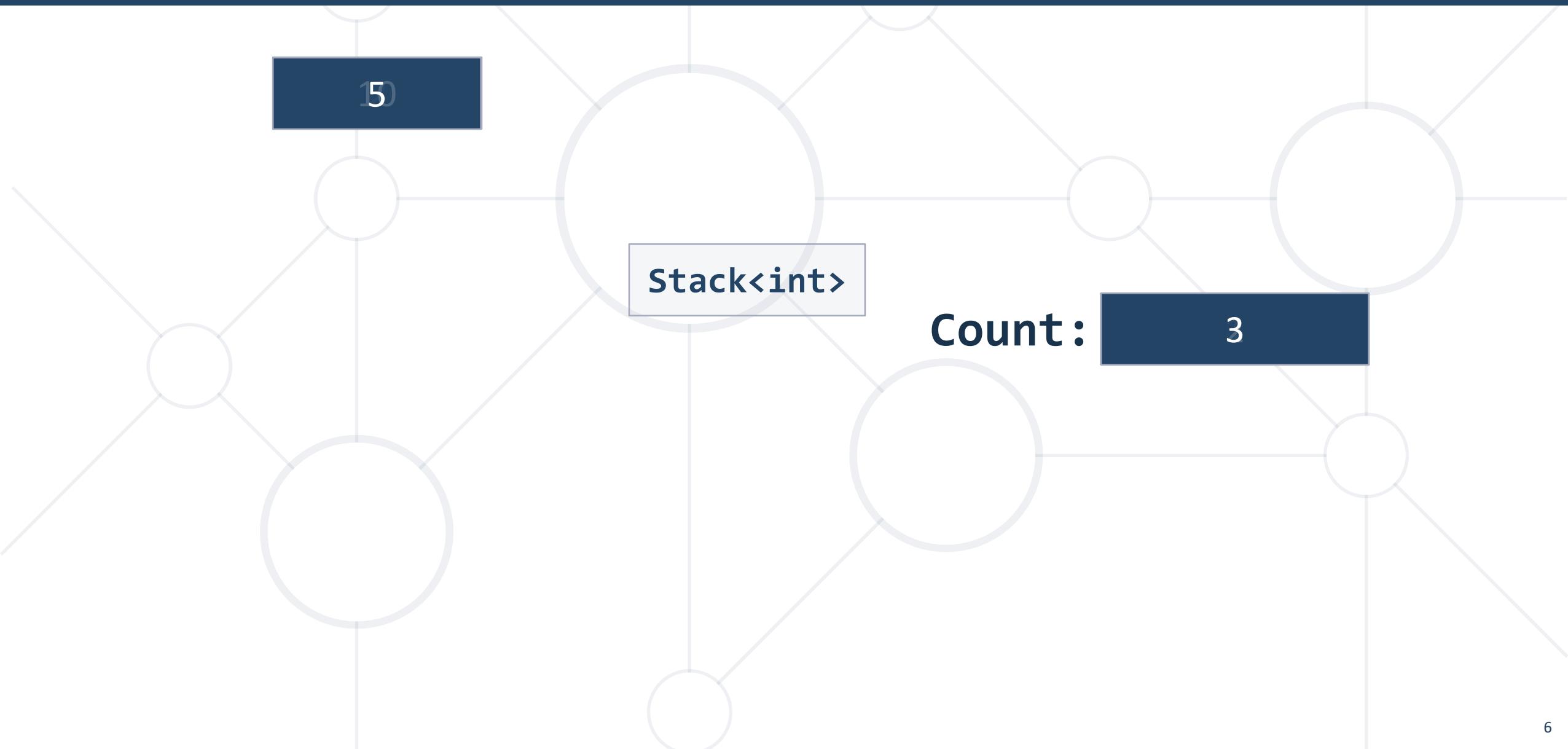


Pop



Peek

# Push() – Adds an Element On Top of the Stack



# Pop() – Returns and Removes the Last Element

Stack<int>

2

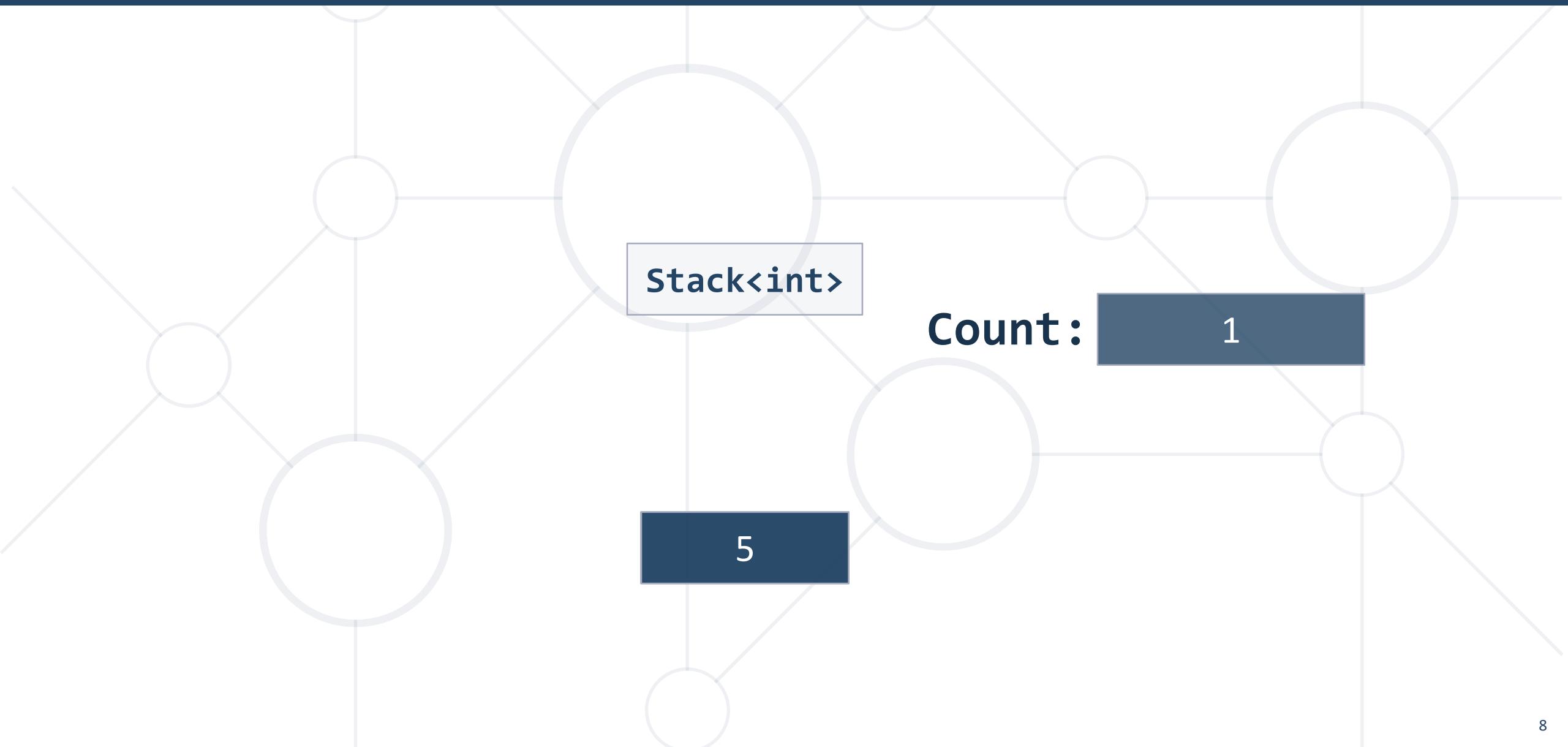
10

5

Count:

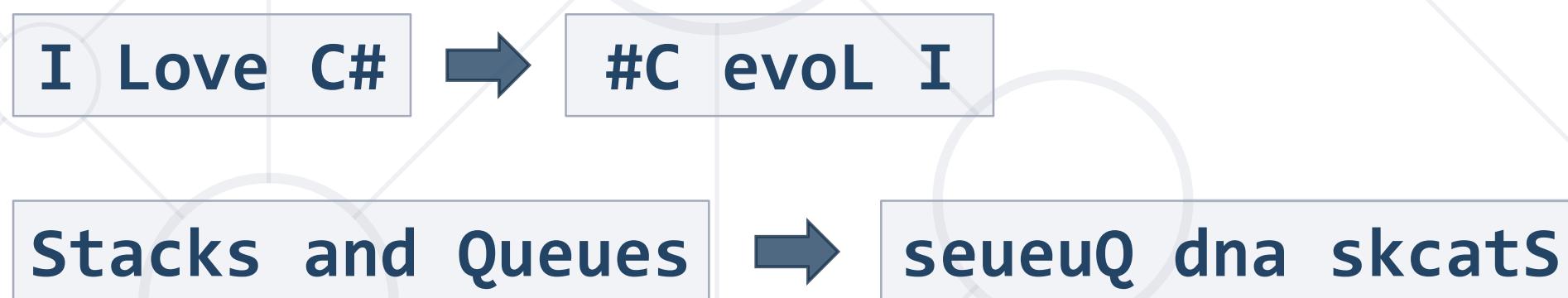
2

# Peek() - Returns the Last Element



# Problem: Reverse a String

- Create a program that:
  - Reads an input string
  - Reverses its letters backwards using a stack



Check your solution here: <https://judge.softuni.bg/Contests/1445/Stacks-and-Queues-Lab>

# Solution: Reverse Strings

```
var input = Console.ReadLine();
var stack = new Stack<char>();
foreach (var ch in input)
{
    stack.Push(ch);
}
while (stack.Count > 0)
{
    Console.Write(stack.Pop());
}
Console.WriteLine();
```

# Stack – Utility Methods

```
Stack<int> stack = new Stack<int>();
```

```
int count = stack.Count;
```

```
bool exists = stack.Contains(2);
```

```
int[] array = stack.ToArray();
```

```
stack.Clear();
```

```
stack.TrimExcess();
```

Retains the order  
of elements

Remove all  
elements

Shrink the  
internal array

# Problem: Stack Sum

- You are given a **list of numbers**. Push them into a stack and execute a sequence of **commands**:
  - Add <n1> <n2>**: adds given two numbers to the stack
  - Remove <count>**: if elements are enough, removes *count* elements
  - End**: print the sum in the remaining elements from the stack and exit

```
1 2 3 4
ADD 5 6
REmove 3
eNd
```

Sum: 6

```
3 5 8 4 1 9
add 19 32
remove 10
add 89 22
end
```

Sum: 192

# Solution: Stack Sum (1)

```
var input =
    Console.ReadLine().Split().Select(int.Parse).ToArray();
Stack<int> stack = new Stack<int>(input);
var commandInfo = Console.ReadLine().ToLower();

while (commandInfo != "end")
{
    var tokens = commandInfo.Split();
    var command = tokens[0].ToLower();
    if (command == "add")
        // TODO: Parse the numbers and push them to the stack
```

# Solution: Stack Sum (2)

```
else if (command == "remove") {
    var countOfRemovedNums = int.Parse(tokens[1]);
    if (countOfRemovedNums <= stack.Count)
        for (int i = 0; i < countOfRemovedNums; i++) {
            stack.Pop();
        }
}
commandInfo = Console.ReadLine().ToLower();
}
var sum = stack.Sum();
Console.WriteLine($"Sum: {sum}");
```

# Problem: Matching Brackets

- We are **given an arithmetic expression** with brackets  
**(nesting is allowed)**
- **Extract all sub-expressions** in brackets

```
1 + (2 - (2 + 3) * 4 / (3 + 1)) * 5
```



```
(2 + 3)
```

```
(3 + 1)
```

```
(2 - (2 + 3) * 4 / (3 + 1))
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1445#3>

# Solution: Matching Brackets

```
var input = Console.ReadLine();
var stack = new Stack<int>();
for (int i = 0; i < input.Length; i++) {
    char ch = input[i];
    if (ch == '(') {
        stack.Push(i);
    } else if (ch == ')') {
        int startIndex = stack.Pop();
        string contents = input.Substring(
            startIndex, i - startIndex + 1);
        Console.WriteLine(contents);
    }
}
```



# The "Queue" Data Structure

Using the `Queue<T>` Class

# Queue – Abstract Data Type

- **Queue** implements a **FIFO** (first in, first out) collection

- **Enqueue:** append an element at the end of the queue



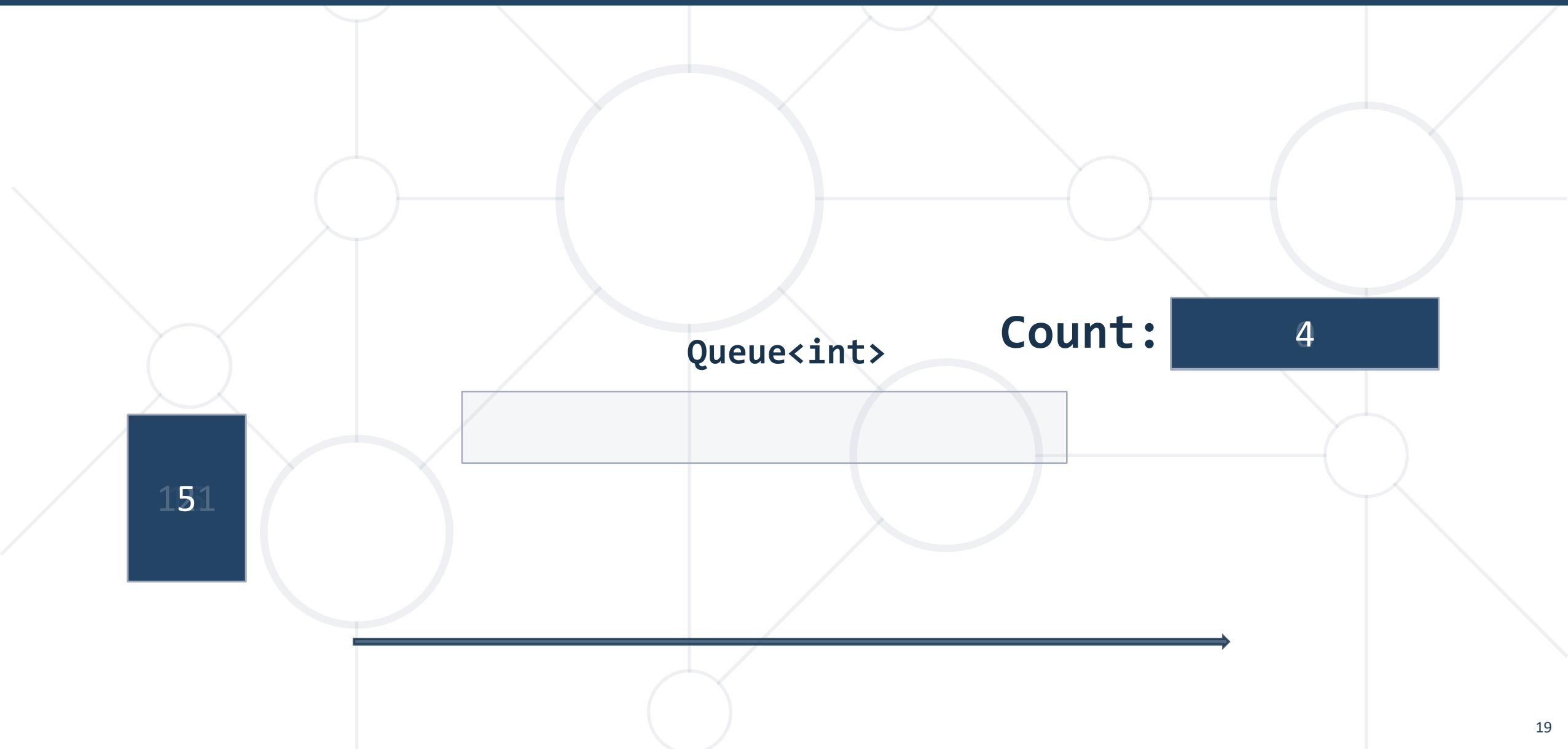
- **Dequeue:** remove the first element from the queue



- **Peek:** retrieve the first element of the queue without removing it



# Enqueue() – Adds an Element to the Front



# Dequeue() – Returns and Removes the First Element



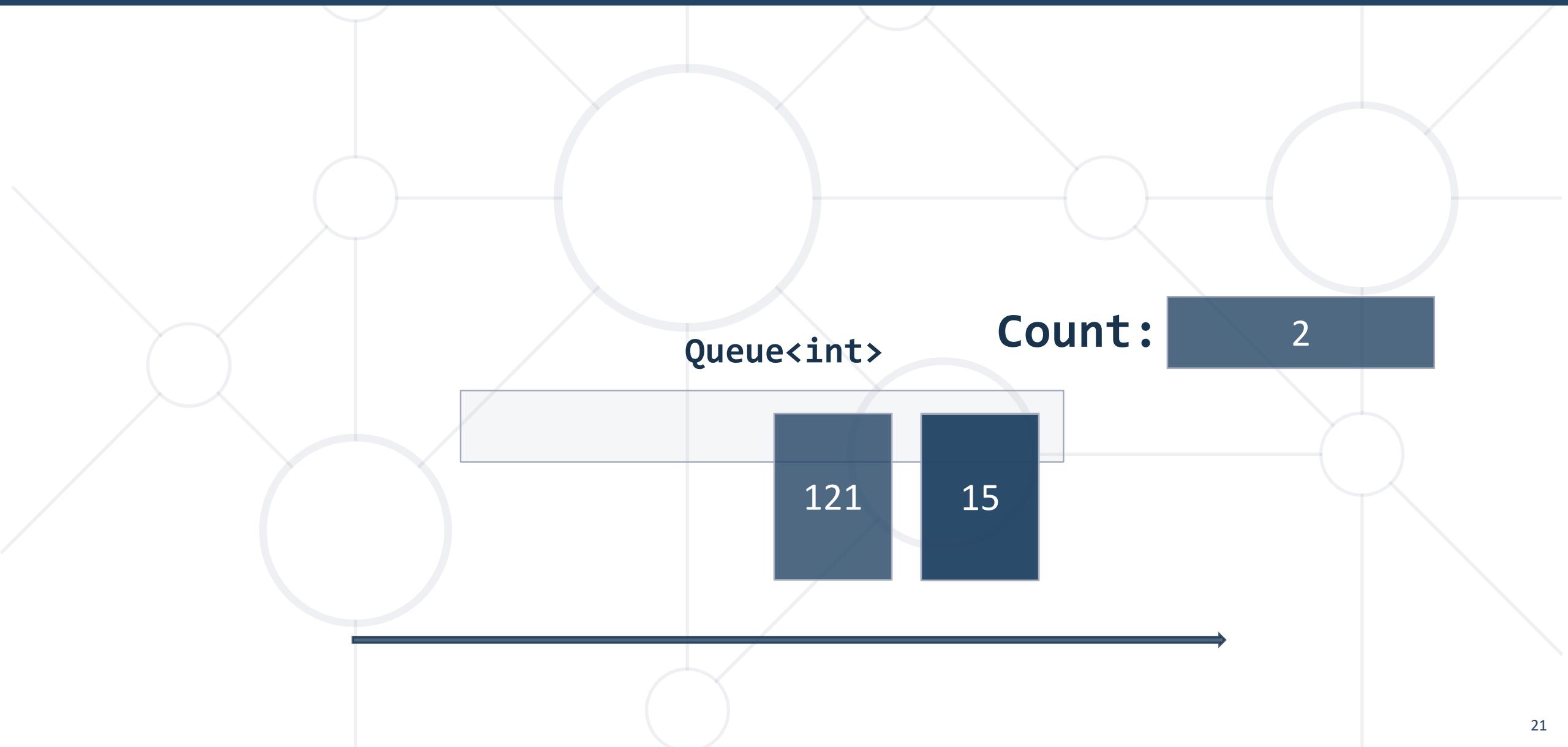
121

15

-3

5

# Peek() – Returns the First Element



# Queue – Utility Methods

```
Queue<int> queue = new  
Queue<int>();  
int count = queue.Count;  
bool exists = queue.Contains(2);  
int[] array = queue.ToArray();  
queue.Clear();  
queue.TrimExcess();
```

Remove all  
elements

Resize the  
internal array

Retains the order  
of elements

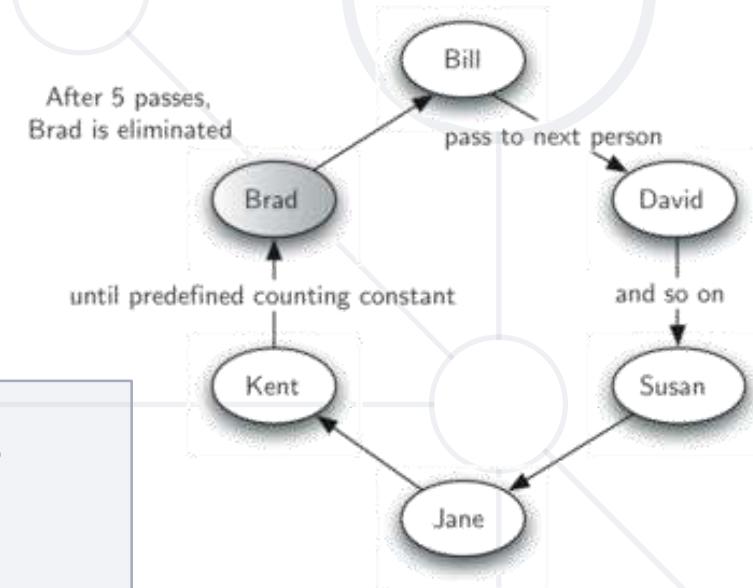
# Problem: Hot Potato

- Children **form a circle** and pass a hot potato **clockwise**
- Every  $n^{\text{th}}$  toss **a child is removed** until **only one remains**
  - **Upon removal** the potato is passed **along**
- Print the child that remains last

Alva James William  
2

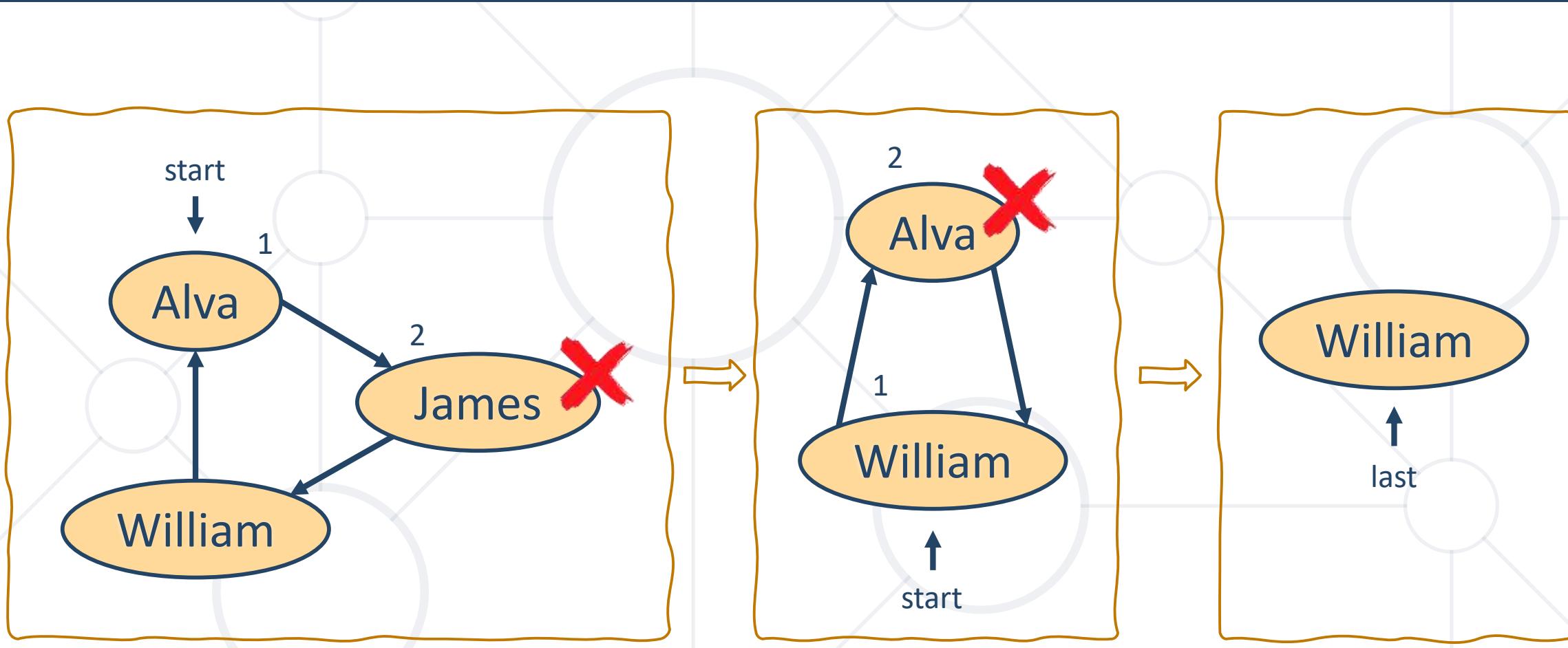


Removed James  
Removed Alva  
Last is William



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1445#6>

# Hot Potato: Illustration



# Solution: Hot Potato

```
var children = Console.ReadLine().Split(' ');
var number = int.Parse(Console.ReadLine());
Queue<string> queue = new Queue<string>(children);
while (queue.Count > 1) {
    for (int i = 1; i < number; i++) {
        queue.Enqueue(queue.Dequeue());
    }
    Console.WriteLine($"Removed {queue.Dequeue()}");
}
Console.WriteLine($"Last in {queue.Dequeue()}");
```

Copies elements from the specified collection and keeps their order

# Problem: Traffic Jam

- Cars are **queuing up** at a **traffic light**
- At every **green light**, n cars **pass** the crossroad
- After the **end command**, print **how many cars have passed**

```
3
Enzo's car
Jade's car
Mercedes CLS
Audi
green
BMW X5
green
end
```



```
Enzo's car passed!
Jade's car passed!
Mercedes CLS passed!
Audi passed!
BMW X5 passed!
5 cars passed the crossroads.
```

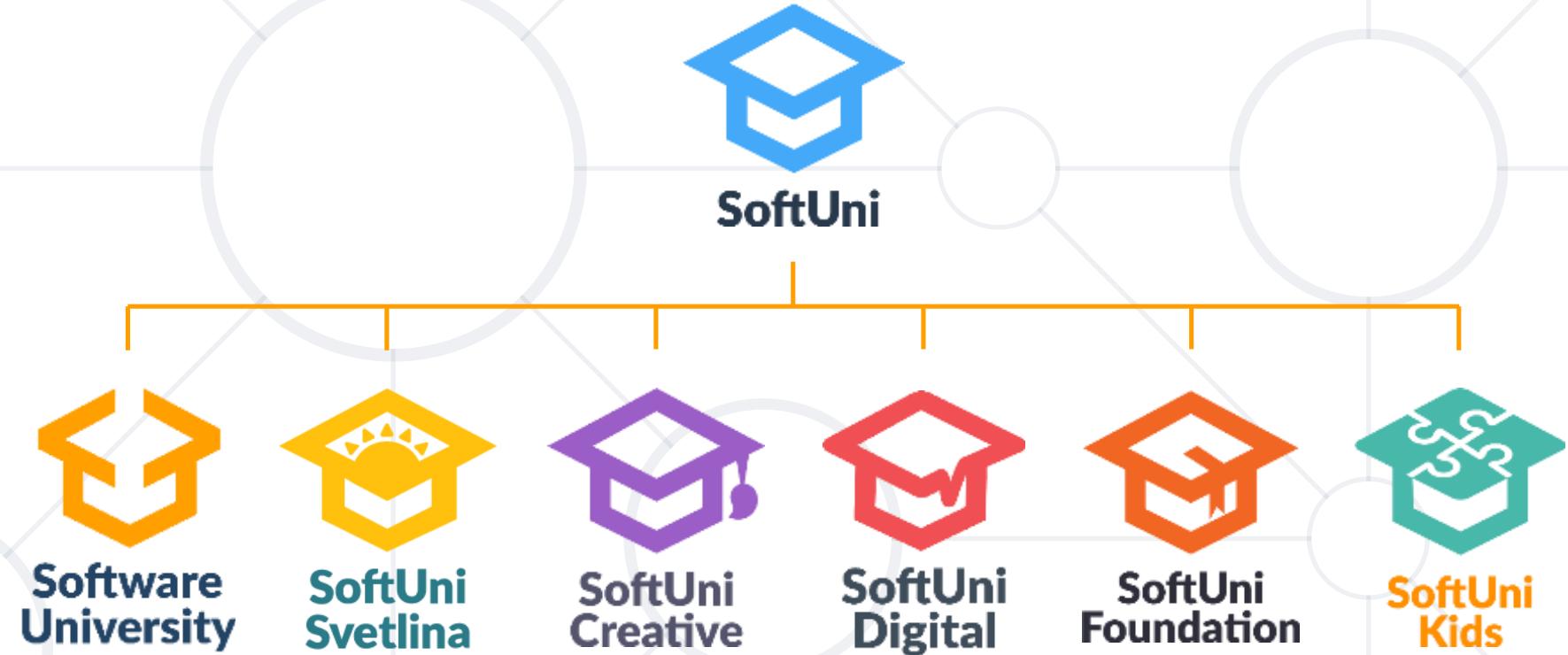
# Solution: Traffic Jam

```
int n = int.Parse(Console.ReadLine());
var queue = new Queue<string>();
int count = 0;
string command;
while ((command = Console.ReadLine()) != "end")
{
    if (command == "green")
        // TODO: Add green Light Logic
    else
        queue.Enqueue(command);
}
Console.WriteLine($"{count} cars passed the crossroads.");
```

- **Stack<T>**
  - LIFO data structure (last-in, first-out)
  - **Push()**, **Pop()**, **Peek()**
- **Queue<T>**
  - FIFO data structure (first-in, first-out)
  - **Enqueue()**, **Dequeue()**, **Peek()**



# Questions?



# SoftUni Diamond Partners

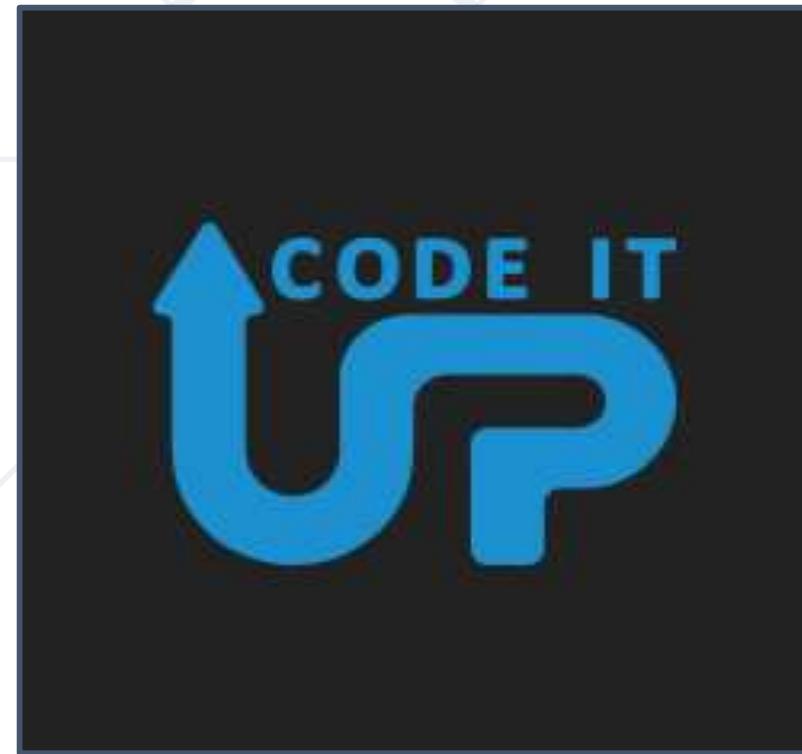


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

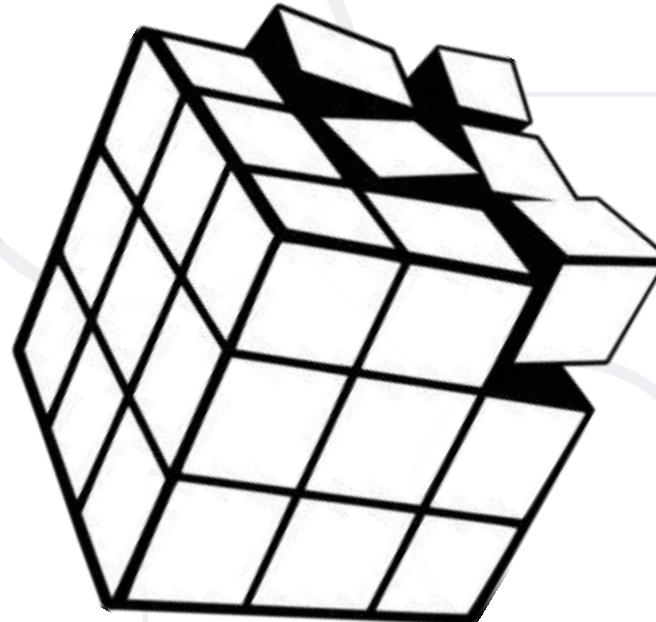


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Multidimensional Arrays

Processing Matrices and Jagged Arrays



**SoftUni**



**SoftUni Team**

**Technical Trainers**

 **Software University**

**Software University**

<https://about.softuni.bg/>

## 1. Multidimensional Arrays

- Creating Matrices and Multidimensional Arrays
- Accessing Their Elements
- Reading and Printing



## 2. Jagged Arrays (Arrays of Arrays)

- Creating a Jagged Array
- Accessing Their Elements
- Reading and Printing

Have a Question?



sli.do

**#csharp-advanced**



# Multidimensional Arrays

Definition and Usage

# What is a Multidimensional Array?

- Array is a systematic arrangement of similar objects
- **Multidimensional arrays** have more than one dimension
  - The most used multidimensional arrays are the 2-dimensional, also called **matrices**



R O W S	COLS				
	[0, 0]	[0, 1]	[0, 2]	[0, 3]	[0, 4]
	[1, 0]	[1, 1]	[1, 2]	[1, 3]	[1, 4]
	[2, 0]	[2, 1]	[2, 2]	[2, 3]	[2, 4]

Col Index

Row Index

# Creating Multidimensional Arrays

- Creating a multidimensional array in C#
  - Use the **new** keyword
  - Must specify the size of each dimension



```
int[,] intMatrix = new int[3, 4];
float[,] floatMatrix = new float[8, 2];
string[,,] stringCube = new string[5, 5, 5];
```

# Initializing Multidimensional Arrays

- Initializing with values:

```
int[,] matrix = {  
    {1, 2, 3, 4}, // row 0 values  
    {5, 6, 7, 8} // row 1 values  
};
```

- Two-dimensional arrays represent **rows with values**
- The **rows** represent the first dimension and the **columns**
  - the second (**the one inside the first**)

# Accessing Elements

- Accessing N-dimensional array element:

```
nDimensionalArray[index1, ... , indexn]
```

- Getting element value:

```
int[,] array = {{10, 20, 30}, {40, 50, 60}};  
int element11 = array[1, 0]; // element10 = 40
```

0	1	2	
10	20	30	row 0
40	50	60	row 1

- Setting element value:

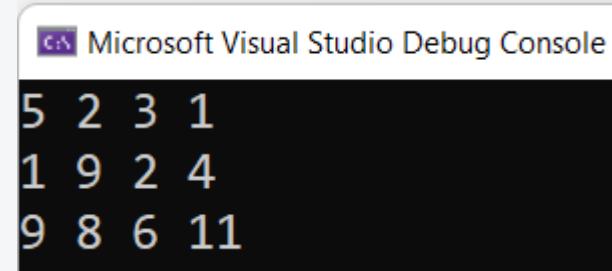
```
int[,] array = new int[3, 4];  
for (int row = 0; row < array.GetLength(0); row++)  
    for (int col = 0; col < array.GetLength(1); col++)  
        array[row, col] = row + col;
```

Returns the size  
of the dimension

# Printing a Matrix – Example (1)

```
int[,] matrix =
{
    { { 5, 2, 3, 1 },
      { 1, 9, 2, 4 },
      { 9, 8, 6, 11 }   };
}

for (int row = 0; row < matrix.GetLength(0); row++)
{
    for (int col = 0; col < matrix.GetLength(1); col++)
    {
        Console.Write("{0} ", matrix[row, col]);
    }
    Console.WriteLine();
}
```



# Printing Matrix – Example (2)

- Foreach iterates through all the elements in the matrix

```
int[,] matrix = {  
    { 5, 2, 3, 1 },  
    { 1, 9, 2, 4 },  
    { 9, 8, 6, 9 }  
};  
  
foreach (int element in matrix)  
{  
    Console.WriteLine(element + " ");  
}
```

Microsoft Visual Studio Debug Console

5 2 3 1 1 9 2 4 9 8 6 9



# Problem: Sum Matrix Elements

- Read a matrix from the console
- Print the number of rows
- Print the number of columns
- Print the **sum of all numbers** in the matrix

```
3, 6  
7, 1, 3, 3, 2, 1  
1, 3, 9, 8, 5, 6  
4, 6, 7, 9, 1, 0
```



```
3  
6  
76
```

```
3, 4  
1, 2, 3, 1  
1, 2, 2, 4  
2, 2, 2, 2
```



```
3  
4  
24
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1452#0>

# Solution: Sum Matrix Elements (1)

```
int[] sizes = Console.ReadLine().Split(", ")  
    .Select(int.Parse).ToArray();  
  
int[,] matrix = new int[sizes[0], sizes[1]];  
  
for (int row = 0; row < matrix.GetLength(0); row++) {  
  
    int[] colElements = Console.ReadLine().Split(", ")  
        .Select(int.Parse).ToArray();  
  
    for (int col = 0; col < matrix.GetLength(1); col++)  
        matrix[row, col] = colElements[col];  
  
}
```

Gets length of 0th dimension (rows)

Gets length of 1st dimension (cols)

# Solution: Sum Matrix Elements (2)

```
int sum = 0;  
  
for (int row = 0; row < matrix.GetLength(0); row++)  
{  
    for (int col = 0; col < matrix.GetLength(1); col++)  
        sum += matrix[row, col];  
}  
  
Console.WriteLine(matrix.GetLength(0));  
Console.WriteLine(matrix.GetLength(1));  
Console.WriteLine(sum);
```

# Problem: Sum Matrix Columns

- Read matrix sizes
- Read a matrix from the console
- Print the **sum of all numbers** in matrix columns

3,	6
7	1
3	3
3	2
1	1
1	3
3	9
8	5
5	6
4	6
6	7
7	9
9	1
1	0

12
10
19
20
8
7

3,	3
1	2
2	3
4	5
5	6
7	8
8	9

12
15
18

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1452#1>

# Solution: Sum Matrix Columns (1)

```
var sizes = Console.ReadLine()
    .Split(", ").Select(int.Parse).ToArray();
int[,] matrix = new int[sizes[0], sizes[1]];
for (int r = 0; r < matrix.GetLength(0); r++) {
    var col = Console.ReadLine().Split().Select(int.Parse).ToArray();
    for (int c = 0; c < matrix.GetLength(1); c++) {
        matrix[r, c] = col[c];
    }
}
```

# Solution: Sum Matrix Columns (2)

```
for (int c = 0; c < matrix.GetLength(1); c++) {  
    int sum = 0;  
    for (int r = 0; r < matrix.GetLength(0); r++) {  
        sum += matrix[r, c];  
    }  
    Console.WriteLine(sum);  
}
```

# Problem: Square with Maximum Sum

- Find **2x2 square** with max sum in given matrix
  - Read matrix from the console
  - Find **bigest sum** of 2x2 submatrix
  - Print the result as a **new matrix**, followed by **the sum**

3, 6	
7, 1, 3, 3, 2, 1	
1, 3, 9, 8, 5, 6	
4, 6, 7, 9, 1, 0	

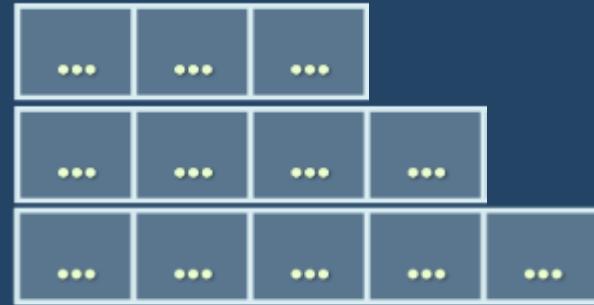


9 8
7 9
33

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1452#4>

# Solution: Square with Maximum Sum

```
// TODO: Read the input from the console
for (int row = 0; row < matrix.GetLength(0) - 1; row++) {
    for (int col = 0; col < matrix.GetLength(1) - 1; col++) {
        var newSquareSum = matrix[row, col] +
            matrix[row + 1, col] +
            matrix[row, col + 1] +
            matrix[row + 1, col + 1];
        // TODO: Check if the sum is bigger
        // → remember the best sum, row and col
    }
}
// TODO: Print the square with the max sum
```



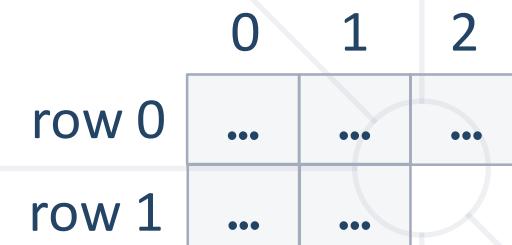
# Jagged Arrays

## Definition and Usage

# What is Jagged Array

- **Jagged arrays** are multidimensional arrays
  - But each dimension may have a different size
  - A jagged array is an **array of arrays**
  - Each of the arrays has **different length**

```
int[][] jagged = new int[2][];  
jagged[0] = new int[3];  
jagged[1] = new int[2];
```



- **Accessing elements**

```
int element = jagged[0][1];
```

Col Index

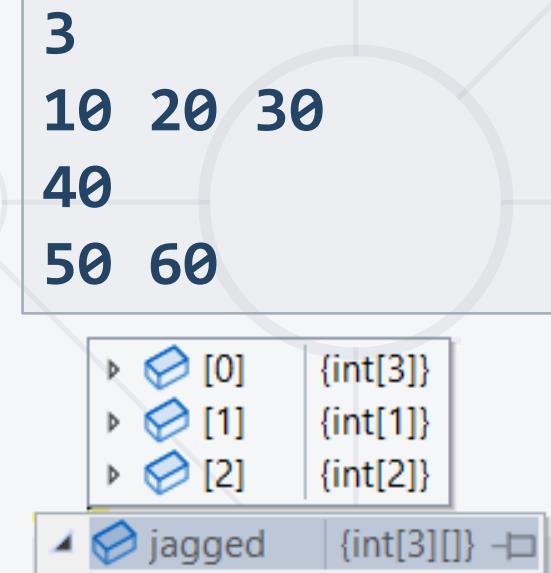
Row Index

# Reading a Jagged Array

```
int rowsCount = int.Parse(Console.ReadLine());
int[][] jagged = new int[rowsCount][];

for (int row = 0; row < jagged.Length; row++)
{
    string[] nums = Console.ReadLine().Split(' ');
    jagged[row] = new int[nums.Length];

    for (int col = 0; col < jagged[row].Length; col++)
    {
        jagged[row][col] = int.Parse(nums[col]);
    }
}
```



# Printing a Jagged Array – Example

- Using a **for** loop

```
int[][] matrix = ReadJaggedArray();
for (int row = 0; row < matrix.Length; row++)
{
    for (int col = 0; col < matrix[row].Length; col++)
        Console.Write("{0} ", matrix[row][col]);
    Console.WriteLine();
}
```

Implement your  
custom method

- Using a **foreach** loop

```
int[][] matrix = ReadJaggedArray();
foreach (int[] row in matrix)
    Console.WriteLine(string.Join(" ", row));
```

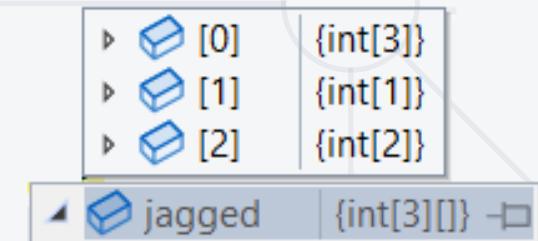
# Read and Print a Jagged Array (Short Version)

```
// Allocate the array rows
int rows = int.Parse(Console.ReadLine());
int[][] jagged = new int[rows][];

// Read the jagged array
for (int row = 0; row < jagged.Length; row++)
    jagged[row] = Console.ReadLine().Split(' ')
        .Select(int.Parse).ToArray();

// Print the jagged array
foreach (int[] row in jagged)
    Console.WriteLine(string.Join(" ", row));
```

```
3
10 20 30
40
50 60
```



# Problem: Jagged-Array Modification

- On the first line you will get the number **rows**
- On the next lines you will get the **elements for each row**
- Until you receive "**END**", read commands
  - **Add {row} {col} {value}**
  - **Subtract {row} {col} {value}**
- If the coordinates are invalid, print "**Invalid coordinates**"
- When you receive "**END**", print the jagged array

# Jagged-Array Modification – Example

```
3  
1 2 3  
4 5 6 7  
8 9 10  
Add 0 0 5  
Subtract 1 2 2  
Subtract 1 4 7  
END
```



Invalid coordinates

6	2	3	
4	5	4	7
8	9	10	

0	1	2		
row 0	1	2	3	
row 1	4	5	6	7
row 2	8	9	10	

# Solution: Jagged-Array Modification (1)

```
int rowSize = int.Parse(Console.ReadLine());
int[][] matrix = new int[rowSize][];

for (int row = 0; row < rowSize; row++)
{
    int[] columns = Console.ReadLine()
        .Split()
        .Select(int.Parse)
        .ToArray();
    matrix[row] = columns;
}
// continues on the next slide...
```

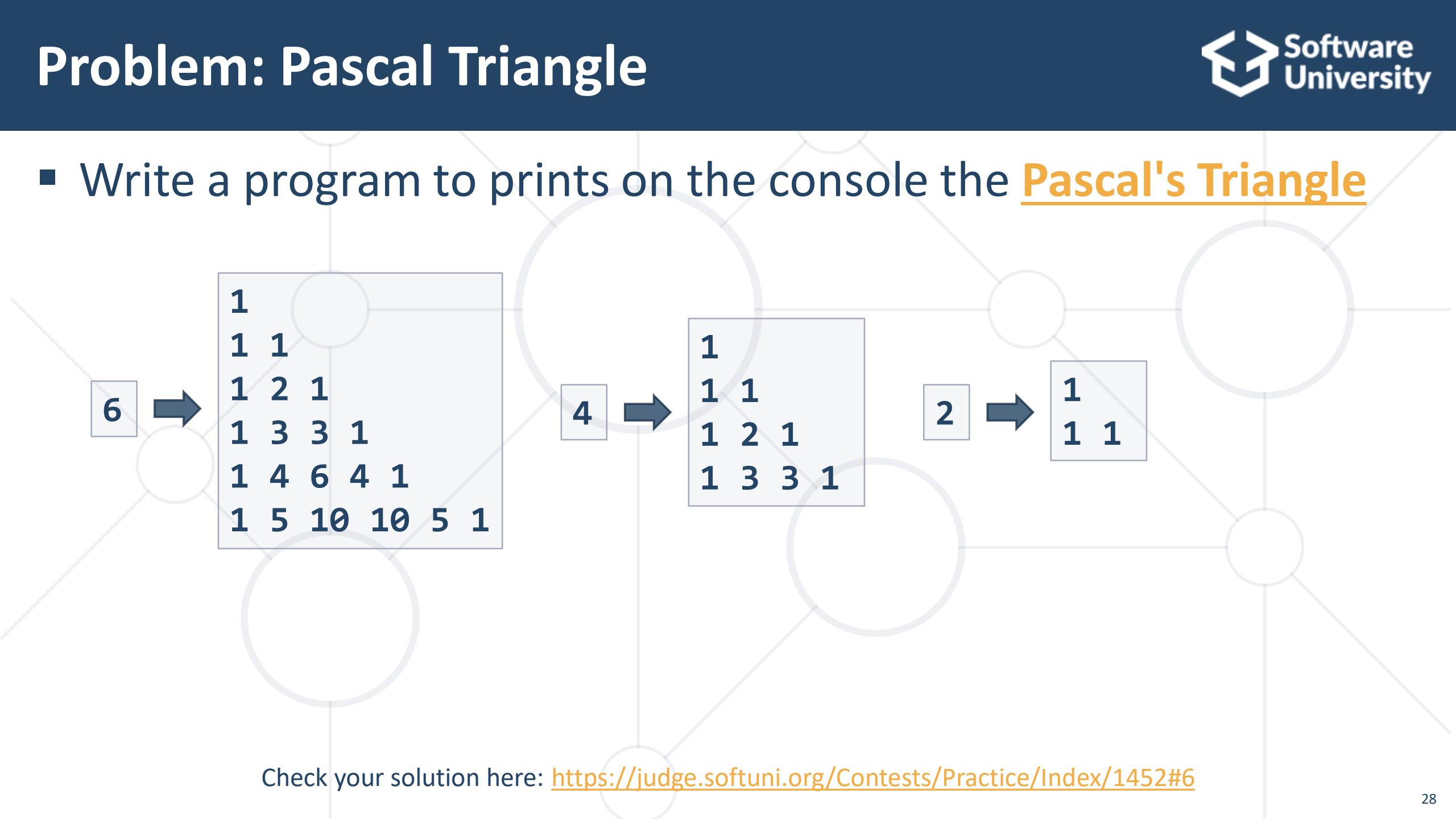
# Solution: Jagged-Array Modification (2)

```
string line;
while ((line = Console.ReadLine()) != "END") {
    string[] tokens = line.Split();
    string command = tokens[0];
    int row = int.Parse(tokens[1]);
    int col = int.Parse(tokens[2]);
    int value = int.Parse(tokens[3]);
    if (row < 0 || row >= matrix.Length || ... )
        Console.WriteLine("Invalid coordinates");
    else
        { // TODO: Execute the command }
}
// TODO: Print the matrix
```

Check the row  
and col ranges

# Problem: Pascal Triangle

- Write a program to prints on the console the Pascal's Triangle



1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	2	1	1	1	1	1
1	3	3	1	1	1	1
1	4	6	4	1	1	1
1	5	10	10	5	1	1

1	1	1	1	1
1	2	1	1	1
1	3	3	1	1
1	4	6	4	1

1	1	1
1	1	1

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1452#6>

# Solution: Pascal Triangle (1)

```
int height = int.Parse(Console.ReadLine());
long[][] triangle = new long[height][];
int currentWidth = 1;

for (long row = 0; row < height; row++)
{
    triangle[row] = new long[currentWidth];
    long[] currentRow = triangle[row];
    currentRow[0] = 1;
    currentRow[currentRow.Length - 1] = 1;
    currentWidth++;
    // TODO: Fill elements for each row (next slide)
}
```

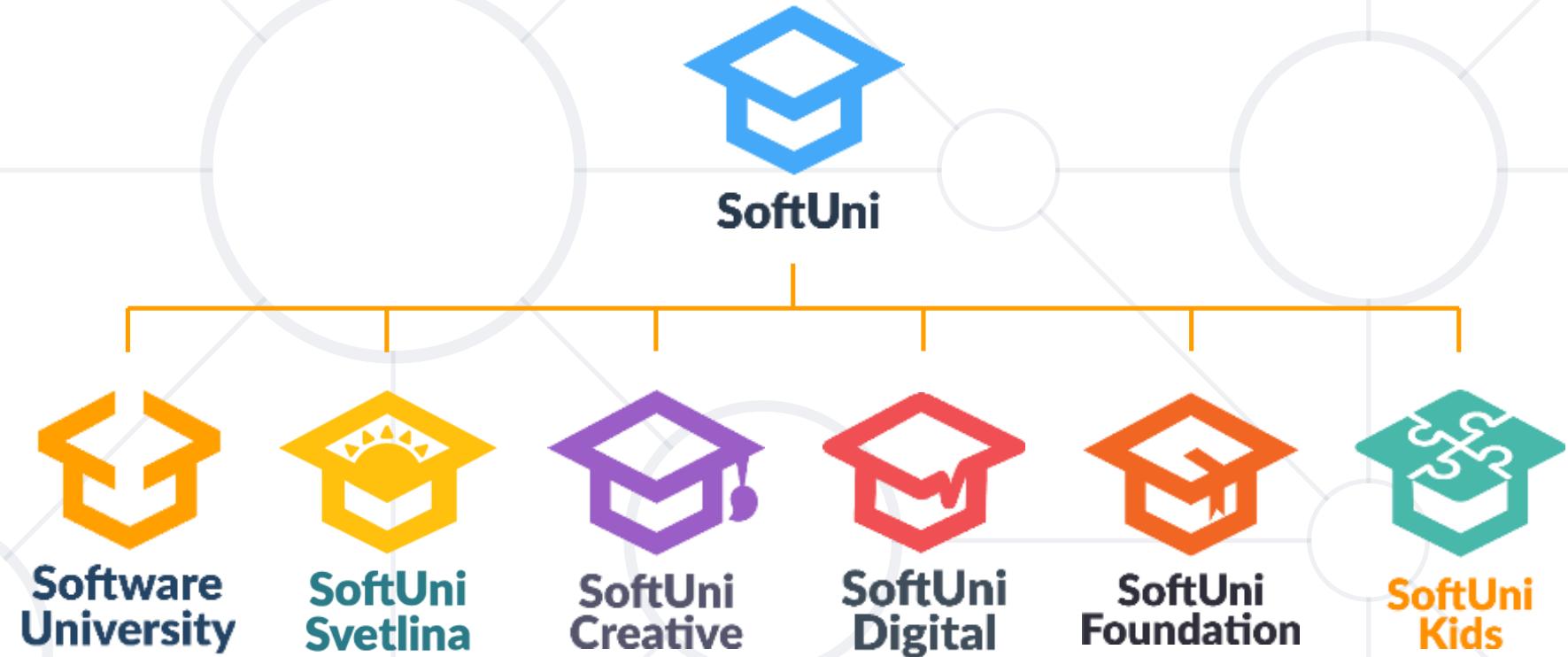
# Solution: Pascal Triangle (2)

```
if (currentRow.Length > 2)
{
    for (int i = 1; i < currentRow.Length - 1; i++)
    {
        long[] previousRow = triangle[row - 1];
        long prevoiousRowSum = previousRow[i] + previousRow[i - 1];
        currentRow[i] = prevoiousRowSum;
    }
}
// TODO: Print triangle
foreach (long[] row in triangle)
    Console.WriteLine(string.Join(" ", row));
```

- Multidimensional arrays
  - Have **more than one** dimension
  - Two-dimensional arrays are like tables with **rows** and **columns**
- Jagged arrays
  - Arrays of arrays
  - Each **element** is an array **itself**



# Questions?



# SoftUni Diamond Partners

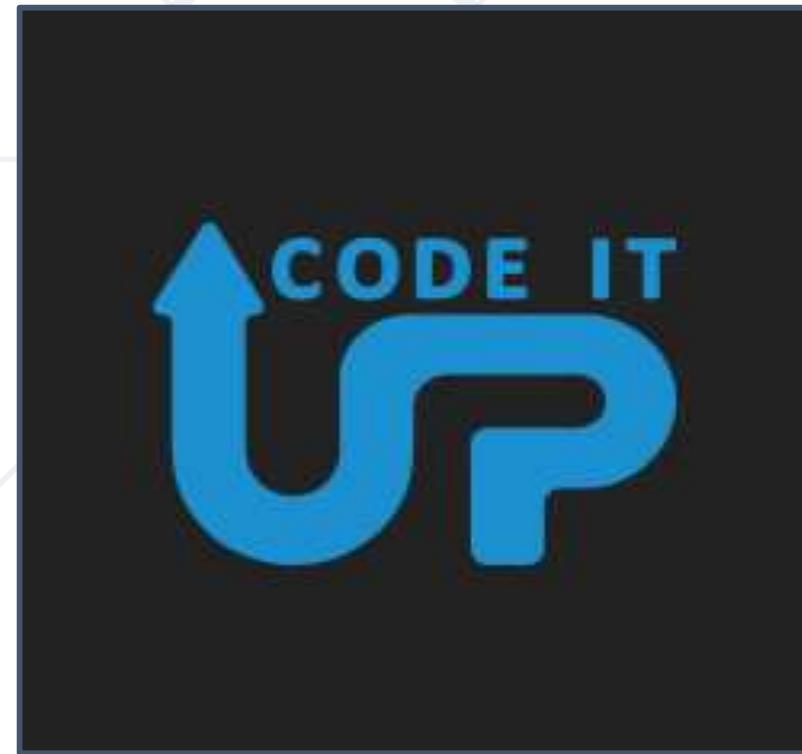


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Sets and Dictionaries Advanced

Sets and Multi-Dictionaries, Nested Dictionaries



SoftUni Team

Technical Trainers



SoftUni

Software University

<https://about.softuni.bg/>

# Table of Contents

- 1. Dictionary<K, V> Overview**
- 2. Multi-Dictionaries**
  - A Key Holds Multiple Values
- 3. Nested Dictionaries**
  - A Dictionary Holding Another Dictionary
- 4. Set<T>**
  - **HashSet<T>** and **SortedSet<T>**
  - **List<T>** vs **Set<T>**

Have a Question?



sli.do

#csharp-advanced



# Dictionary<K, V> Overview

Collection of Keys Mapped to Values

# Associative Arrays (Maps, Dictionaries)

- Associative arrays are arrays indexed by keys
  - Not by the numbers 0, 1, 2, ... (like arrays)
- Hold a set of pairs {key → value}



Key	Value
John Smith	+1-555-8976
Lisa White	+1-555-1234
Sam Doe	+1-555-5030

- **Dictionary<K, V>**: collection of {key, value} pairs
- Keys are **unique**, each mapping to a value
- **Dictionary<K, V>** keeps the keys in their **order of addition**

```
var fruits = new Dictionary<string, double>();  
fruits["banana"] = 2.20;  
fruits["apple"] = 1.40;  
fruits["kiwi"] = 3.20;  
Console.WriteLine(string.Join(", ", fruits.Keys));
```

# Sorted Dictionary

- **SortedDictionary<K, V>**: collection of {key, value} pairs
  - Keeps its **keys** always **sorted**
  - Implemented internally by a balanced search tree

```
var fruits = new SortedDictionary<string, double>();  
fruits["kiwi"] = 4.50;  
fruits["orange"] = 2.50;  
fruits["banana"] = 2.20;  
Console.WriteLine(string.Join(", ", fruits.Keys));
```

# Built-In Methods (1)

- Add(key, value) method

```
var airplanes = new Dictionary<string, int>();  
airplanes.Add("Boeing 737", 130);  
airplanes.Add("Airbus A320", 150);
```

- Remove(key) method

```
var airplanes = new Dictionary<string, int>();  
airplanes.Add("Boeing 737", 130);  
airplanes.Remove("Boeing 737");
```

# Built-In Methods (2)

- ContainsKey(key) – fast!

```
var dictionary = new Dictionary<string, int>();  
dictionary.Add("Airbus A320", 150);  
if (dictionary.ContainsKey("Airbus A320"))  
    Console.WriteLine($"Airbus A320 key exists");
```

- ContainsValue(value) – slow!

```
var dictionary = new Dictionary<string, int>();  
dictionary.Add("Airbus A320", 150);  
Console.WriteLine(airplanes.ContainsKey(150)); // True  
Console.WriteLine(airplanes.ContainsKey(100)); // False
```

# Problem: Count Same Values in Array

- Read a list of **real numbers** and print them along with their **number of occurrences**

8 2.5 2.5 8 2.5

8 - 2 times  
2.5 - 3 times

1.5 5 1.5 3

1.5 - 2 times  
5 - 1 times  
3 - 1 times

# Solution: Count Same Values in Array

```
double[] nums = Console.ReadLine().Split(' ')
    .Select(double.Parse).ToArray();

var counts = new Dictionary<double, int>();

foreach (var num in nums)
{
    if (counts.ContainsKey(num))
        counts[num]++;
    else
        counts[num] = 1;
}

foreach (var num in counts)
    Console.WriteLine($"{num.Key} - {num.Value} times");
```

counts[num] hold show many times num occurs in nums

# Sorting Collections

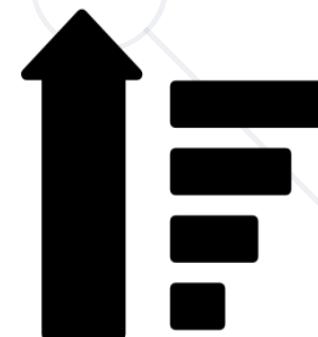
- Using OrderBy() to sort collections:

```
List<int> nums = { 1, 5, 2, 4, 3 };
nums = nums
    .OrderBy(num => num)
    .ToList();
```



- Using OrderByDescending() to sort collections:

```
List<int> nums = { 1, 5, 2, 4, 3 };
nums = nums.OrderByDescending(num => num).ToList();
Console.WriteLine(String.Join(", ", nums));
```



# Sorting Collections by Multiple Criteria

- Using ThenBy() to sort collections by multiple criteria:

```
var products = new Dictionary<int, string>();  
  
Dictionary<int, string> sortedDict = products  
    .OrderBy(pair => pair.Value)  
    .ThenBy(pair => pair.Key)  
    .ToDictionary(pair => pair.Key,  
                 pair => pair.Value);
```



# Problem: Largest 3 Numbers

- Read a list of integers
- Print the **largest 3** of them (or less for shorter lists)
- Print them in **descending order**

10 30 15 20 50 5



50 30 20

1 2 3



3 2 1

20 30



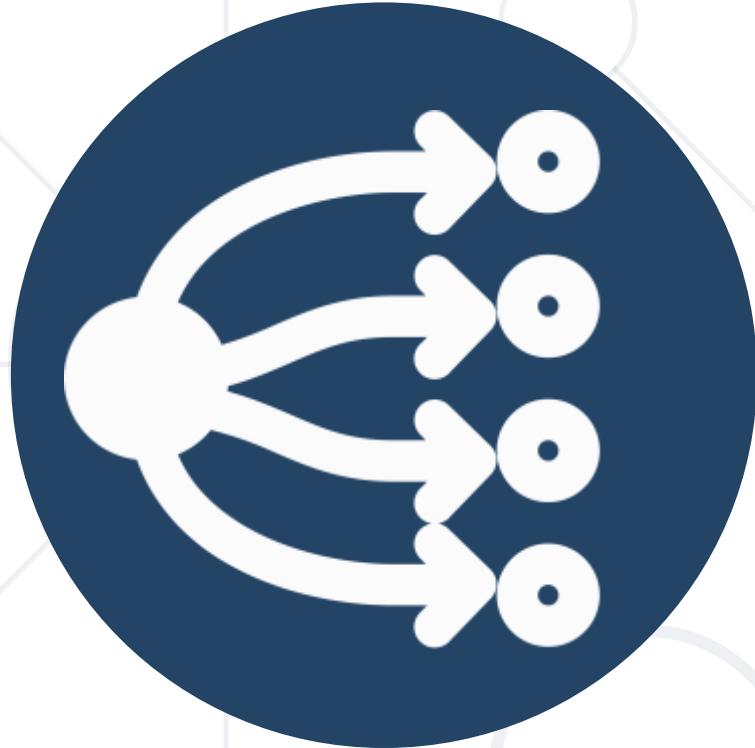
30 20

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1465#2>

# Solution: Largest 3 Numbers

```
int[] numbers = Console.ReadLine()  
    .Split()  
    .Select(int.Parse)  
    .OrderByDescending(n => n)  
    .ToArray();  
  
int count = numbers.Length >= 3 ? 3 : numbers.Length;  
  
for (int i = 0; i < count; i++)  
    Console.WriteLine($"{numbers[i]}");
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1465#2>



# Multi-Dictionaries

Dictionaries Holding a List of Values

# Multi-Dictionaries

- A dictionary could hold a **set of values** by given key
  - Example: student may have multiple grades:
    - Peter → [5, 5, 6]
    - Kevin → [6, 6, 3, 4, 6]



```
var grades = new Dictionary<string, List<int>>();  
grades["Peter"] = new List<int>();  
grades["Peter"].Add(5);  
grades["Kevin"] = new List<int>() { 6, 6, 3, 4, 6 };  
Console.WriteLine(string.Join(" ", grades["Kevin"]));
```

# Adding Elements to Multi-Dictionary

```
static void AddStudentGrade(  
    Dictionary<string, List<double>> grades,  
    string studentName, double grade)  
{  
    if (!grades.ContainsKey(studentName))  
        grades.Add(studentName, new List<double>());  
    grades[studentName].Add(grade);  
}  
  
AddStudentGrade(grades, "Peter", 6);  
AddStudentGrade(grades, "Maria", 5);
```

Ensure that the list  
of grades exist for  
the target student

# Problem: Average Student Grades

- Write a program to read student **names + grades**
- Print the **students + average grade** for each student

```
6
Barney 5.20
Melissa 5.50
Melissa 2.50
Ted 2.00
Melissa 3.46
Ted 3.00
```



```
Barney -> 5.20 (avg: 5.20)
Melissa -> 5.50 2.50 3.46 (avg: 3.82)
Ted -> 2.00 3.00 (avg: 2.50)
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1465#1>

# Solution: Average Student Grades (1)

```
var grades = new Dictionary<string, List<decimal>>();  
var n = int.Parse(Console.ReadLine());  
for (int i = 0; i < n; i++) {  
    var tokens = Console.ReadLine().Split();  
    var name = tokens[0];  
    var grade = decimal.Parse(tokens[1]);  
    if (!grades.ContainsKey(name))  
        grades[name] = new List<decimal>();  
    grades[name].Add(grade);  
}  
// continues on next slide...
```

Make sure the  
list is initialized

Add the grade  
into the list

# Solution: Average Student Grades (2)

```
foreach (var (name, studentGrades) in grades)
{
    var average = studentGrades.Average();
    Console.WriteLine($"{name} -> ");
    foreach (var grade in studentGrades)
        Console.Write($"{grade:f2} ");
    Console.WriteLine($"(avg: {average:f2})");
}
```

Europe	{Bulgaria → Sofia} {France → Paris} {Germany → Berlin}
Asia	{China → Beijing} {India → New Delhi}
Africa	{Nigeria → Abuja} {Kenya → Nairobi}

# Nested Dictionaries

Dictionaries Holding Other Dictionaries

# Nested Dictionaries

- A dictionary may hold another **dictionary** as value
- Example: population by country and city



BG



Sofia → 1,211,000  
Plovdiv → 338,000  
Varna → 335,000

UK



London → 8,674,000  
Manchester → 2,550,000

USA



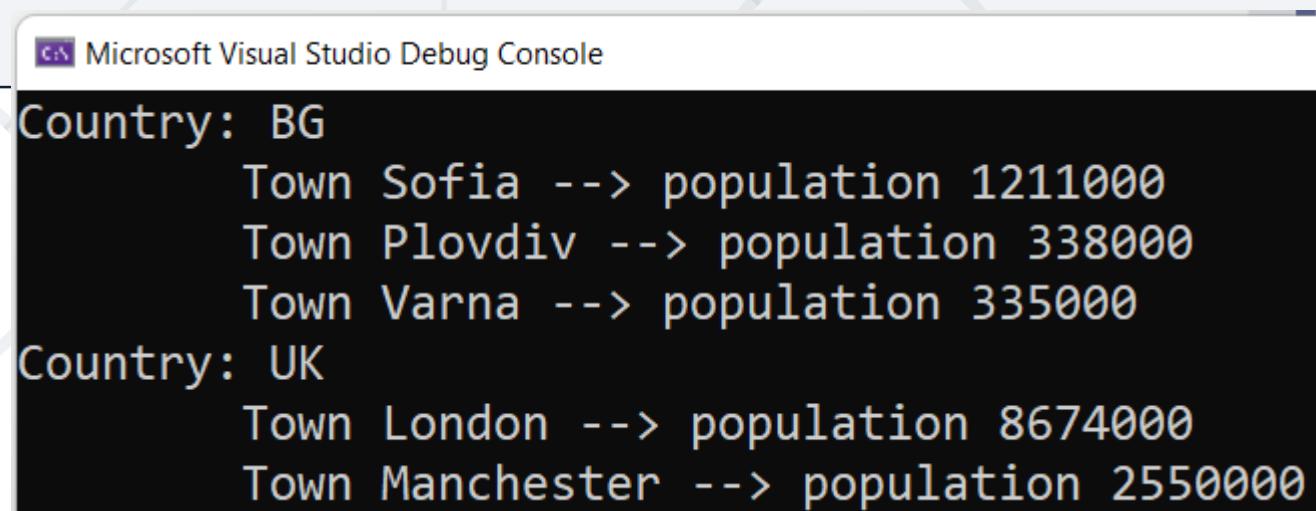
New York City → 8,406,000  
Washington → 659,000

# Nested Dictionaries: Initialization

```
var population = new Dictionary<string, Dictionary<string, int>> {  
    {"BG",  
        new Dictionary<string, int> {  
            { "Sofia", 1_211_000 },  
            { "Plovdiv", 338_000 },  
            { "Varna", 335_000 },  
        }  
    },  
    {"UK",  
        new Dictionary<string, int> {  
            { "London", 8_674_000 },  
            { "Manchester", 2_550_000 },  
        }  
    }, ...  
};
```

# Nested Dictionaries: Printing

```
foreach (var (countryName, towns) in population)
{
    Console.WriteLine("Country: " + countryName);
    foreach (var (townName, townPop) in towns)
        Console.WriteLine(
            $"\\tTown {townName} --> population {townPop}");
}
```



Microsoft Visual Studio Debug Console

```
Country: BG
    Town Sofia --> population 1211000
    Town Plovdiv --> population 338000
    Town Varna --> population 335000
Country: UK
    Town London --> population 8674000
    Town Manchester --> population 2550000
```

# Nested Dictionaries: Adding New Entry

```
AddPopulation("China", "Shanghai", 24_300_000);  
AddPopulation("China", "Beijing", 18_800_000);  
AddPopulation("China", "Shenzhen", 12_700_000);  
AddPopulation("BG", "Stara Zagora", 250_000);
```

```
void AddPopulation(string country, string town, int townPop)  
{  
    if (! population.ContainsKey(country))  
        population[country] = new Dictionary<string, int>();  
    population[country][town] = townPop;  
}
```

# Problem: Product Shop

- Write a program to keep information about **food shops**
  - The input holds triples: **{shop, product, price}**
  - If you receive an existing {shop + product}, **replace the price**
- Your output must be **ordered by shop name**

```
lidl, juice, 2.30
kaufland, banana, 1.10
lidl, grape, 2.20
```

Revision

End command

```
kaufland->
Product: banana, Price: 1.1
lidl->
Product: juice, Price: 2.3
Product: grape, Price: 2.2
```

# Solution: Product Shop (1)

```
var shops = new Dictionary<string, Dictionary<string, double>>();  
string line;  
while ((line = Console.ReadLine()) != "Revision")  
{  
    string[] productsInfo = line.Split(", ");  
    string shop = productsInfo[0];  
    string product = productsInfo[1];  
    double price = double.Parse(productsInfo[2]);  
    // continues on next slide...
```

# Solution: Product Shop (2)

```
if (!shops.ContainsKey(shop))  
{  
    shops.Add(shop, new Dictionary<string, double>());  
}  
shops[shop].Add(product, price);  
  
var orderedShops = shops.OrderBy(s => s.Key)  
.ToDictionary(x => x.Key, x => x.Value);  
// TODO: Print the ordered dictionary
```

Make sure the inner  
dictionary is initialized

# Problem: Cities by Continent and Country

- Write a program to read **continents**, **countries** and their **cities**, put them in a nested dictionary and print them

6

Europe Bulgaria Sofia  
Asia China Beijing  
Asia Japan Tokyo  
Europe Poland Warsaw  
Europe Germany Berlin  
Europe Poland Poznan



Europe:

Bulgaria -> Sofia

Poland -> Warsaw, Poznan

Germany -> Berlin

Asia:

China -> Beijing

Japan -> Tokyo

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1465#3>

# Solution: Cities by Continent and Country (1)

```
var continentsData =  
    new Dictionary<string, Dictionary<string, List<string>>>();  
  
var n = int.Parse(Console.ReadLine());  
  
for (int i = 0; i < n; i++) {  
    var tokens = Console.ReadLine().Split();  
    var continent = tokens[0];  
    var country = tokens[1];  
    var city = tokens[2];  
    // continues on next slide...
```

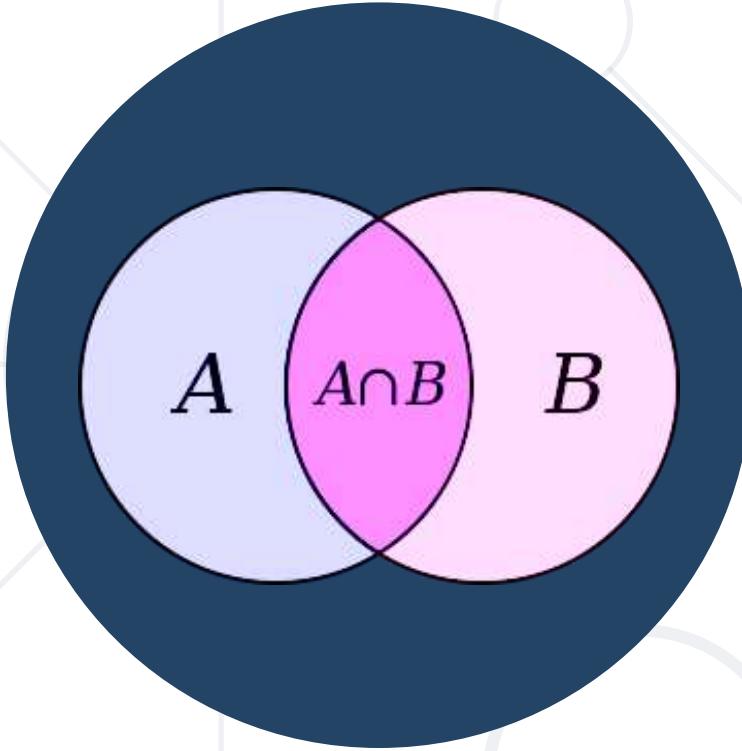
# Solution: Cities by Continent and Country (2)

```
if (!continentsData.ContainsKey(continent)) {           Initialize continent  
    continentsData[continent] = new Dictionary<string, List<string>>();  
}  
  
if (!continentsData[continent].ContainsKey(country)) {   Initialize cities  
    continentsData[continent][country] = new List<string>();  
}  
  
continentsData[continent][country].Add(city);  
}  
  
// continues on next slide...
```

Append the city  
to the country

# Solution: Cities by Continent and Country (3)

```
foreach (var (continentName, countries) in continentsData)
{
    Console.WriteLine($"{continentName}:");
    foreach (var (countryName, cities) in countries)
    {
        // TODO: Print each country with its cities
    }
}
```



**Set<T>**

HashSet<T> and SortedSet<T>

# Sets in C#

- A set keeps **unique elements**
  - Allows **add / remove / search** elements
  - Very **fast performance**
  - Example: Towns = {London, Tokyo, Paris, Rome}
- **HashSet<T>**
  - Keeps a set of elements in a **hash-table**
  - Elements are in **no particular order**
  - Similar to **List<T>**, but more efficient implementation



# HashSet<T> – Example

```
HashSet<string> set = new HashSet<string>();  
set.Add("Peter");  
set.Add("Peter"); // Existing element → not added again  
set.Add("George");  
Console.WriteLine(string.Join(", ", set)); // Peter, George  
Console.WriteLine(set.Contains("Maria")); // False  
Console.WriteLine(set.Contains("Peter")); // True  
set.Remove("Peter");  
Console.WriteLine(set.Count); // 1
```

# List<T> vs HashSet<T>

- List<T>
  - Fast "add", **slow** "search" and "remove" (pass through each element)
  - **Duplicates** are allowed
  - The insertion **order** is guaranteed
- HashSet<T>
  - **Fast** "add", "search" and "remove" thanks to **hash-table**
  - **No duplicates** are allowed
  - Does not guarantee the insertion **order**



# Problem: Record Unique Names

- Read a sequence of names and print only the **unique ones**

```
8  
Isabel  
Patrick  
Isabel  
Steven  
Patrick  
Alice  
Peter  
Patrick
```



```
Isabel  
Patrick  
Steven  
Alice  
Peter
```

```
9  
Lyle  
Lyle  
Bruce  
Alice  
Easton  
Shawn  
Alice  
Shawn  
Alice
```



```
Lyle  
Bruce  
Alice  
Easton  
Shawn
```

```
7  
Roki  
Roki  
Roki  
Roki  
Roki  
Roki  
Roki
```



```
Roki
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1465#4>

# Solution: Record Unique Names

```
var names = new HashSet<string>();
var n = int.Parse(Console.ReadLine());
for (int i = 0; i < n; i++)
{
    var name = Console.ReadLine();
    names.Add(name);
}
foreach (var name in names)
    Console.WriteLine(name);
```

HashSet stores  
unique values

Adds non-existing names only

# SortedSet<T>

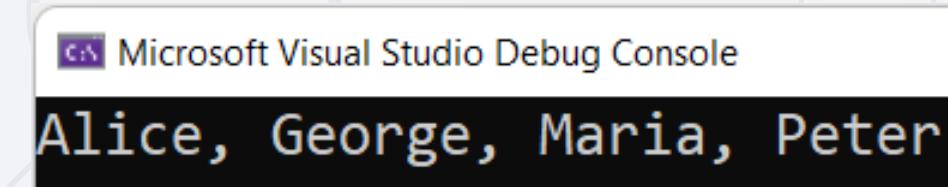
- **SortedSet<T>**
  - The elements are **ordered incrementally**



```
var set = new SortedSet<string>();

set.Add("Peter");
set.Add("Peter");
set.Add("George");
set.Add("Maria");
set.Add("Alice");

Console.WriteLine(string.Join(", ", set));
```

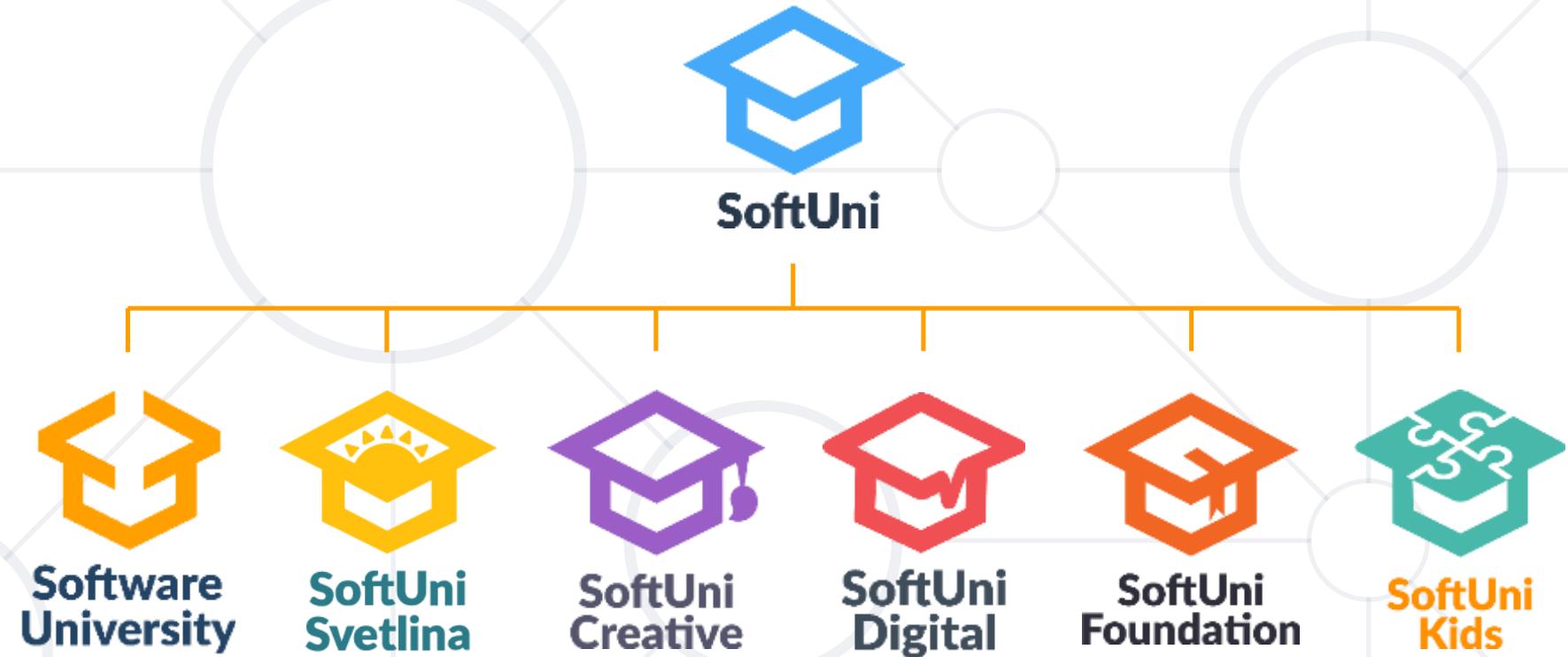
A screenshot of the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console area displays the output of the code execution: "Alice, George, Maria, Peter".

# Summary

- Multi-dictionaries allow **keeping a collection as a dictionary value**
- Nested dictionaries **allow keeping a dictionary as dictionary value**
- Sets allow keeping **unique values in unspecified order**
  - No duplicates
  - Fast add, search & remove



# Questions?



# SoftUni Diamond Partners

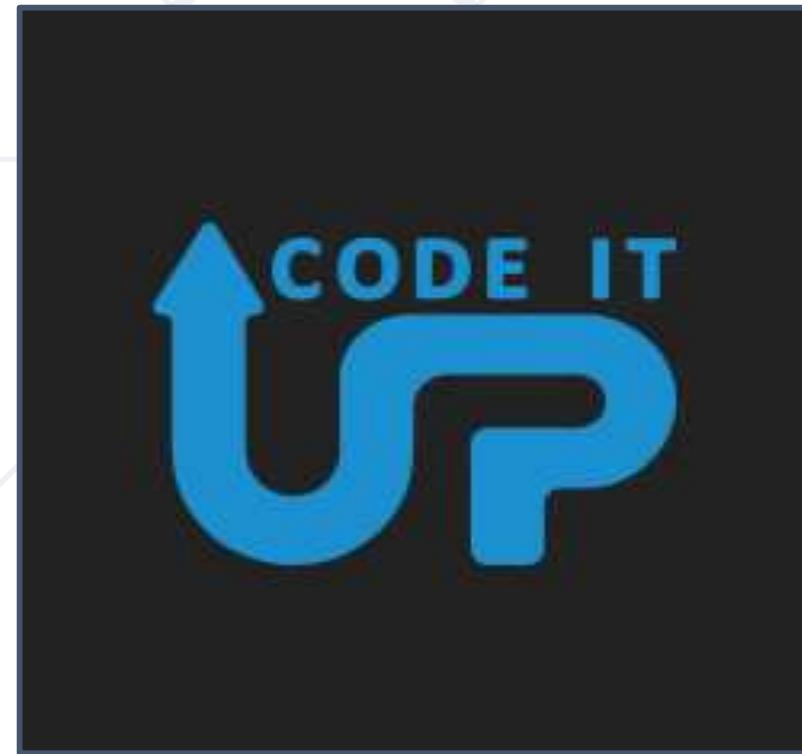


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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





# Table of Contents

1. What are **Streams**?
2. **Readers** and **Writers**
3. **File Streams**
4. **File Class**
5. **Directory Class**



Have a Question?



sli.do

#csharp-advanced

# Streams: Basic Concepts

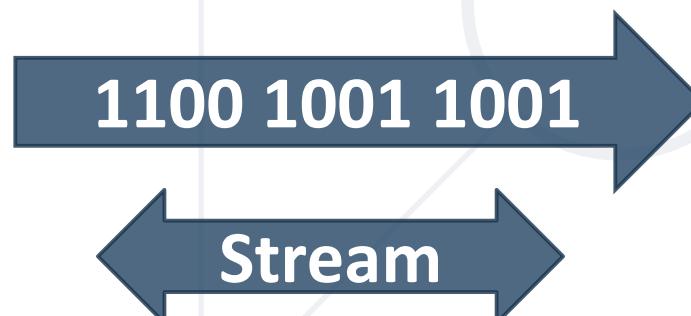
## What are Streams?



1000000100001011000110110001101100  
001001010110001011001001101000100011  
1100110011011001010010000010001100011  
01110001000001000010110001011000100011  
1000000100010101100010111000100010011

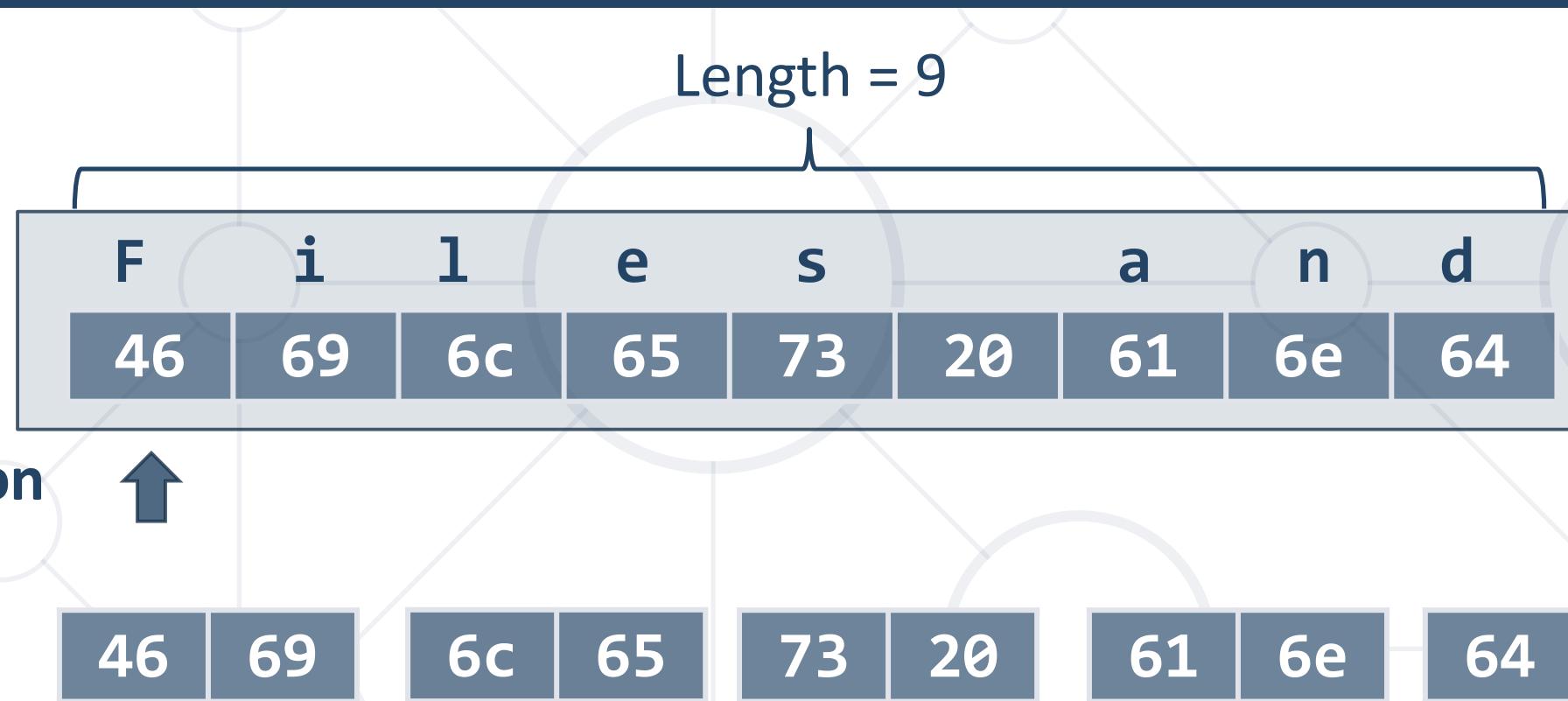
# What is a Stream?

- In programming, **streams** are used to **transfer data** between two endpoints (apps / devices / programs)
- Developers use a **stream** to:
  - **Read** (receive) data
  - **Write** (send) data



- Streams are means for **transferring** (reading and writing) **data**
  - Example: downloading a file from Internet uses streams
- Streams are ordered **sequences of bytes**
  - Provide **sequential** access to its elements (follow the FIFO rule)
- Different types of streams are access different data sources:
  - **File streams**, **network** streams, **memory** streams and others
- Typical use scenario: **open** a stream → **read / write** → **close**
- Streams use **buffering**: data is sent and comes in **chunks**

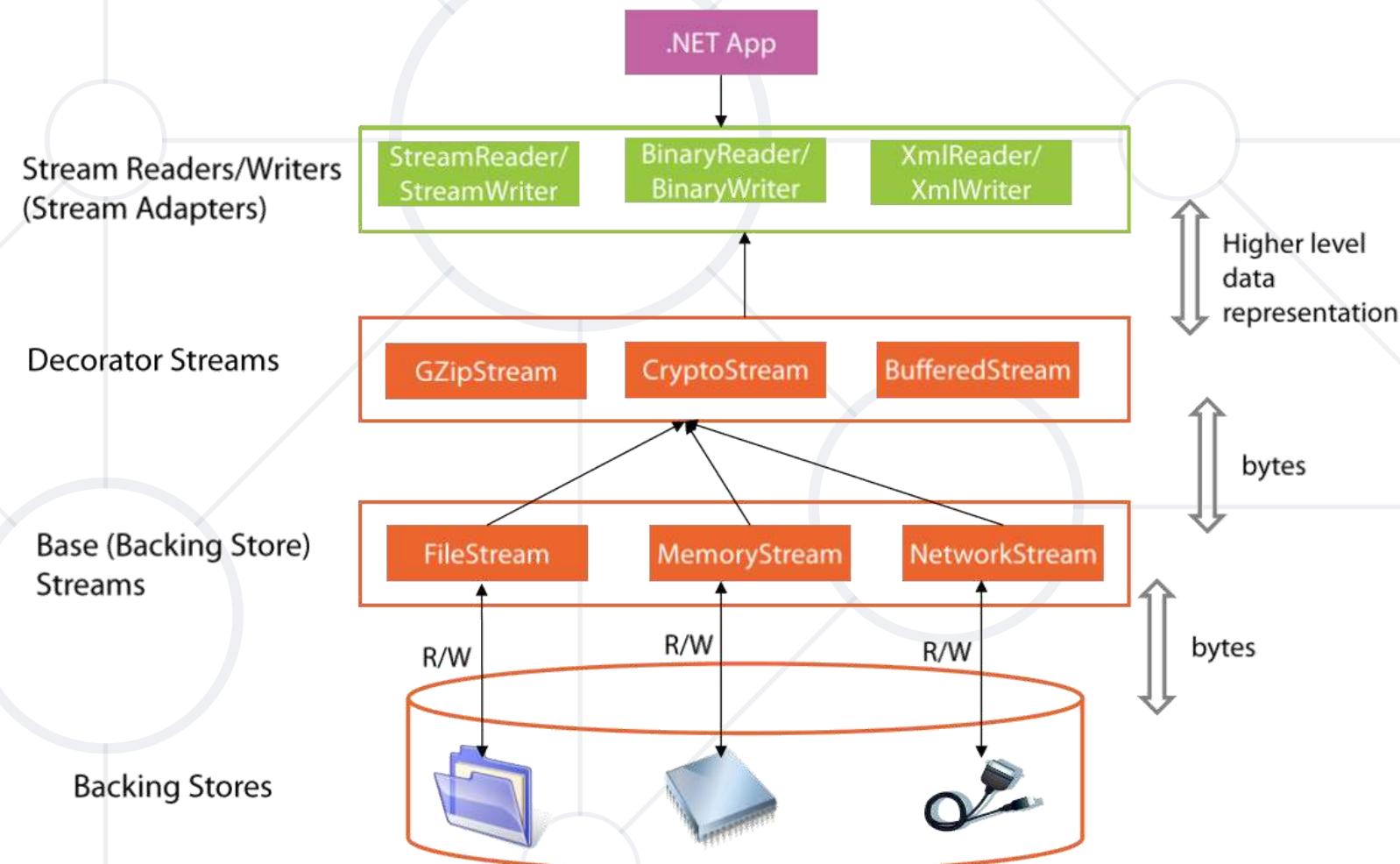
# Streams and Buffering – Example



- **Position** is the current offset from the stream start
- **Buffer** keeps **n** bytes of the stream from the current position

# Stream Types in .NET

## The Overall Architecture





# **Text Readers and Writers**

Readers and Writers in C#

# Using StreamReader

- **StreamReader** in C# reads **text** from a file / stream
- The **using(...)** statement closes properly the stream at the end



```
var reader = new StreamReader(fileName);
using (reader)
{
    // Use the reader here, e.g.
    string line = reader.ReadLine();
}
```

# Example: Reading a Text File

- Read the content from a text file **input.txt**
- Print on the console each **line number + line text** (start from 1)

First line

Second Line

Third line



1. First line

2. Second Line

3. Third line

# Example: Reading a Text File – C# Code

```
var reader = new StreamReader("../.../input.txt");
using (reader)
{
    int counter = 1;
    while (true)
    {
        string line = reader.ReadLine();
        if (line == null)
            break;
        Console.WriteLine(++counter + ". " + line);
    }
}
```

# Using StreamWriter

- **StreamWriter** in C# writes **text** to a file / stream
- The **using(...)** statement closes properly the stream at the end



```
var writer = new StreamWriter(fileName);
using (writer)
{
    // Use the writer here, e.g.
    writer.WriteLine("Some text");
}
```

# Problem: Odd Lines

- Read the content from a text file **input.txt**
- Print the **odd lines** in a text file **output.txt**
- Counting starts from **0**

Two households, both alike in dignity,  
In fair Verona, where we lay our scene,  
From ancient grudge break to new mutiny,  
Where civil blood makes civil hands unclean.



In fair Verona, where we lay our scene,  
Where civil blood makes civil hands unclean.

# Solution: Odd Lines

```
var reader = new StreamReader("input.txt");
using (reader) {
    int counter = 0;
    string line = reader.ReadLine();
    using (var writer = new StreamWriter("output.txt")) {
        while (line != null)
            if (counter % 2 == 1)
                writer.WriteLine(line);
        counter++;
        line = reader.ReadLine();
    }
}
```

# Problem: Line Numbers

- Read the text file **input.txt**
- Insert a **line number** in front of each line of the file
- Save the result in a text file **output.txt**



Two households, both alike in dignity,  
In fair Verona, where we lay our scene,  
From ancient grudge break to new mutiny,  
Where civil blood makes civil hands unclean.

1. Two households, both alike in dignity,
2. In fair Verona, where we lay our scene,
3. From ancient grudge break to new mutiny,
4. Where civil blood makes civil hands unclean.

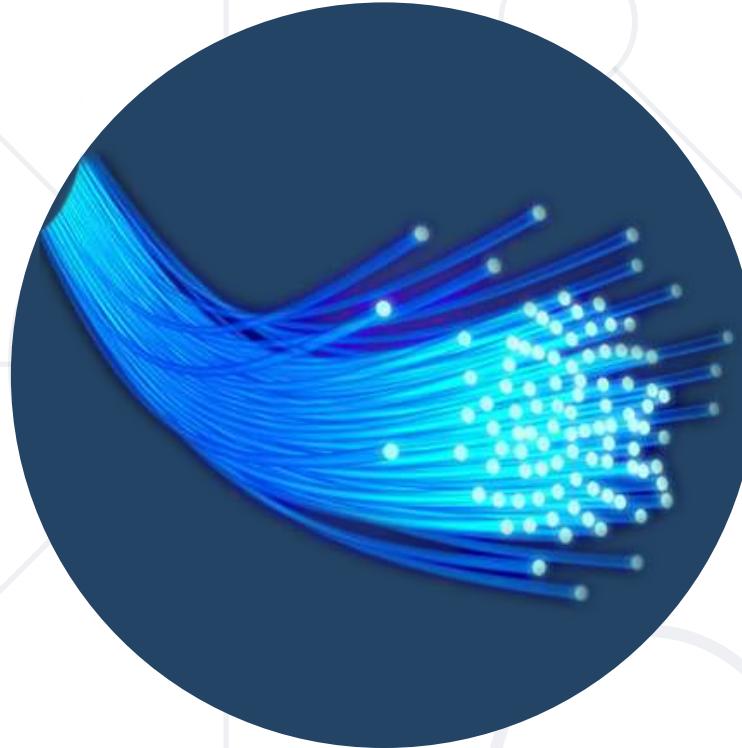
# Solution: Line Numbers

```
using (var reader = new StreamReader("input.txt"))
{
    string line = reader.ReadLine();
    int counter = 1;
    using (var writer = new StreamWriter("output.txt"))
        while (line != null)
    {
        writer.WriteLine($"{counter}. {line}");
        line = reader.ReadLine();
        counter++;
    }
}
```

# Open → Read → Close with Try-Catch-Finally

```
StreamReader reader = null;
int linesCount = 0;
try {
    reader = new StreamReader("input.txt");
    while (reader.ReadLine() != null)
        linesCount++;
    Console.WriteLine("Lines count: {0}", linesCount);
}
catch (Exception ex) {
    Console.Error.WriteLine("Error reading file: {0}", ex);
}
finally {
    if (reader != null) reader.Close();
}
```

Instead of **try-finally**, you  
can use **using(reader)**



# Base Streams in .NET

## System.IO.Stream

# The System.IO.Stream Class

- The base class for all streams is **System.IO.Stream**
  - Provides the basic read / write functionality
  - **Read(buffer)**, **Write(buffer, offset, count)**
- Some streams do not support read, write or positioning operations
  - Properties **CanRead**, **CanWrite** and **CanSeek** are provided
  - Streams which support positioning, have also the properties **Position** and **Length**

# Methods of System.IO.Stream Class (1)

- **int Read(byte[] buffer, int offset, int count)**
  - Read as many as **count** bytes from input stream, starting from the given **offset** position
  - Returns the number of read bytes or 0, if end of stream is reached
  - Can freeze for undefined time while reading at least 1 byte
  - Can read less than the claimed number of bytes

F	i	l	e	s	a	n	d	
46	69	6c	65	73	20	61	6e	64

# Methods of System.IO.Stream Class (2)

- **Write(byte[] buffer, int offset, int count)**
  - Writes a sequence of **count** bytes to an output stream, starting from the given **offset** position
  - Can freeze for undefined time, until it sends all bytes to their destination
- **Flush()**
  - Sends the internal buffers data to its destination (data storage, I/O device, etc.)

# Methods of System.IO.Stream Class (3)

- **Close()**
  - Calls **Flush()**
  - Closes the connection to the device (mechanism)
  - Releases the used resources
- **Seek(offset, SeekOrigin)**
  - Moves the position (if supported) with given offset towards the beginning, the end or the current position



# File Streams

Reading / Writing Binary Files

# File Streams

- **File streams** reads / writes sequences of bytes from a file
- **Creating** a new binary file:

```
using (var fs = new FileStream("file.bin", FileMode.Create))  
{  
    // Write to the file: fs.Write(byte[]) ...  
}
```

- **Opening** an existing file

```
using (var fs = new FileStream("file.bin", FileMode.Open))  
{    // Read from file or write to the file ... }
```

# Writing Text to File – Example

```
string text = "Кирилица";
var fileStream =
    new FileStream("log.txt", FileMode.Create);
using(fileStream)
{
    byte[] bytes = Encoding.UTF8.GetBytes(text);
    fileStream.Write(bytes, 0, bytes.Length);
}
```

Encoding.UTF8.GetBytes() returns  
the underlying bytes of the characters

# Encrypt / Decrypt File with XOR

```
using (var fin = new FileStream("example.png", FileMode.Open))  
using (var fout = new FileStream("example-encrypted.png", FileMode.Create))  
{  
    byte[] buffer = new byte[4096];  
    while (true)  
    {  
        int bytesRead = fin.Read(buffer);  
        if (bytesRead == 0) break;  
        const byte secret = 183;  
        for (int i = 0; i < bytesRead; i++)  
            buffer[i] = (byte)(buffer[i] ^ secret);  
        fout.Write(buffer, 0, bytesRead);  
    }  
}
```

Encrypting the read bytes  
with the constant parameter  
**secret** using XOR operator



# .NET API for Easily Working with Files

File Class in .NET

# Reading Text Files

- File.ReadAllText() → **string** - reads a text file at once

```
using System.IO;  
...  
string text = File.ReadAllText("file.txt");
```

- File.ReadAllLines() → **string[]** - reads a text file's lines

```
using System.IO;  
...  
string[] lines = File.ReadAllLines("file.txt");
```

# Writing Text Files

- Writing a **string** to a text file:

```
File.WriteAllText("output.txt", "Files are fun :));
```

- Writing a **sequence** of strings to a text file, at separate lines:

```
string[] names = { "peter", "irina", "george", "mary" };
File.WriteAllLines("output.txt", names);
```

- Appending additional text to an existing file:

```
File.AppendAllText("output.txt", "\nMore text\n");
```

# Reading / Writing Binary Files

- Writing a **byte[]** to a text file:

```
using System.IO;  
...  
byte[] bytesToWrite = { 0, 183, 255 };  
File.WriteAllBytes("output.txt", bytesToWrite);
```

- Reading a binary file into **byte[]**:

```
using System.IO;  
...  
byte[] bytesRead = File.ReadAllBytes("binaryFile.txt");
```



# .NET API for Working with Directories

Directory Class in .NET

# Basic Directory Operations

- Creating a directory (with all its subdirectories at the specified path), unless they already exists:

```
Directory.CreateDirectory("TestFolder");
```

- Deleting a directory (with its contents):

```
Directory.Delete("TestFolder", true);
```

- Moving a file or a directory to a new location:

```
Directory.Move("Test", "New Folder");
```

# Listing Directory Contents

- **GetFiles()**

- Returns the names of the files in the specified directory (including their paths)

```
string[] filesInDir = Directory.GetFiles("TestFolder");
```

- **GetDirectories()**

- Returns the names of the subdirectories (including their paths) in the specified directory

```
string[] subDirs = Directory.GetDirectories("TestFolder");
```

# Problem: Calculate Folder Size

- You are given a folder named **TestFolder**
- Calculate the **size of all files in the folder** (with its subfolders)
- Print the result in a file "**output.txt**" in **megabytes**

output.txt
5.16173839569092

# Solution: Calculate Folder Size

```
double sum = 0;

DirectoryInfo dir = new DirectoryInfo("TestFolder");
FileInfo[] infos = dir.GetFiles("*", SearchOption.AllDirectories);

foreach (FileInfo fileInfo in infos)
{
    sum += fileInfo.Length;
}

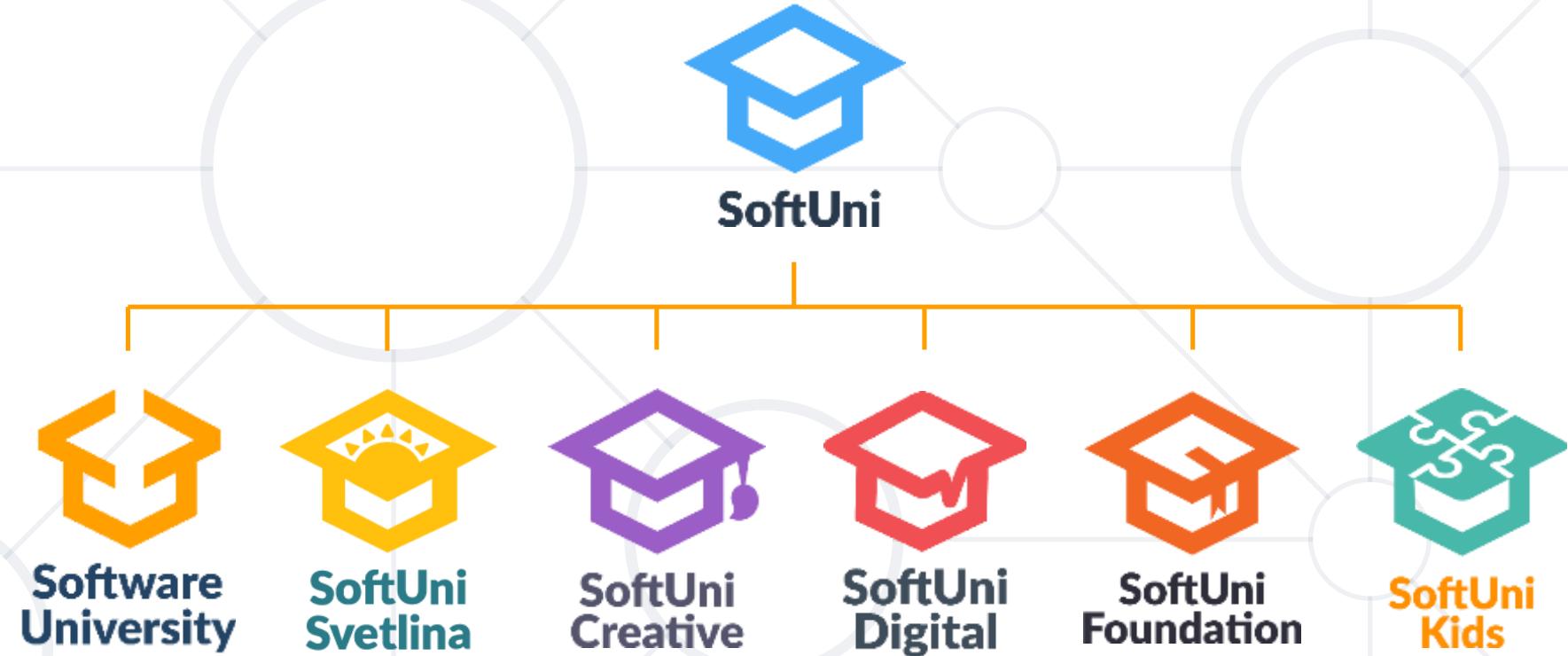
sum = sum / 1024 / 1024;

File.WriteAllText("output.txt", sum.ToString());
```

Gets all files from the given folder and its subfolders

- **Streams** are ordered sequences of bytes
  - Operations: **open** → **read / write** → **close**
  - Always close streams with **try-finally** or **using(...)**
- Use **StreamReader** / **StreamWriter** for text data
- Use **FileStream** to read / write binary files
- Use the **File** class to read / write files at once
- Use the **Directory** class to work with directories

# Questions?



# SoftUni Diamond Partners



SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Functional Programming in C#

Lambda Expressions, Functions, Actions and Delegate



SoftUni

SoftUni Team

Technical Trainers



Software University

<https://about.softuni.bg/>

# Table of Contents

## 1. Functional Programming: Concepts

## 2. Lambda Expressions in C#

## 3. Delegates, Functions, Actions, Predicates

- Func<T, TResult>, Action<T>, Predicate<T>

## 4. Higher-Order Functions

- Passing Functions to Methods

- Returning a Function from a Method



Have a Question?



sli.do

#csharp-advanced

 $f(x)$ 

# Functional Programming

## Paradigms and Concepts

# What is a Function?

- Mathematical functions

$$f(x) = x^2$$

Name

Input

Output

- A function is a calculation (expression or transformation), which maps **input values** to an **output value**
- In **programming** functions take **parameters**, perform some **work** and may return a **result**

X	f(x)
3	9
1	1
0	0
4	16
-4	16



# Functional Programming (FP)

- **Functional programming** (FP)
  - Programming by composing **pure functions**, avoiding **shared state**, **mutable data**, and **side-effects**
  - **Declarative** programming approach (not **imperative**)
    - Program state flows through pure functions
- **Pure function** == function, which returns **value only determined by its input**, without side effects
  - Examples: *sqrt(x)*, *sort(list)* → sorted list (new list)
  - Pure function == consistent result



f(x)

# Functional Programming – Examples

- Read several numbers and **find the biggest** of them (in C#)

- **Functional style**

```
Console.WriteLine(  
    Console.ReadLine()  
        .Split(" ")  
        .Select(int.Parse)  
        .Max()  
)
```

- Sequence of functional transformations

- **Imperative style**

```
var input = Console.ReadLine();  
var items = input.Split(" ");  
var nums = items.Select(int.Parse);  
var maxNum = nums.Max();  
Console.WriteLine(maxNum);
```

- Describes an **algorithm** (steps)

# Functional Programming Concepts

- Functional programming is **declarative**
  - Instead of statements, it makes use of expressions
  - **First-class functions**: functions can be stored in variables and passed as arguments

```
Func<int, int> twice = x => 2 * x;  
var d = twice(5); // 10
```

- **Higher-order functions**: either take other functions as arguments or return them as results

```
int aggregate(start, end, func) { ... }  
int sum = aggregate(1, 10, (a, b) => a + b); // 55
```



# Pure Functional Programming (Pure FP)

- Pure FP treats computation as the evaluation of mathematical functions, avoiding state and mutable data (**variables are immutable**)
- Always produce the same output with the same arguments disregard of other factors (**deterministic**)
  - No other input data besides the input parameters
  - The output value of a function **depends only on the arguments** that are passed to the function
- No **for** and **while** loops, instead, functional languages rely on **recursion** for iteration



# Functional Programming Languages

- **Purely functional languages** are **unpractical** and rarely used
  - The program is **pure function** without side effects, e.g. **Haskell**
- **Impure functional languages**
  - Emphasize functional style, but allow side effects, e.g. **Clojure**
- **Multi-paradigm languages**
  - Combine multiple programming paradigms:  
**functional, structured, object-oriented, ...**
  - Examples: **JavaScript, C#, Python, Java**



# Lambda Expressions in C#

Implicit / Explicit Lambda Expressions

# Lambda Expressions in C# (1)

- **Lambda expressions** are anonymous functions containing expressions and statements
- Lambda syntax in C#

**(parameters) => {body}**

  - Use the lambda operator "**=>**" (**goes to**)
  - Parameters can be enclosed in parentheses **()**
  - The body holds the expression or statement and can be enclosed in braces **{}**



# Lambda Expressions in C# (2)

- Implicit lambda expression

```
msg => Console.WriteLine(msg);
```

- Explicit lambda expression

```
(String msg) => { Console.WriteLine(msg); }
```

- Zero parameters

```
() => { Console.WriteLine("hi"); }
```

```
() => MyMethod();
```

- Multiple parameters

```
(int x, int y) => { return x + y; }
```

# Problem: Sort Even Numbers

- Read integers from the console
- Print the **even numbers**, sorted in ascending order
- Use two **lambda expressions**
- Examples:

4, 2, 1, 3, 5, 7, 1, 4, 2, 12



2, 2, 4, 4, 12

1, 3, 3, 4, 5, 6, 10, 9, 8, 2



2, 4, 6, 8, 10

1, 3, 4, 13, 10, 23, 45, 5, 1



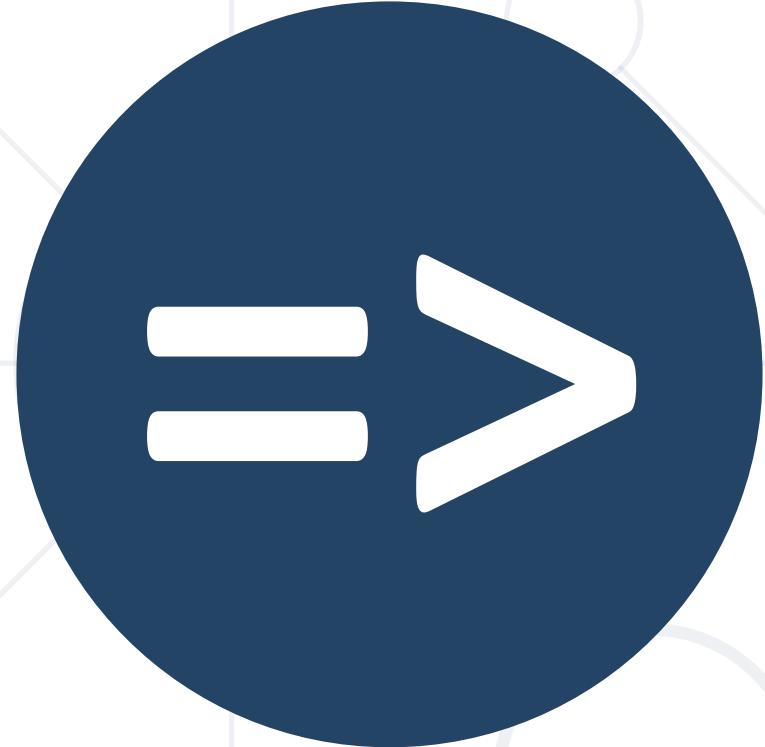
4, 10

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1472#0>

# Solution: Sort Even Numbers

```
int[] numbers = Console.ReadLine()
    .Split(new string[] { ", " },
        StringSplitOptions.RemoveEmptyEntries)
    .Select(n => int.Parse(n))
    .Where(n => n % 2 == 0)
    .OrderBy(n => n)
    .ToArray();

string result = string.Join(", ", numbers);
Console.WriteLine(result);
```



**Delegates, Functions, Actions, Predicates**

`Func<T, TResult>, Action<T>, Predicate<T>`

- A **delegate** in C# is a data type that **holds a method** with a certain parameter list and return type
  - Used to pass **methods as arguments** to other methods
- Can be used to define **callback methods**

```
public delegate int Combine(int x, int y);  
  
Combine multiply = (x, y) => x * y;  
Combine add = (x, y) => x + y;  
int mult = multiply(3, 5); // 15  
int sum = add(3, 5); // 8
```

# Generic Delegates: Func<T, TResult>

- Initialization of a function

Input type

Output type

Lambda expression

```
Func<int, string> func = n => n.ToString();
```

Name

Input parameter

Return expression

- Input and output type can be **different types**
- Input and output type **must be from the declared type**
- Func<...> delegate uses type parameters to define the number and types of input parameters and returns the type of the delegate

# Generic Delegates: Action<T>

- In .NET Action<T> is a void method:

```
private void Print(string message)
{ Console.WriteLine(message); }
```

- Instead of writing the method we can do:

```
Action<string> print =
message => Console.WriteLine(message);
```

- Then we use it like that:

```
print("Peter");           // Peter
print(5.ToString());     // 5
```

# Problem: Sum Numbers

- Read numbers from the console
- Use your own **function to parse** each element
- Print the **count** of numbers
- Print the **sum**

```
4, 2, 1, 3, 5, 7, 1, 4, 2, 12
```

10  
41

```
85, 47, 91, 32, 83, 75, 81, 2
```

8  
496

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1472#1>

# Solution: Sum Numbers

```
string input = Console.ReadLine();
Func<string, int> parser = n => int.Parse(n);
int[] numbers = input.Split(new string[] {"", },
    StringSplitOptions.RemoveEmptyEntries)
    .Select(parser).ToArray();
Console.WriteLine(numbers.Length);
Console.WriteLine(numbers.Sum());
```

# Generic Delegates: Predicate<T>

- In .NET Predicate<T> is a Boolean method:

```
Predicate<int> isNegative = x => x < 0;
```

```
Console.WriteLine(isNegative(5)); // false  
Console.WriteLine(isNegative(-5)); // true
```

```
var nums = new List<int> { 3, 5, -2, 10, 0, -3 };  
var negs = nums.FindAll(isNegative);  
Console.WriteLine(string.Join(", ", negs)); // -2, -3
```

# Problem: Count Uppercase Words

- Read a text from the console
- Filter only words, that **start** with a **capital** letter
- Use **Predicate<T>**
- Print each of the words on a new line

The following example shows how to use Predicate

The Predicate

Print count of words

Print

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1472#2>

# Solution: Count Uppercase Words

```
Predicate<string> checker = n => n[0] == n.ToUpper()[0];
string[] words = Console.ReadLine()
    .Split(" ", StringSplitOptions.RemoveEmptyEntries)
    .Where(w => checker(w))
    .ToArray();
foreach (string word in words)
{
    Console.WriteLine(word);
}
```

# Problem: Add VAT

- Read from the console **prices of items**
- Add **VAT** of 20% to all of them

1, 3, 5, 7



1.20  
3.60  
6.00  
8.40

10, 24, 12, 71

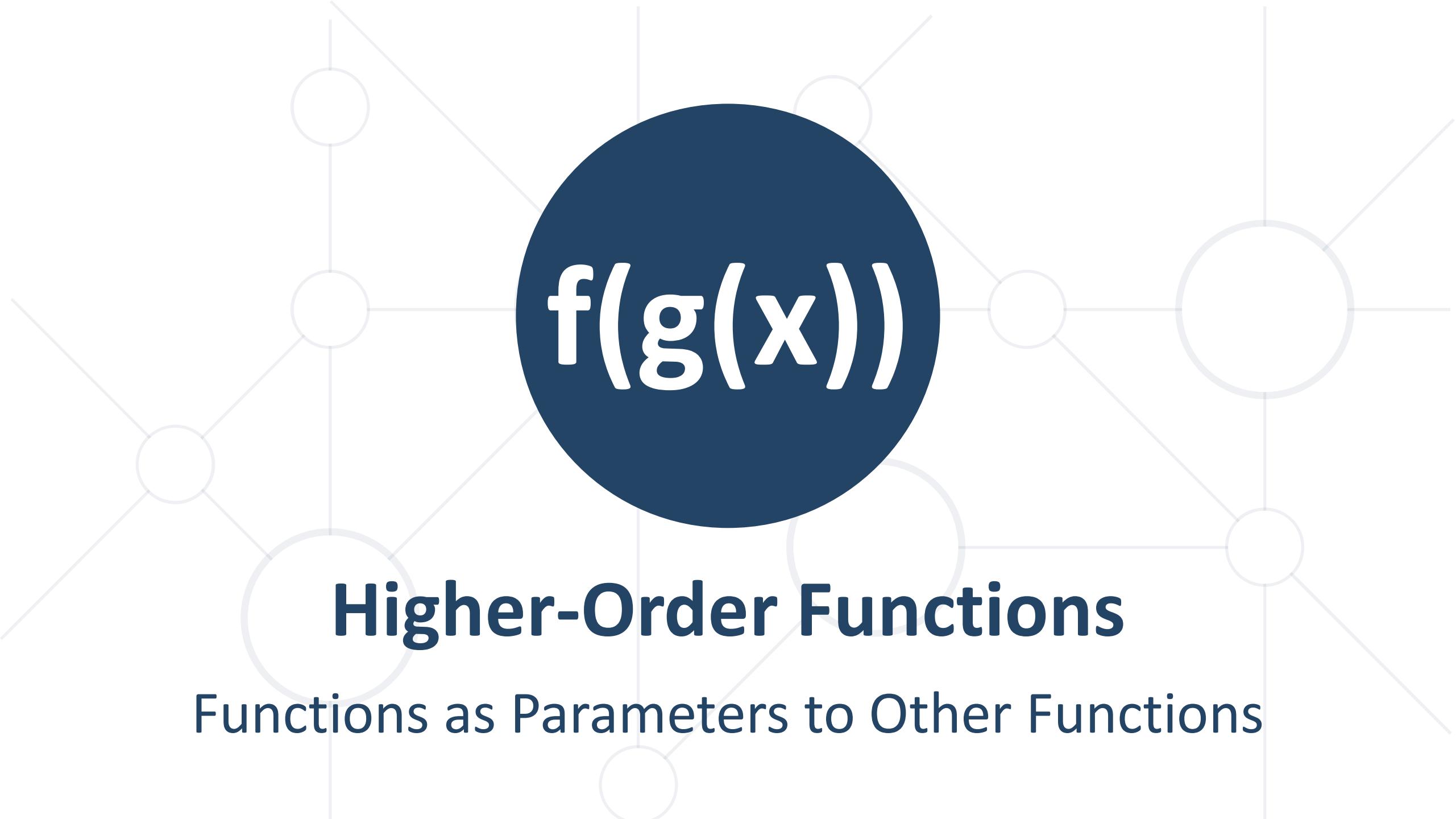


12.00  
28.80  
14.40  
85.20

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1472#3>

# Solution: Add VAT

```
double[] prices = Console.ReadLine()  
    .Split(new string[] { ", " },  
          StringSplitOptions.RemoveEmptyEntries)  
    .Select(double.Parse)  
    .Select(n => n * 1.2)  
    .ToArray();  
  
foreach (var price in prices)  
    Console.WriteLine($"{price:f2});
```

 $f(g(x)))$ 

## Higher-Order Functions

Functions as Parameters to Other Functions

# Higher-Order Functions

- We can pass **Func<T>** to methods:

```
private int Operation(int number, Func<int, int> operation)
{
    return operation(number);
}
```

- **Higher-order function:** take a function as parameter
- We pass **lambda function** to the higher-order function:

```
int a = 5;
int b = Operation(a, number => number * 5); // 25
int c = Operation(a, number => number - 3); // 2
int d = Operation(b, number => number % 2); // 1
```

# Higher-Order Functions: More Examples

```
long Aggregate(int start, int end, Func<long, long, long> op)
{
    long result = start;
    for (int i = start + 1; i <= end; i++)
        result = op(result, i);
    return result;
}
```

```
Aggregate(1, 10, (a, b) => a + b) // 55
```

```
Aggregate(1, 10, (a, b) => a * b) // 3628800
```

```
Aggregate(1, 10, (a, b) => long.Parse("" + a + b)) // 12345678910
```

# Problem: Filter by Age

- Read from the console **n people** (name + age)
- Read a **condition** (older, younger) and an **age filter**
- Read a **format pattern** for the output → print the filtered people

```
4
Lucas, 20
Mia, 29
Noah, 31
Simo, 16
older
20
name age
```



```
Lucas - 20
Mia - 29
Noah - 31
```

```
4
Lucas, 20
Noah, 18
Mia, 29
Simo, 16
younger
20
name
```



```
Noah
Simo
```

# Solution: Filter by Age (1)

```
List<Person> people = ReadPeople();
Func<Person, bool> filter = CreateFilter(condition, age);
Action<Person> printer = CreatePrinter(format);
PrintFilteredPeople(people, filter, printer);
```

```
public static Func<Person, bool> CreateFilter
    (string condition, int ageThreshold) {
    switch (condition) {
        case "younger": return x => x < ageThreshold;
        case "older": return x => x >= ageThreshold;
        default: throw new ArgumentException(condition);
    }
}
```

# Solution: Filter by Age (2)

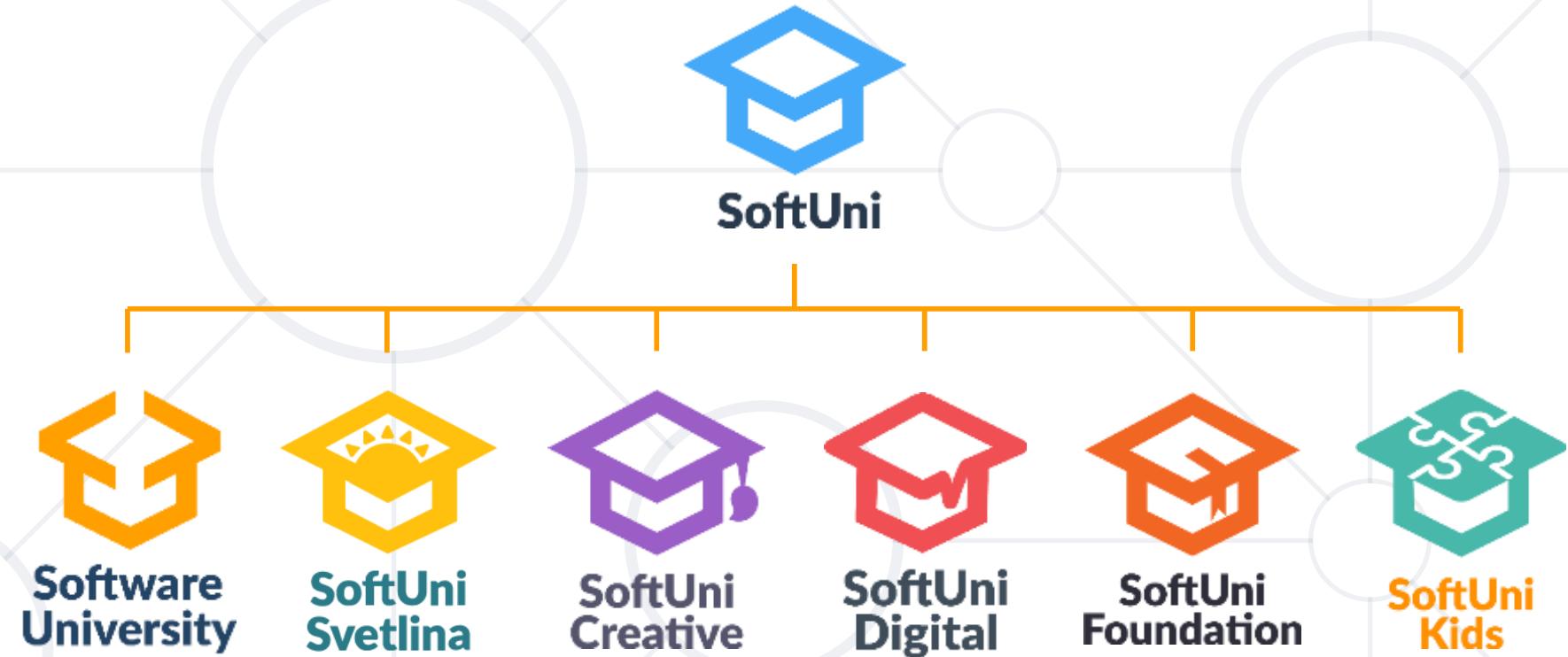
```
public static Action<Person> CreatePrinter(string format)
{
    switch (format)
        case "name":
            return person => Console.WriteLine($"{person.Name}");
    // TODO: complete the other cases
    default: throw new ArgumentException(format);
}
```

```
public static void PrintFilteredPeople(List<Person> people,
    Func<Person, bool> filter, Action<Person> printer) { ... }
```

- **Lambda expressions** are **anonymous functions**, often used with delegates
- **Func<T, TResult>** is a function that takes type **T** and returns **TResult** type
  - **Action<T>** is a void function (no return value)
  - **Predicate<T>** is a Boolean function
- Functions can be passed as **method parameters** and **returned as result** from a method invocation



# Questions?



# SoftUni Diamond Partners

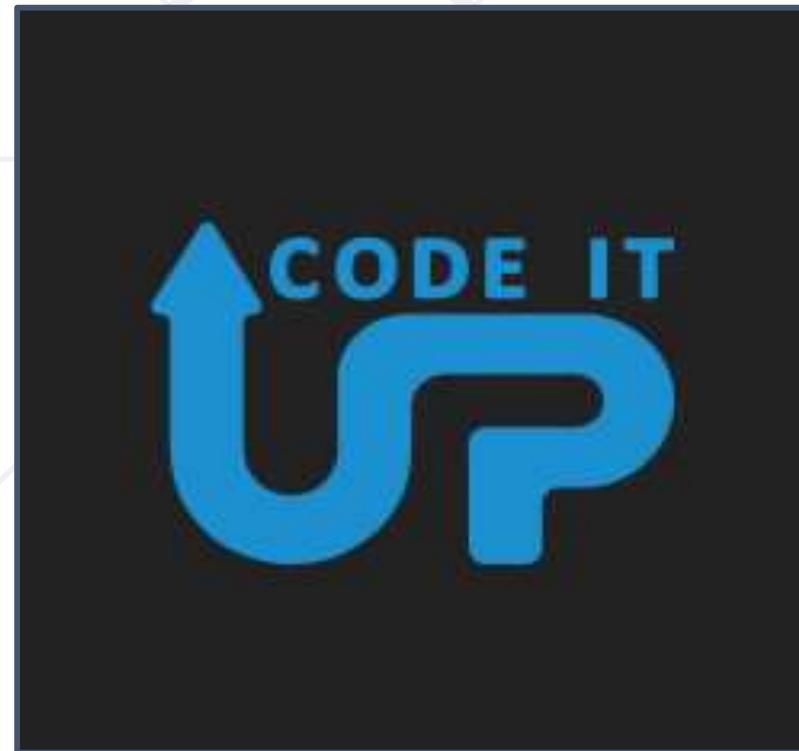


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

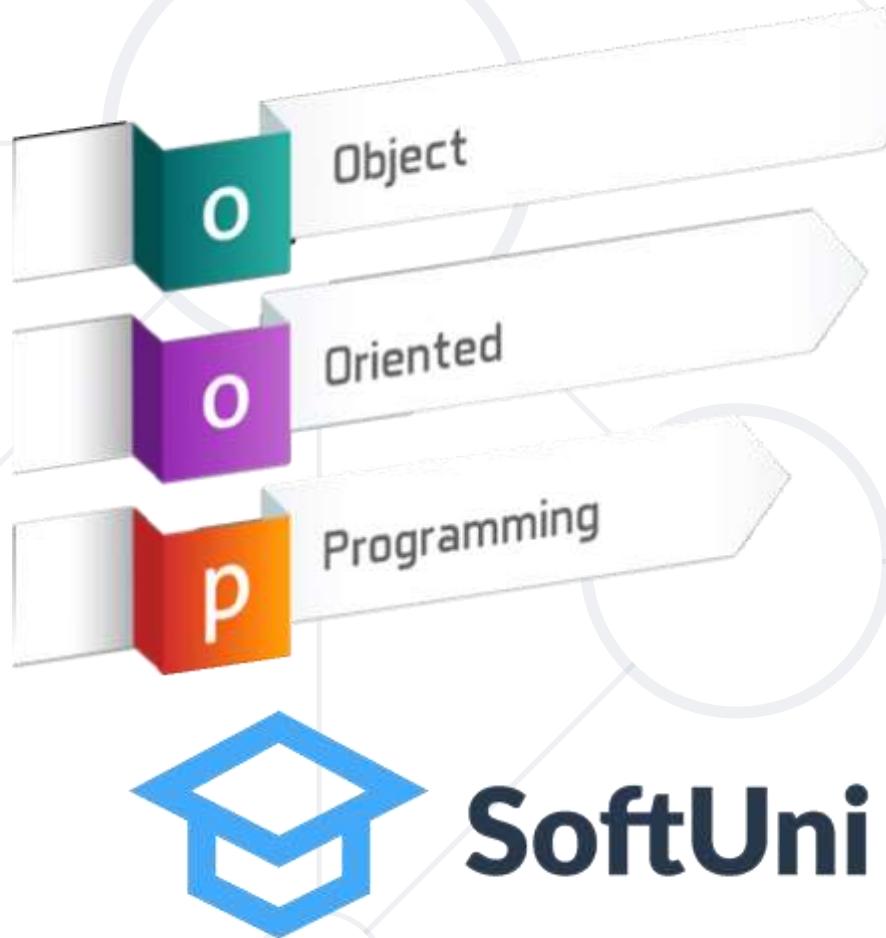


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Defining Classes

Classes, Fields, Constructors, Properties, Methods



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://about.softuni.bg/>

# Table of Contents

## 1. Defining Simple Classes

- Fields and Properties
- Methods
- Constructors

## 2. Enumerations

## 3. Static Classes

## 4. Namespaces

Have a Question?



sli.do

**#csharp-advanced**



# Defining Simple Classes

Creating Class for an ADT

# Defining Simple Classes

- Class is a **concrete implementation** of an ADT
- Classes provide **structure** for **describing** and **creating** objects

A large yellow lightbulb icon is positioned on the left side of the slide, partially obscured by a dark blue vertical bar.

Keyword

Class name

```
class Dice
```

{

...

}

Class body

# Naming Classes

- Name classes with nouns using **PascalCasing**
- Use **descriptive nouns**
- **Avoid abbreviations** (except widely known, e.g. URL, HTTP, etc.)

```
class Dice { ... }  
class BankAccount { ... }
```



```
class TPMF { ... }  
class bankaccount { ... }  
class intcalc { ... }
```



# Class Members

- **Members** are **declared** in the class and they have certain accessibility, which can be specified
- They can be:
  - Fields
  - Properties
  - Methods
  - Etc.

```
class Dice
{
    int sides;
    string Sides { get; }
    void Roll() { ... }
}
```

Field

Property

Method

# Creating an Object

- A class can have **many instances** (objects)

```
class Program
{
    public static void Main()
    {
        Dice diceD6 = new Dice();
        Dice diceD8 = new Dice();
    }
}
```

Use the **new keyword**

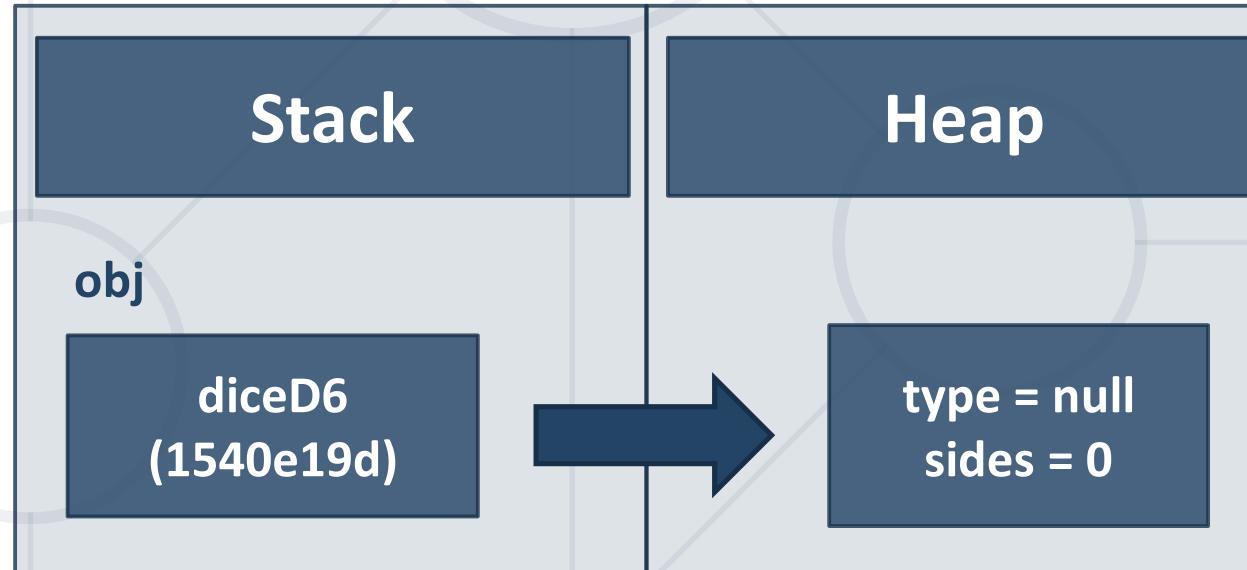
A variable holds an  
object **reference**



# Object Reference

- Declaring a variable creates a **reference** in the stack
- The **new** keyword allocates memory on the heap

```
Dice diceD6 = new Dice();
```



# Classes vs. Objects

- Classes provide **structure** for describing and creating objects
- An **object** is a **single instance** of a class



# Classes vs. Objects

- **Classes** provide structure for creating objects

```
class Dice
type: string
sides: int
Roll(...)
```

Class name

Class data

Class actions

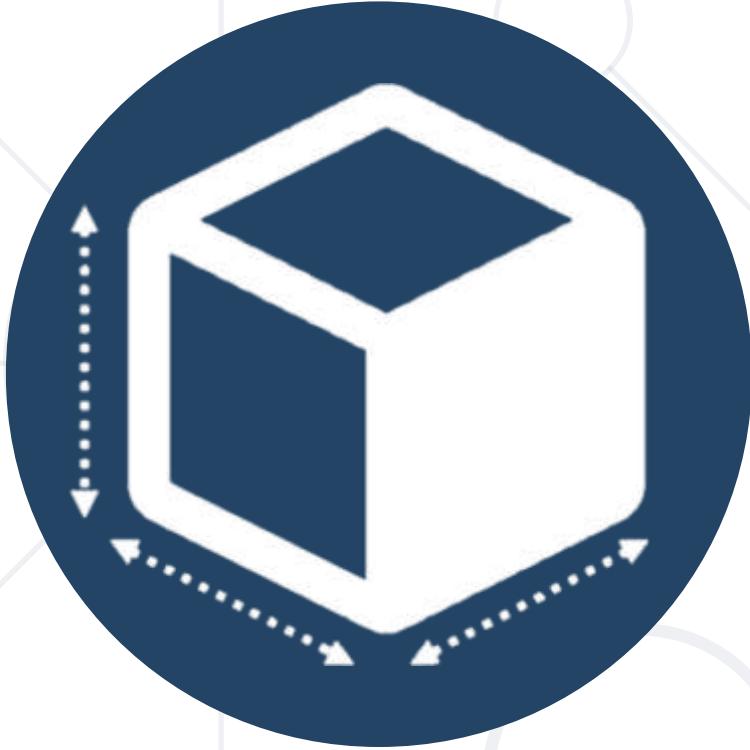
- An **object** is a single instance of a class

```
object diceD6
type = "six sided"
sides = 6
```

Object name

Object data





# Fields and Properties

Storing Data Inside a Class

# Fields and Modifiers

- Class **fields** have type and name
- Access modifiers (like **public** / **private**) define accessibility

Class modifier

Fields should  
always be private

Fields can be  
of **any type**

```
public class Dice
{
    private string type;
    private int sides;
    private int[] rollFrequency;
    private Person owner;
    public void Roll () { ... }
}
```

# Properties

- Used to create accessors and mutators (**getters** and **setters**)

```
public class Dice
{
    private int sides;
    public int Sides
    {
        public get { return this.sides; }
        public set { this.sides = value; }
    }
}
```

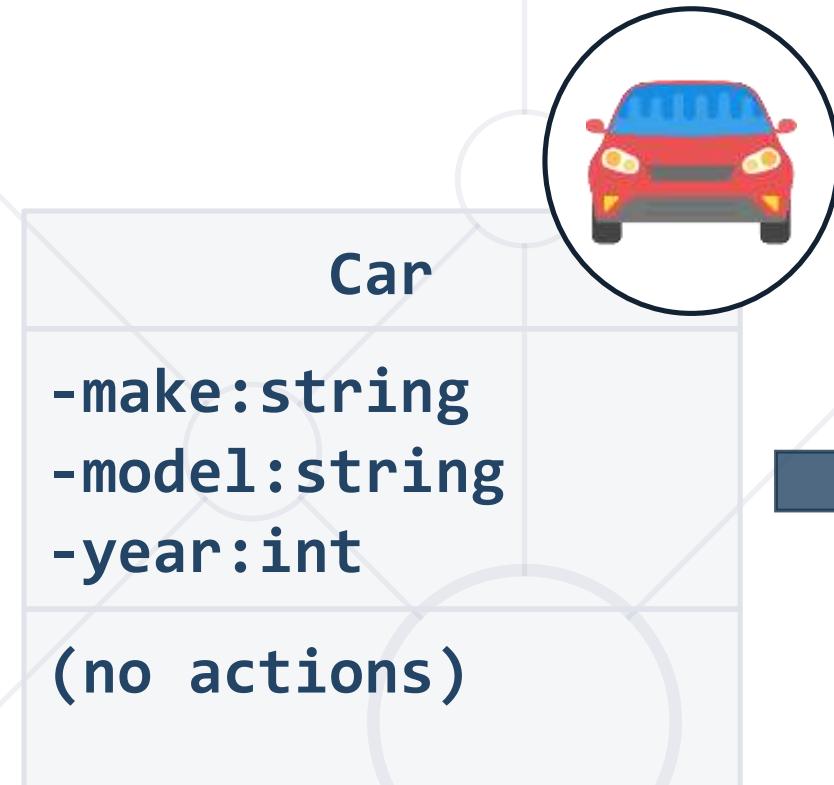
The field is hidden

The getter provides access to the field

The setter provides field change

# Problem: Car

- Create a class **Car**



```
private string make;  
private string model;  
private int year;  
public string Make  
{  
    get { return this.make; }  
    set { this.make = value; }  
}  
// TODO: Model and Year Getter & Setter
```



# Methods

Defining a Class Behaviour

# Methods

- Store executable code (an algorithm)

```
public class Dice
{
    private int sides;
    private Random rnd = new Random();
    public int Roll() {
        int rollResult = rnd.Next(1, this.sides + 1);
        return rollResult;
    }
}
```

this points to the  
current instance

# Problem: Car Extension

- Create a class **Car**

```
class Car {  
    -make:string  
    -model:string  
    -year:int  
    -fuelQuantity:double  
    -fuelConsumption:double  
  
    +Drive(double distance):void  
    +WhoAmI():string}
```



Check your solution here: <https://judge.softuni.bg/Contests/1478/Defining-Classes-Lab>

# Solution: Car Extension (1)

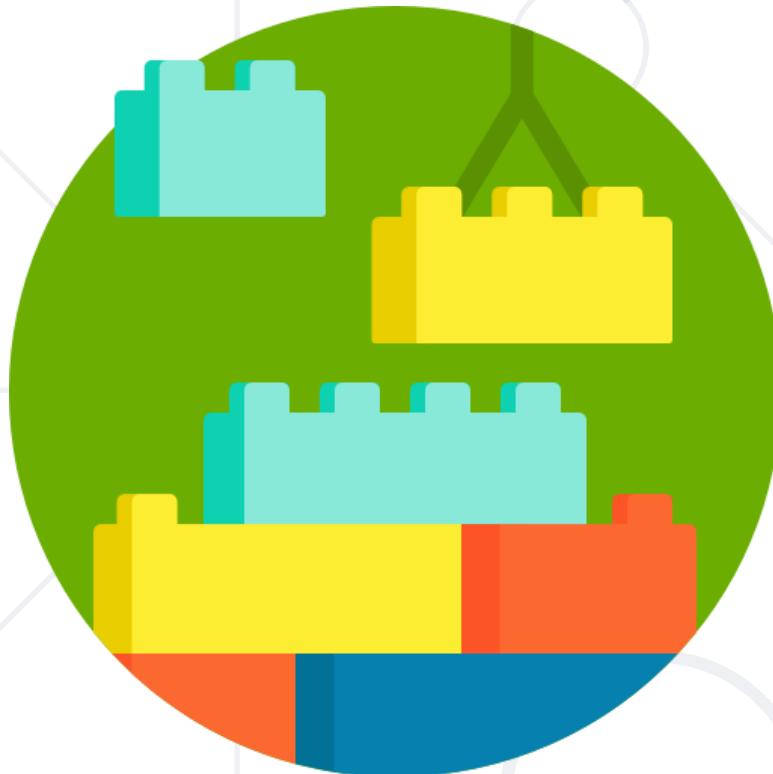
```
// TODO: Get the other fields from previous problem
private double fuelQuantity;
private double fuelConsumption;
// TODO: Get the other properties from previous problem
public double FuelQuantity {
    get { return this.fuelQuantity; }
    set { this.fuelQuantity = value; }
}
public double FuelConsumption {
    get { return this.fuelConsumption; }
    set { this.fuelConsumption = value; }}
```

# Solution: Car Extension (2)

```
public void Drive(double distance)
{
    bool canContinue = this.FuelQuantity - (distance *
                                              this.FuelConsumption) >= 0;
    if (canContinue)
        this.FuelQuantity -= distance * this.FuelConsumption;
    else
        Console.WriteLine("Not enough fuel to perform this trip!");
}
```

# Solution: Car Extension (3)

```
public string WhoAmI()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine($"Make: {this.Make}");
    sb.AppendLine($"Model: {this.Model}");
    sb.AppendLine($"Year: {this.Year}");
    sb.Append($"Fuel: {this.FuelQuantity:F2}L");
    return sb.ToString();
}
```



# Constructors

## Object Initialization

# Constructors

- When a constructor is invoked, it creates an instance of its class and usually initializes its members
- Classes in C# are instantiated with the keyword **new**



```
public class Dice
{
    public Dice() { }
}
```

```
public class StartUp
{
    static void Main()
    {
        var dice = new Dice();
    }
}
```

# Object Initial State

- Constructors **set object's initial state**

```
public class Dice
{
    int sides;
    int[] rollFrequency;
    public Dice(int sides) {
        this.sides = sides;
        this.rollFrequency = new int[sides];
    }
}
```

Always ensure  
correct state

# Multiple Constructors

- You can have multiple constructors in the same class

```
public class Dice  
{  
    private int sides;  
    public Dice() { }  
    public Dice(int sides)  
    {  
        this.sides = sides;  
    }  
}
```

Constructor without parameters

Constructor with parameters

# Constructor Chaining

- Constructors can call each other

```
public class Person {  
    private string name;  
    private int age;  
    public Person() {  
        this.age = 18;  
    }  
    public Person(string name) : this()  
    {  
        this.name = name;  
    }  
}
```

Calls default constructor

# Problem: Car Constructors

- Extend the previous problem and **create 3 constructors**
- Default values are:
  - Make - VW
  - Model - Golf
  - Year - 2025
  - FuelQuantity = 200
  - FuelConsumption = 10

```
Car
+Car()
+Car(string make, string model,
int year)
+Car(string make, string model,
int year, double fuelQuantity,
double fuelConsumption)
```

# Solution: Car Constructors (1)

```
public Car() {  
    this.Make = "VW";  
    this.Model = "Golf";  
    this.Year = 2025;  
    this.FuelQuantity = 200;  
    this.FuelConsumption = 10;}  
public Car(string make, string model, int year) : this()  
{  
    this.Make = make;  
    this.Model = model;  
    this.Year = year;}
```

# Solution: Car Constructors (2)

```
public Car(string make, string model, int year,  
double fuelQuantity, double fuelConsumption)  
: this(make, model, year)  
{  
    this.FuelQuantity = fuelQuantity;  
    this.FuelConsumption = fuelConsumption;  
}
```

# Problem: Car Engine and Tires

- Create the two classes and extend the Car class

Engine

**-horsePower:int**

**-cubicCapacity:double**

**+Engine(int horsePower,  
double cubicCapacity)**

Tire

**-year:int**

**-pressure:double**

**+Tire(int year,  
double pressure)**

Car

**+Car(string make, string model, int year,  
double fuelQuantity, double fuelConsumption,  
Engine engine, Tire[] tires)**

# Solution: Car Engine and Tires (1)

```
private int horsePower;  
  
private double cubicCapacity;  
  
public Engine(int horsePower, double cubicCapacity) {  
    this.HorsePower = horsePower;  
    this.CubicCapacity = cubicCapacity; }  
  
public int HorsePower {  
    get { return this.horsePower; }  
    set { this.horsePower = value; } }  
  
public double CubicCapacity {  
    get { return this.cubicCapacity; }  
    set { this.cubicCapacity = value; } }
```

# Solution: Car Engine and Tires (2)

```
private int year;  
private double pressure;  
public Tire(int year, double pressure) {  
    this.Year = year;  
    this.Pressure = pressure; }  
public int Year {  
    get { return this.year; }  
    set { this.year = value; }}  
public double Pressure {  
    get { return this.pressure; }  
    set { this.pressure = value; }}
```

# Solution: Car Engine and Tires (3)

```
public Car(string make, string model, int year,  
double fuelQuantity, double fuelConsumption, Engine engine,  
Tire[] tires)  
    : this(make, model, year, fuelQuantity, fuelConsumption)  
{  
    this.Engine = engine;  
    this.Tires = tires;  
}
```



# Enumerations

## Syntax and Usage

# Enumerations (1)

- Represent a numeric value from a fixed set as a text
- We can use them to pass **arguments** to **methods** without making code confusing

```
enum Day { Mon, Tue, Wed, Thu, Fri, Sat, Sun }
```

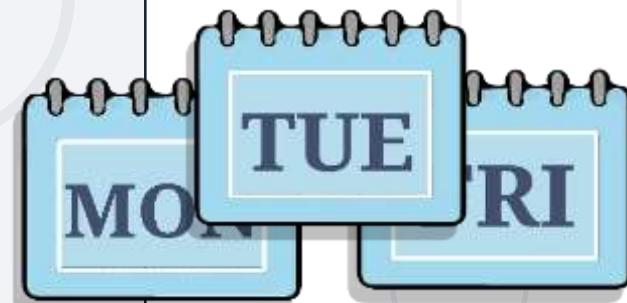
GetDailySchedule(0) → GetDailySchedule(Day.Mon)
- By default **enums** start at 0
- Every next value is incremented by 1



# Enumerations (2)

- We can **customize enum values**

```
enum Day {  
    Mon = 1,  
    Tue, // 2  
    Wed, // 3  
    Thu, // 4  
    Fri, // 5  
    Sat, // 6  
    Sun // 7  
}
```



```
enum CoffeeSize {  
    Small = 100,  
    Normal = 150,  
    Double = 300  
}
```





# Static Classes

## Static Class Members

# Static Class

- A **static** class is declared by the **static** keyword
- It **cannot** be **instantiated**
- You **cannot declare** variables from its **type**
- You access its **members** by using the **its name**



```
double roundedNumber = Math.Round(num);
int absoluteValue = Math.Abs(num);
int pi = Math.PI;
```

# Static Members (1)

- Both **static** and **non-static** classes can contain **static** members:
  - Methods, fields, properties, etc.
- A **static member** is **callable** on a class even when no instance of the class has been created
- Accessed by the **class'** name, not the **instance** name
- Only **one copy** of a static member **exists**, regardless of how many **instances** of the class are created



# Static Members (2)

- Static methods can be overloaded but not overridden
- A **const field** is essentially **static** in its **behavior** and it belongs to the **type, not the instance**
- Static members are initialized **before** the static member is **accessed for the first time** and **before** the static **constructor**

```
Bus.Drive();  
int wheels = Human.NumberOfWheels;
```

# Example: Static Members

```
public class Engine
{
    public static void Run() {
        Console.WriteLine("This is a static method");
    }
}
```

```
public static void Main() {
    Engine.Run();
}
// Output: This is a static method
```



# Namespaces

Definition and Usage

# Namespaces

- Used to organize classes
- The **using** keyword allows us not to write their names
- Declaring your own namespaces can help you control the scope of class and method names

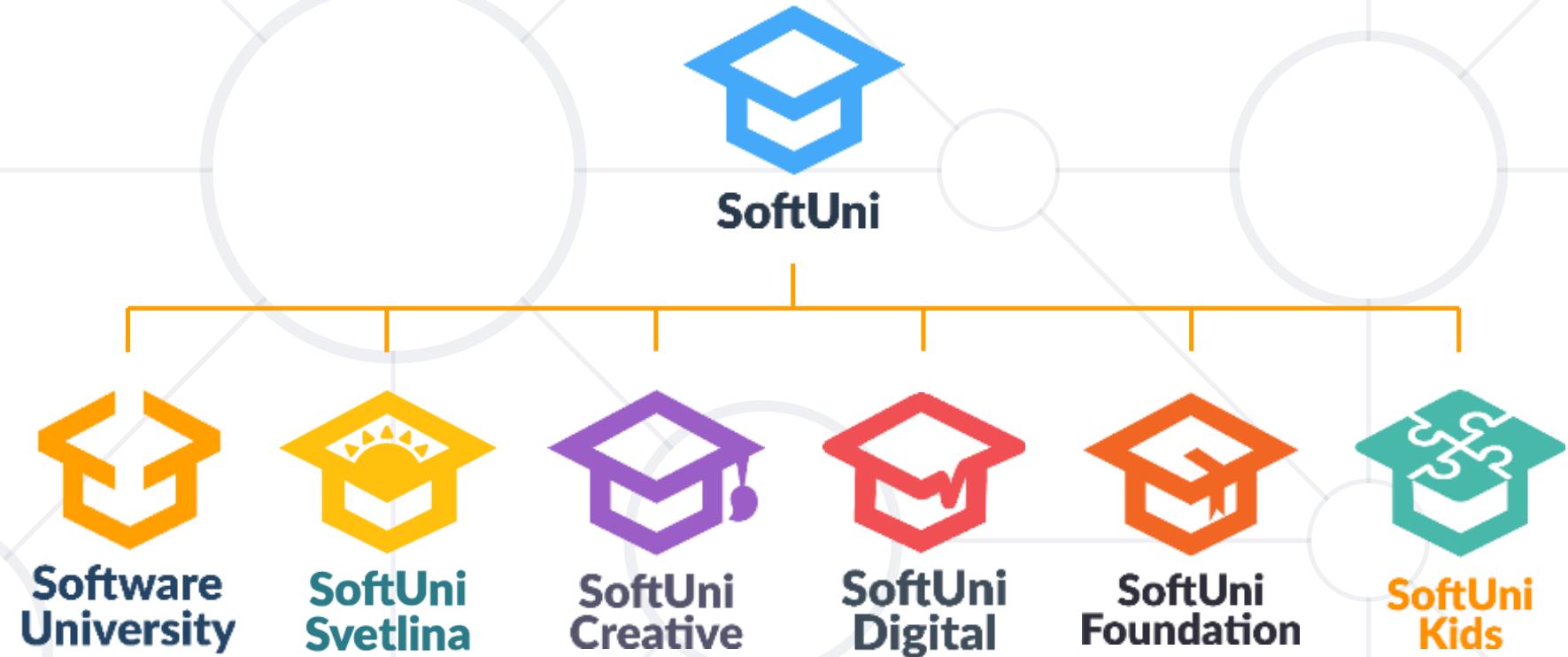


```
System.Console.WriteLine("Hello world!");  
var list = new  
    System.Collections.Generic.List<int>();
```

- Classes define **structure** for objects
- Objects are **instances of a class**
- Classes define **fields, methods, constructors** and other members
- Constructors:
  - **Invoked** when creating **new instances**
  - **Initialize** the **object's state**



# Questions?



# SoftUni Diamond Partners

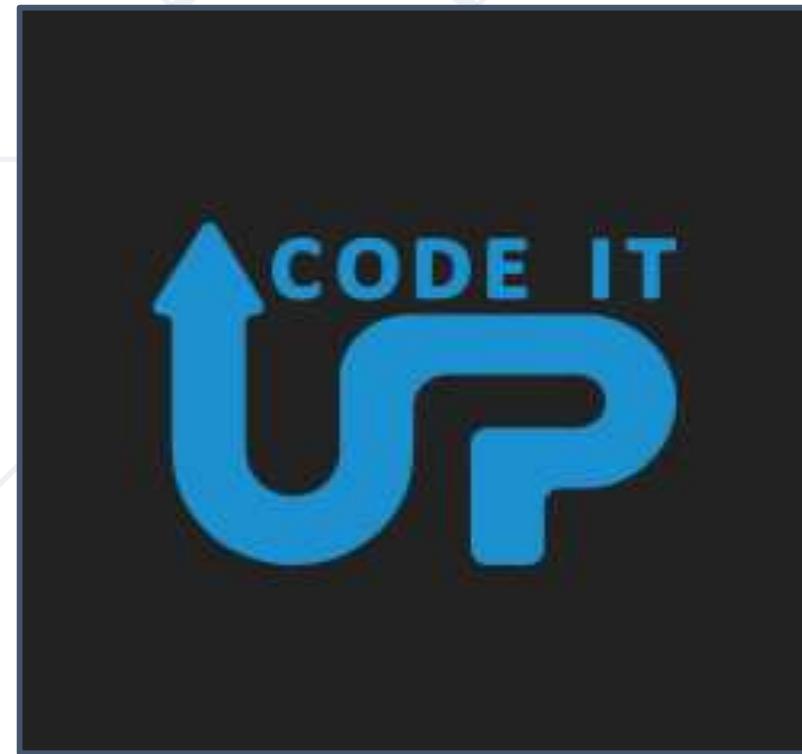


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Generics

Adding Type Safety and Code Reusability

SoftUni Team

Technical Trainers



SoftUni



Software University

<https://about.softuni.bg/>

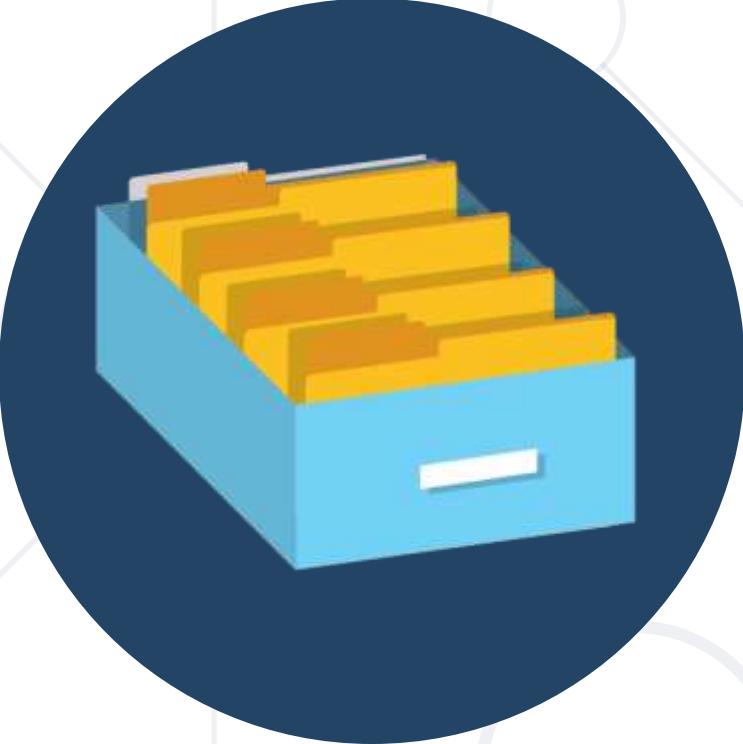
# Table of Contents

1. Generics
2. Generic Classes
3. Generic Methods
4. Generic Constraints



sli.do

# #csharp-advanced



# Generics

Definition, Type Parameters and Safety

# What Are Generics?

- Generics introduce the concept of **Type Parameters**
- Allow designing classes and methods without **parameter type specification**
- A generic **class** or a **method** accepts a certain type when it is **instantiated** by client code



```
public class CustomStack<T> { }
CustomStack<int> =
    new CustomStack<int>();
```

# Generics - Type Safety

- Add **type safety** for the client
- Provide a powerful way to **reuse** code

```
List<int> strings = new List<int>();  
List<Person> people = new List<Person>();
```

- Example: we need a collection that will store only strings

```
List<string> strings = new List<string>();  
strings.Add(3); // Compile time error
```

# Type Parameters

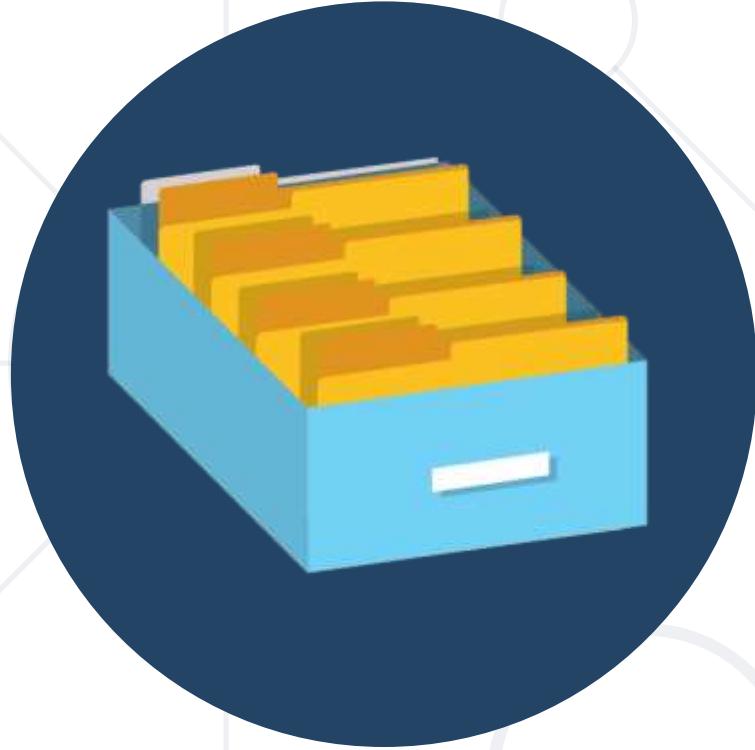
- Blueprint for a **type** - **T** (**Type Parameter**)
- You can use it **anywhere** inside the **generic** class

```
class List<T>
{
    public Add (T element) {...}

    public T Remove () {...}

    public T Peek { get; }

}
```



# Generic Classes

# Non-Generic Classes (1)

```
public class ObjectList
{
    private object[] elements;
    public ObjectList ()
    { this.elements = new object[4]; }
    public void Add(object value){}
    public object Get(int index)
    {
        return this.elements[index];
    }
}
```

# Non-Generic Classes (2)

```
var objectList = new ObjectList();
objectList.Add(1);
objectList.Add(new Customer());
objectList.Add(new Account());
```

```
var firItem = objectList[0]; // firItem is object
var secItem = (Customer)objectList[1]; // cast
```

- Encapsulate operations to a **non-particular data type**
- Defined with **Type Parameters - T**

```
class List<T> { }  
class Stack<T> { }
```

- Most commonly used are **generic collections**:
  - Linked Lists, Hash tables, Stacks, Queues, Trees, etc.
  - Collections with **multiple type parameters** – Dictionary<T, V>



# Generic Methods

# Non-Generic Methods

- Take a **certain** input and a **certain** output **type**

```
public class CustomerList
{
    public Customer Remove(Customer customer)
    {
        return removedCustomer;
    }
}
```

# Generic Methods

- Take **generic input** and return **generic output**

```
public List<T> CreateList<T>(T item)
{
    List<T> list = new List<T>();
    ...
    return list;
}
```

# Problem: Box of T

- Create a collection, that can store anything and has the following **methods**:
  - **Add()** should add on top of its contents
  - **Remove()** should remove the topmost element and **return it**
- It should have two public methods:
  - **void Add(T element)**
  - **T Remove()**
  - **int Count**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1474#0>

# Solution: Box of T

```
public class Box<T>
{
    // TODO: Add fields and constructor
    public int Count => this.data.Count;

    public void Add(T item) { this.data.Add(item);

    public T Remove() {
        var rem = this.data.Last();
        this.data.RemoveAt(this.data.Count - 1);
        return rem; }

}
```

# Problem: Generic Array Creator

- Create a class **ArrayCreator** with a single method:

```
static T[] Create(int length, T item)
```

- It should return an array with the given length
- Every element should be **set to the default item**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1474#1>

# Solution: Generic Array Creator

```
public static class ArrayCreator
{
    public static T[] Create<T>(int length, T item)
    {
        T[] array = new T[length];
        for(int i = 0; i < length; i++)
        { array[i] = item; }
        return array;
    }
}
```



# Generic Constraints

## Apply Restrictions

# Generic Constraints (1)

- **Constraints** are represented in generics using **where**
- Restricting generic classes to **reference types** only:

```
public void MyMethod<T>()  
    where T : class
```

- **class** is the keyword

```
public void MyMethod<T>()  
    where T : struct
```

- **struct** is the keyword

# Why Constraints?

- IL generated for `Equals<string>` would be different to that of `Equals<int>`

```
public static bool Equals<T> (T t1, T t2)
{
    return (t1 == t2);
}
```

- The case could be different if the `types` that are being compared have a `new definition of == operator`

# Generic Constraints (2)

- Specifying a **constructor** as a constraint

```
public void MyMethod<T>()  
    where T : new()
```

- Only a **default constructor** can be used in the constraints
- **Parameterized constructor** will be a **compilation error**

# Generic Constraints (3)

- Specifying a static **base class** as a constraint

```
public void MyMethod<T>()  
    where T : BaseClass
```

- The type **argument** must **be or derive from** the **specified base class**

# Generic Constraints (4)

- Specifying **a generic base class** as a constraint

```
public void AddAll<TItem>(List<TItem> items)
```

```
    where TItem : T
```

```
{
```

```
...
```

```
}
```

- The **type argument** supplied for **T** must **be** or **derive from** the **argument** supplied for **TItem**
- **T** comes from the **generic class**

# Combine Generic Constraints

- Specifying **a generic base class** as a constraint

```
public void MyMethod<T>()
    where T : BaseClass, new()
{
    ...
}
```

- Invalid combination of constraints: **class** and **struct**

# Problem: Equality Scale

- Create a class **EqualityScale<T>** that:
  - Holds two elements: **left** and **right**
  - Receives the elements through its single constructor:
    - **EqualityScale(T left, T right)**
  - Has a method: **bool AreEqual()**
  - The greater of the two elements is the heavier



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1474#2>

# Solution: Equality Scale

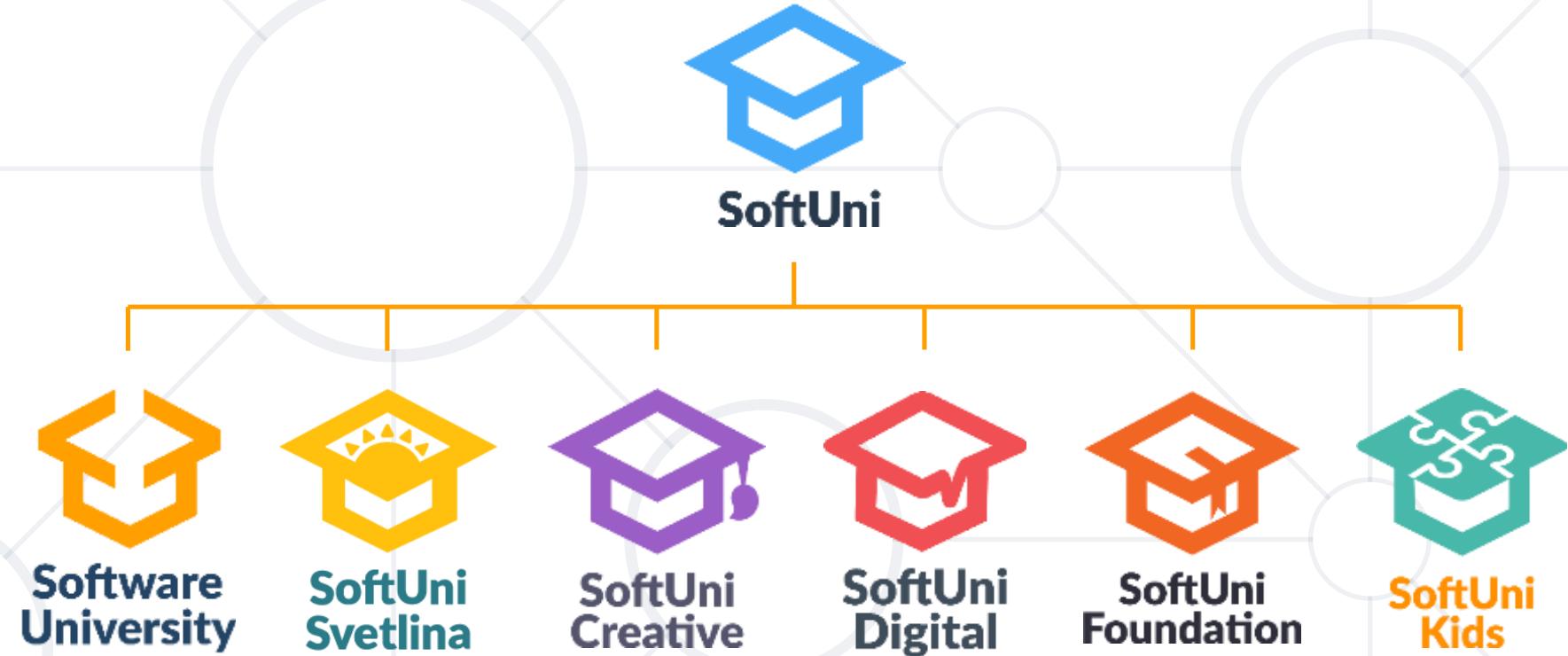
```
public class EqualityScale<T>
{
    private T left;
    private T right;
    public EqualityScale(T left, T right) {
        this.left = left;
        this.right = right;
    }
    public bool AreEqual() {
        bool result = this.left.Equals(this.right);
        return result;
    }
}
```

# Summary

- **Generics** add type safety
- Generic code is more **reusable**
- Classes, interfaces and methods can be generic
- Generic **Constraints** can validate generic types



# Questions?



# SoftUni Diamond Partners

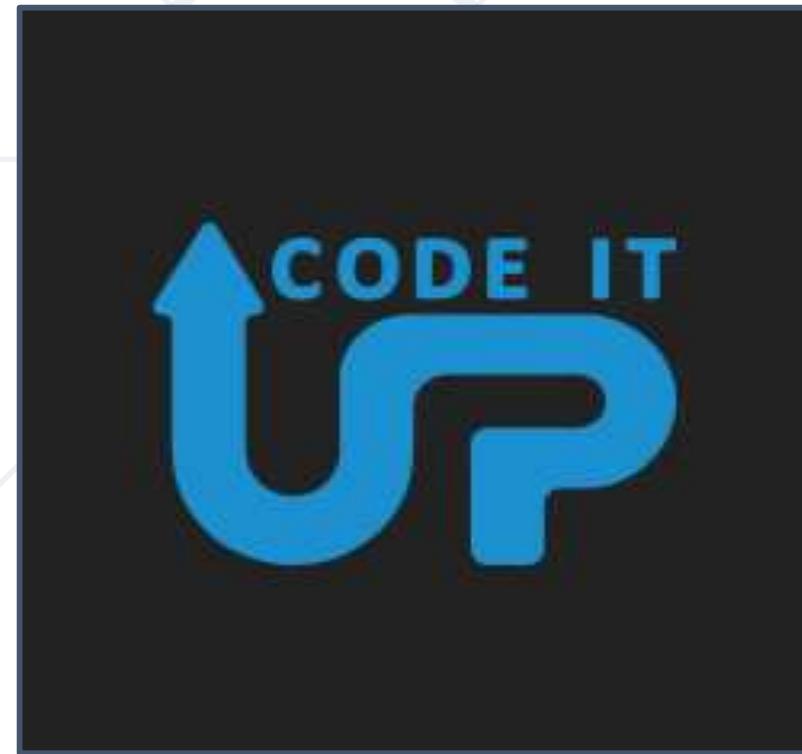


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Iterators and Comparators in C#

SoftUni Team  
Technical Trainers



SoftUni



Software University  
<https://about.softuni.bg/>

# Table of Contents

## 1. Iterators in C#

- Enumerable Collections and **foreach** Operator
- The **IEnumerable<T>** Interface
- The "yield return" Construction
- Variable Number of Parameters: the "**params**" keyword

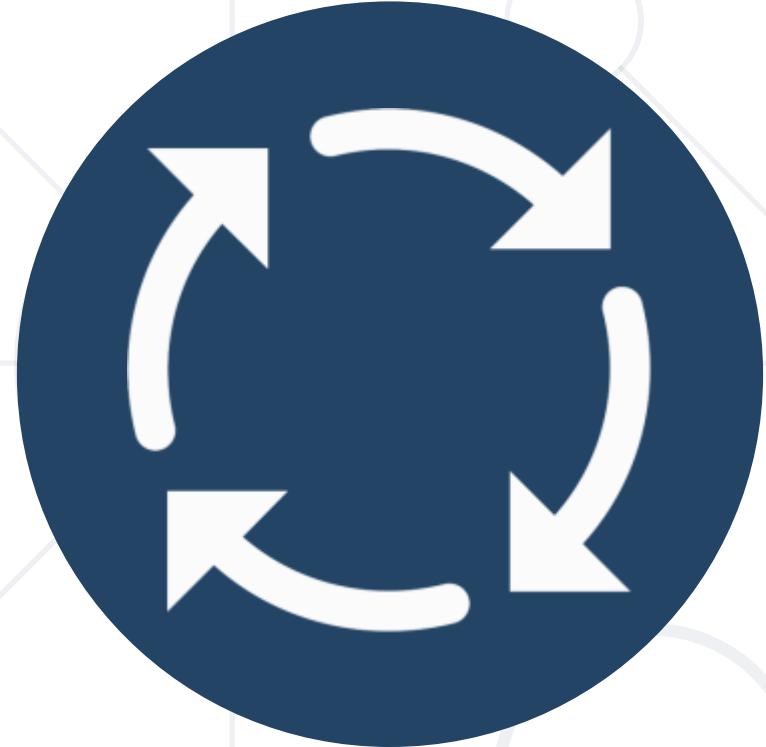
## 2. Comparators in C#

- **IComparable<T>**: Compare "this" with Another Object
- **IComparer<T>**: Compare Two Objects



sli.do

# #csharp-advanced



# Iterators in C#

**IEnumerable<T>** and **IEnumerator<T>**

# Enumerable Collections and "foreach"

- In C# **enumerable collections** and types can be traversed through the "**foreach**" loop:

```
List<int> nums = new List<int>() { 10, 20, 30 };

// Lists in .NET are enumerable → "foreach" is available
foreach (int num in nums)
    Console.WriteLine(num);
```



- Internally, **foreach** works through **iterators**:
  - The collection should implement **IEnumerable<T>**

# IEnumerable<T>

- IEnumerable<T> == the root interface for .NET types, which support **iteration** over elements
  - Defines a single method **GetEnumerator()**, which returns an **IEnumerator<T>**
  - **IEnumerator<T>** allows passing through the elements
- Types, which implement **IEnumerable<T>** can be used in a **foreach** loop traversals

```
IEnumerable<int> nums = new int[] {10, 20, 30};  
foreach (int num in nums)  
    Console.WriteLine(num);
```



# IEnumerable<T>: Definition

```
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

// Non-generic version
// (compatible with the legacy .NET 1.1)
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```



# IEnumerator<T>

- **IEnumerator<T>** implements a **sequential, forward-only iteration** over a collection
  - **Current** – returns the current element of the enumerator
  - **MoveNext()** – goes to the next element of the collection
  - **Reset()** – goes to the initial (start) position

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IEnumerator<T>
: IEnumerator
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

# The "params" Keyword in C#

- Methods can take a **variable number** of arguments:

```
PrintNames("Steve", "Teddy");
PrintNames("Peter", "Sam", "Jay", "Chriss");
```

```
void PrintNames(params string[] names)
{
    foreach(var name in names)
        Console.WriteLine(name);
}
```

- Only **one params** declaration per method; should be put **last**

# Problem: Library Iterator (1)

- Create a class **Library** to store a **collection of books** and implement the **IEnumerable<Book>** interface

**Book**

- + Title: string
- + Year: int
- + Authors: List<string>

<<**IEnumerable<Book>**>>  
**Library**

- books: List<Book>
- GetEnumerator(): IEnumerable<Book>

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1489#0>

# Problem: Library Iterator (2)

- Inside the **Library** class create nested class **LibraryIterator**, which implements **IEnumerator<Book>**

```
<<IEnumerator<Book>>>
class LibraryIterator
{
    -currentIndex: int
    -books: List<Book>
    +Current: Book

    +Reset(): void
    +MoveNext(): bool
    +Dispose(): void
}
```



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1489#1>

# Solution: Library Iterator (1)

```
public class Book {  
    public Book(string title, int year, params string[] authors) {  
        this.Title = title;  
        this.Year = year;  
        this.Authors = authors.ToList();  
    }  
    public string Title { get; private set; }  
    public int Year { get; private set; }  
    public List<string> Authors { get; private set; }  
}
```

# Solution: Library Iterator (2)

```
public class Library : IEnumerable<Book> {
    private List<Book> books;

    public Library(params Book[] books) {
        this.books = new List<Book>(books);
    }

    public IEnumerator<Book> GetEnumerator() {
        return new LibraryIterator(this.books);
    }

    IEnumerator IEnumerable.GetEnumerator()
        => this.GetEnumerator();
}
```

# Solution: Library Iterator (3)

```
private class LibraryIterator : IEnumerator<Book> {
    private readonly List<Book> books;
    private int currentIndex;
    public LibraryIterator(IEnumerable<Book> books) {
        this.books = books;
        this.Reset();
    }
    public void Dispose() {}
    public bool MoveNext() =>
        ++this.currentIndex < this.books.Count;
    public void Reset() => this.currentIndex = -1;
    public Book Current => this.books[this.currentIndex];
    object IEnumerator.Current => this.Current;
}
```

# Yield Return

- The "yield return" statement simplifies **IEnumerator<T>** implementations:

```
private readonly List<Book> books;  
  
public IEnumerator<Book> GetEnumerator()  
{  
    for (int i = 0; i < this.books.Count; i++)  
        yield return this.books[i];  
}
```

- Returns **one element** upon **each** loop cycle



# Comparators

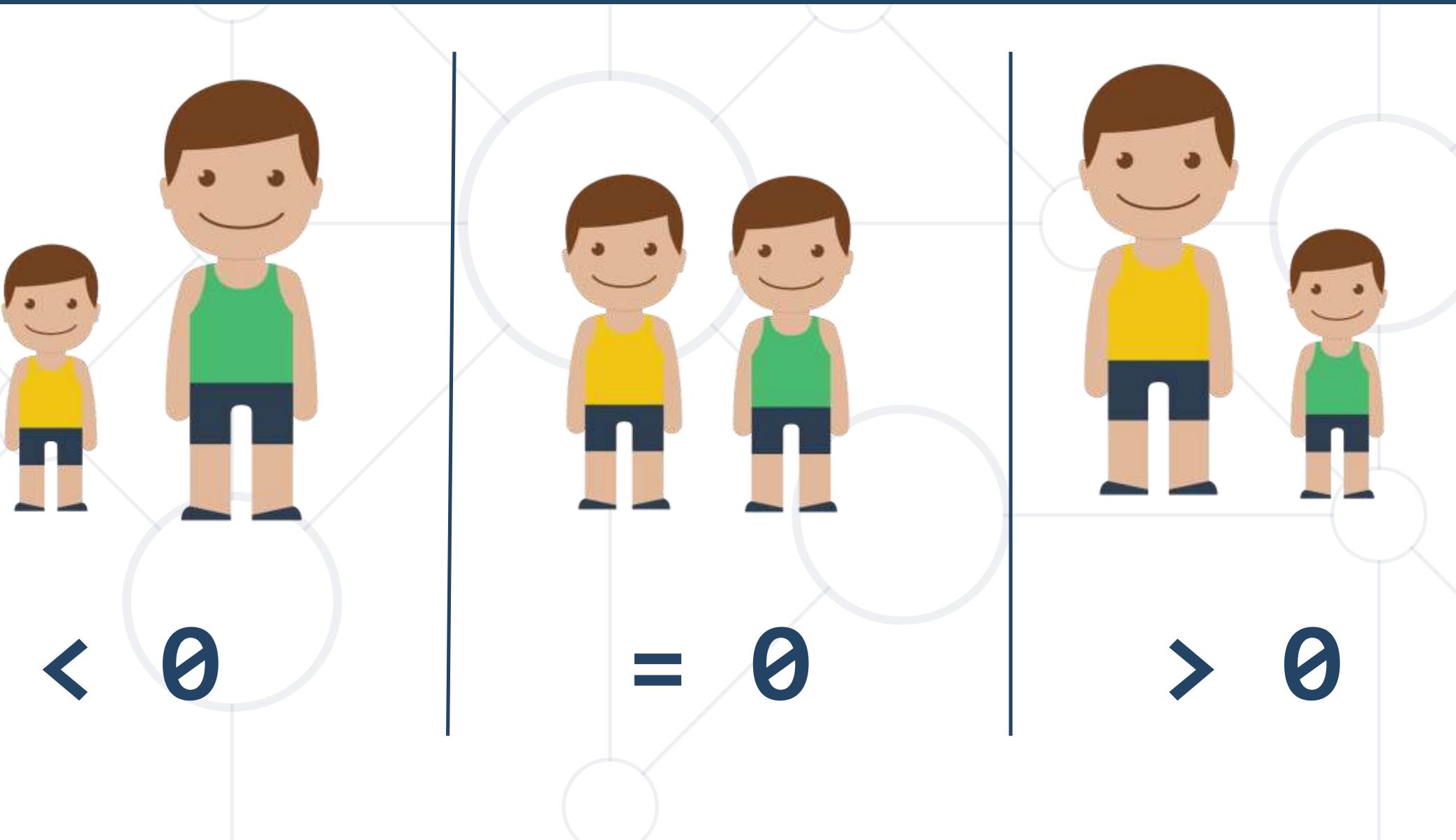
**IComparable<T>** and **IComparer<T>**

# IComparable<T>

- Reads out as "**I am Comparable**"
- Provides a method of **comparing two objects** of a particular type - **CompareTo()**
- Sets a **default sort order** for the particular object type
- **Affects** the original class



# CompareTo(T) Method Returns



# IComparable<T>: Example

```
class Point : IComparable<Point>
{
    public int X { get; set; }
    public int Y { get; set; }

    public int CompareTo(Point otherPoint)
    {
        if (this.X != otherPoint.X)
            return (this.X - otherPoint.X);
        if (this.Y != otherPoint.Y)
            return (this.Y - otherPoint.Y);
        return 0;
    }
}
```



# Problem: Comparable Book

- Implement the **IComparable<Book>** interface in the existing class **Book**
  - First sort them in **ascending chronological** order (by year)
  - If two books are published in the **same year**, sort them **alphabetically**
- Override the **ToString()** method in your **Book** class, so it returns a string in the format:
  - "**{title} - {year}**"
- Change your **Library** class so that **it stores the books** in the **correct** order

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1489#2>

# Solution: Comparable Book

```
public class Book : IComparable<Book>
{
    public int CompareTo(Book other)
    {
        int result = this.Year.CompareTo(other.Year);
        if (result == 0)
        {
            result = this.Title.CompareTo(other.Title);
        }
        return result;
    }
}
```

# IComparer<T>

- Reads out as "I'm a comparer" or "I compare"
- Provides a way to **customize** the **sort order** of a **collection**
- Defines a **method** that a type implements to **compare two objects**
- Does not **affect** original class (it's a **separate class**)

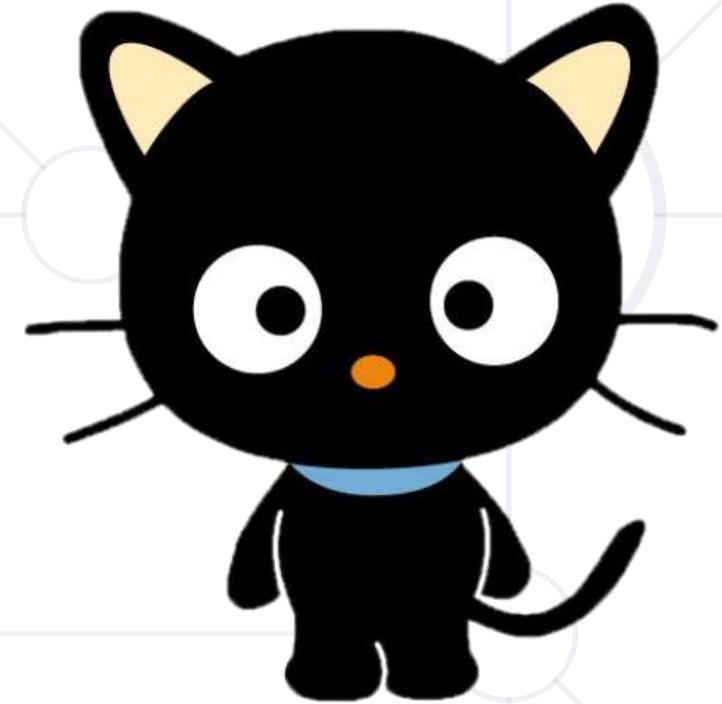


# IComparer<T> - Example

```
class Cat
{
    public string Name { get; set; }
}
```

```
class CatComparer : IComparer<Cat>
{
    public int Compare(Cat x, Cat y)
    {
        return x.Name.CompareTo(y.Name);
    }
}
```

```
IComparer<Cat> comparer = new CatComparer();
var catsByName = new SortedSet(comparer);
```



# Problem: Book Comparer

- Create a class **BookComparator**, which implements the **IComparer<Book>** interface
- **BookComparator** must **compare two books by:**
  - Book title - **alphabetical order**
  - Year of publishing a book - from **the newest to the oldest**
- Modify your **Library** class once again to implement the new sorting

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1489#3>

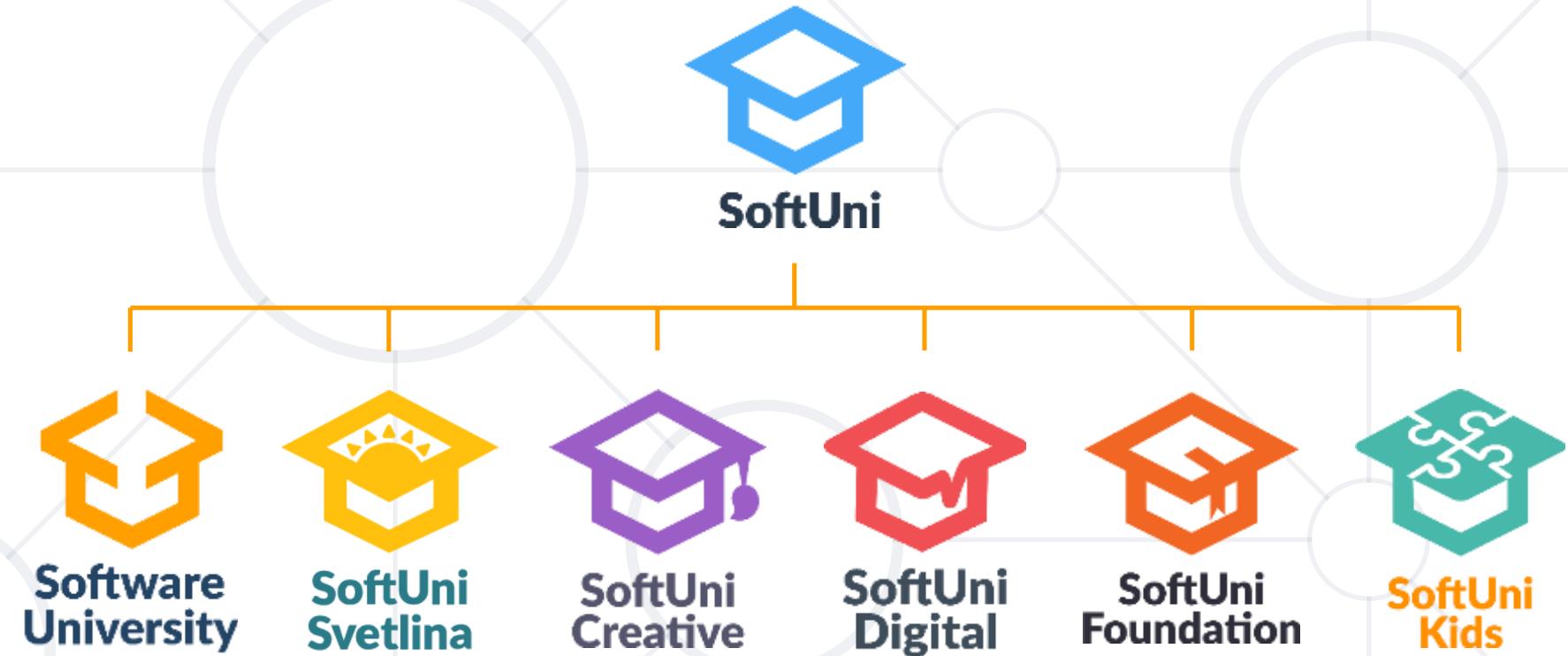
# Solution: Book Comparer

```
public class BookComparator : IComparer<Book>
{
    public int Compare(Book x, Book y)
    {
        int result = x.Title.CompareTo(y.Title);
        if (result == 0)
        {
            result = y.Year.CompareTo(x.Year);
        }
        return result;
    }
}
```

- Iterators in C#
  - **IEnumerable<T>**
  - **IEnumerator<T>**
  - **yield return**
- Params: variable number of arguments
- Comparators in C#
  - **IComparable<T>**
  - **IComparer<T>**



# Questions?



# SoftUni Diamond Partners

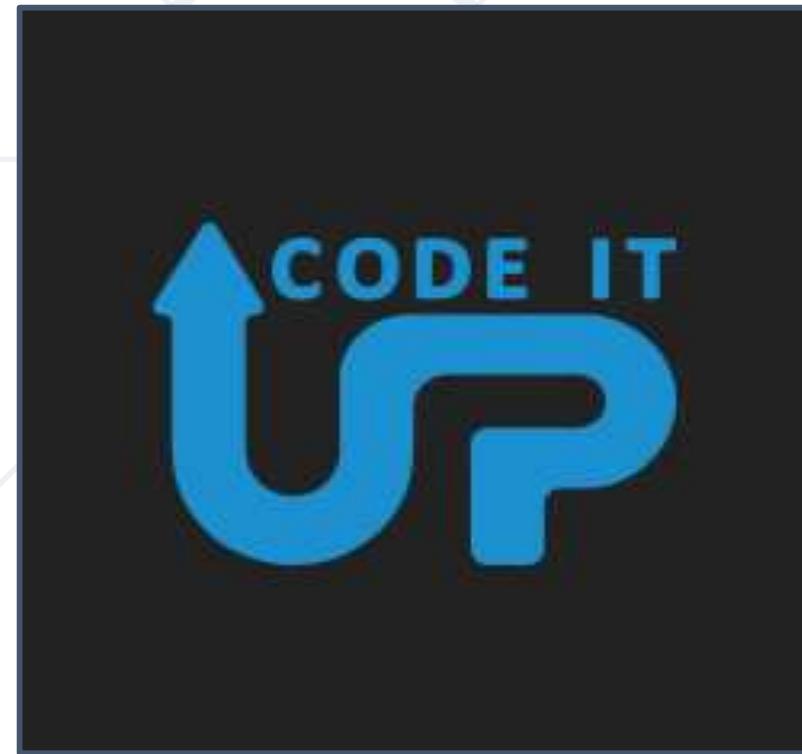


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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

