Functional Programming in C#

Lambda Expressions, Functions, Actions and Delegate

SoftUni Team

Technical Trainers

Software University

Software University

https://about.softuni.bg/

SoftUni

Table of Contents

Have a Question?

[sli.do](sli.do)

#csharp-advanced

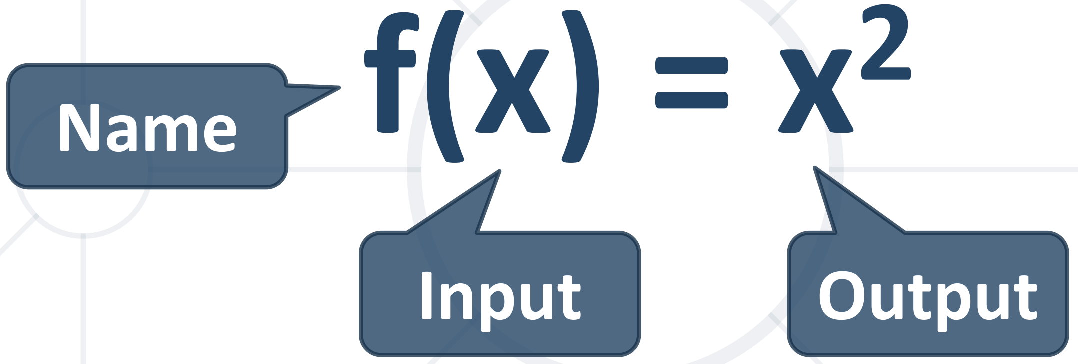
f(x)

Functional Programming

Paradigms and Concepts

What is a Function?

- **Mathematical** functions



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

- 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**

Functional Programming (FP)

- **Functional programming** (FP)
  - Programming by composing **pure functions**, avoiding **shared state**, **mutable data**, and **side-effects**
  - **Declarative** programing 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

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 programing 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>

Delegates

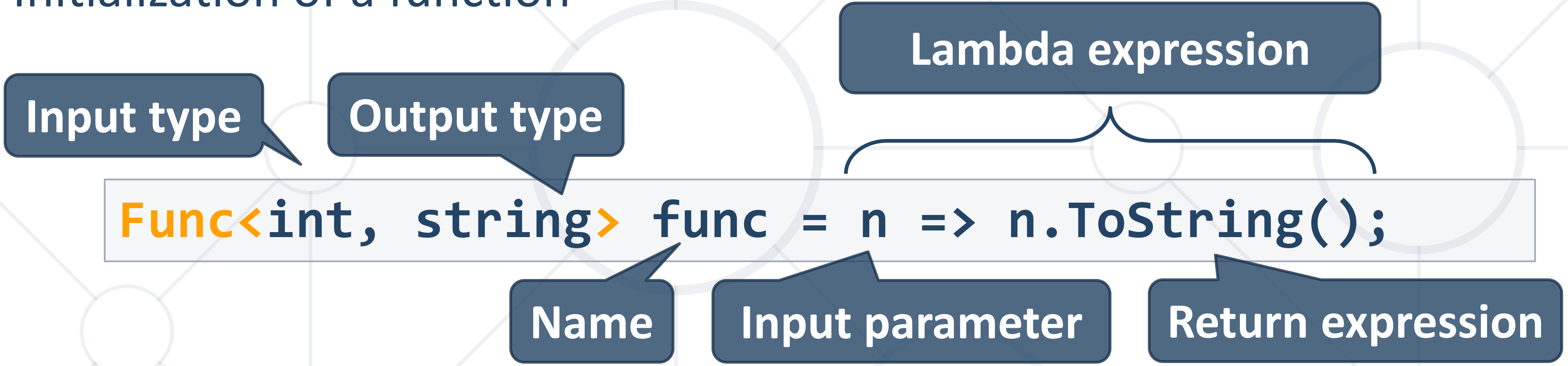
- A delegate in C# is a data type that **holds a method** with a certain parameter list and return type
  - Used to pass **methods as arguments** to other methods
- Can be used to define **callback methods**

```
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



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

- 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
```

Check your solution here: https://judge.softuni.org/Contests/Practice/Index/1472#4

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)

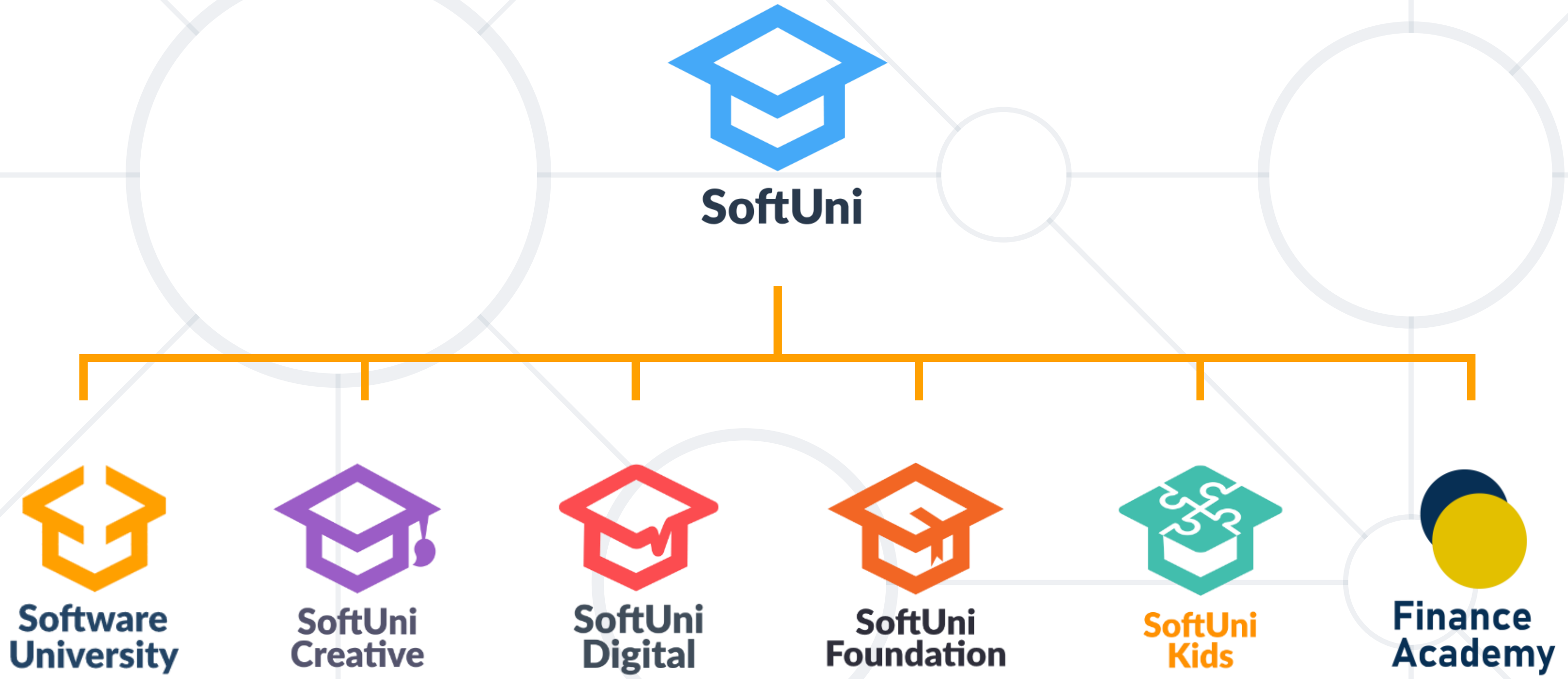
```
public static Action<Person> CreatePrinter(string format)
{
  switch (format)
    case "name":
      return person => Console.WriteLine($"{person.Name}");
    // TODO: complete the other cases
    default: throw new ArgumentException(format);
}
```

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

Summary

- **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
Software University
SoftUni Creative
SoftUni Digital
SoftUni Foundation
SoftUni Kids
Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License