

# Section 5

PL-3

## What is Function Composition?

- Combining two or more functions into a single function.
- Output of one function becomes the input of the next.

## Why Use Function Composition?

- Simplifies complex function chains.
- Makes code more readable and modular.

# Operators for Function Composition:

## 1. >> (Left-to-Right Composition):

Combines functions such that execution flows from left to right.

$$(f \gg g)(x) = g(f(x))$$

## 2. << (Right-to-Left Composition):

Combines functions such that execution flows from right to left

$$(f \ll g)(x) = f(g(x))$$

# Basic Example:

```
let add2 x = x + 2  
let multiplyBy3 x = x * 3
```

```
let addThenMultiply = add2 >> multiplyBy3  
let result = addThenMultiply 4 // Output: 18
```

```
let multiplyThenAdd = add2 << multiplyBy3  
let result = multiplyThenAdd 4 // Output: 14
```

# Real-World Example (Text Processing):

---

```
let trimText text = text.Trim()
let toUpperCase text = text.ToUpper()
let addExclamation text = text + "!"

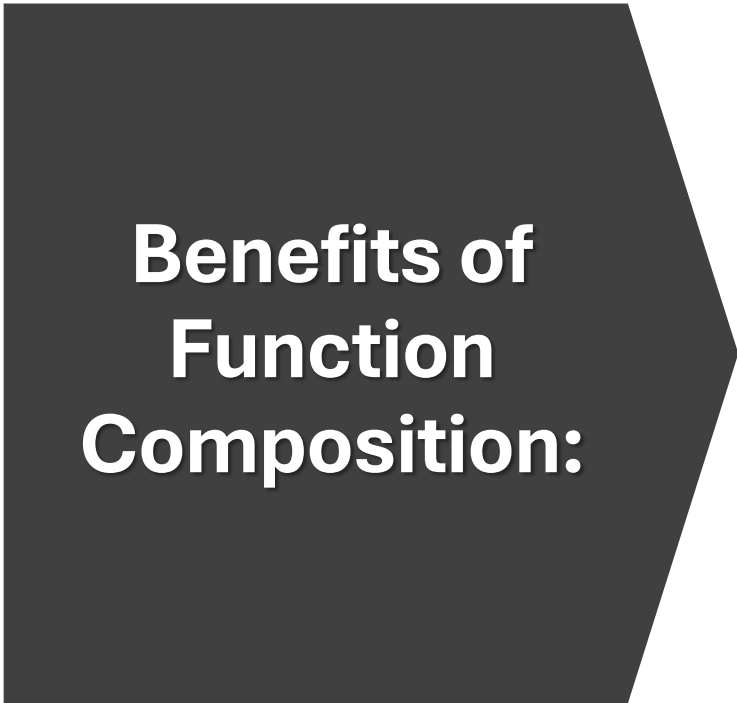
let processText = trimText >> toUpperCase >> addExclamation
let result = processText " hello f# " // Output: "HELLO F#!"
```

# Real-World Example (Text Processing):



## Execution Flow:

1. Trim whitespace.
2. Convert to uppercase.
3. Add an exclamation mark.



## **Benefits of Function Composition:**

### **1. Code Readability:**

- Simplifies chaining functions.
- Avoids deep nesting of function calls.

### **2. Reusability:**

- Individual functions can be reused in other compositions.

### **3. Functional Clarity:**

- Emphasizes the core principle of functional programming.

# Computations Expressions:

- **Computation Expressions are a flexible framework for simplifying the writing of complex code.**
- **They provide powerful tools for handling errors, loops, and asynchronous operations**
- **The code becomes shorter, clearer, and easier to maintain.**



# Computations Expressions:

- ☐ Async expressions .
- ☐ Task expressions .
- ☐ Lazy expressions.
- ☐ Option expressions.
- ☐ Result expressions.

# 1. Async Expressions in F#:

## What are Async Expressions?

- A way to write code that **does not block the entire program** while waiting for long-running tasks to complete.
- Enables the program to **continue other operations** in the meantime.

## Why use Async?

- To handle tasks that may take time (e.g., downloading data, waiting for user input).
- Keeps the application responsive, especially for user interfaces.

# How Async Works in F#:

## Key Features:

- `async {} block` → Defines an asynchronous workflow.
- `do!` Keyword → Executes an asynchronous task and waits for it to complete.
- Runs in the background without blocking the main program.

## Key Function:

- `Async.RunSynchronously` → Runs the asynchronous expression and waits for it to finish

# Example: Without Async

```
printfn "Boiling water..."  
System.Threading.Thread.Sleep 5000 // Program pauses for 5 seconds  
printfn "Water is ready!"
```

## What happens?

- The program **freezes** for 5 seconds while waiting.
- Nothing else can happen during this time.

# Example: With Async

```
let boilWater = async {  
    printfn "Boiling water..."  
    do! Async.Sleep 5000 // Simulates waiting without blocking  
    printfn "Water is ready!"  
}  
Async.RunSynchronously boilWater
```

What happens?

- The waiting ( `Async.Sleep` ) runs in the **background**.
- The program remains responsive during the wait.

# Without vs. With Async

Feature	Without Async	With Async
Responsiveness	Program blocks while waiting	Program remains responsive
Efficiency	Other tasks cannot run simultaneously	Other tasks can run simultaneously
Code simplicity	Easier to write but blocks execution	Requires more structure but non-blocking

# do! Vs. Let!

do! → For async tasks where you don't need to store the result.

let! → For async tasks where you need to store the result.

# Example with Actual Result:

When we fetch data from an API, we use **let!** to store the result:

```
open System.Net.Http

let fetchData = async {
    let client = new HttpClient()
    let! response = client.GetStringAsync("http://example.com")
    printfn "Fetched data: %s" response
}

Async.RunSynchronously fetchData
```



# Example with Actual Result:

When we fetch data from an API, we use **let!** to store the result:

```
open System.Net.Http

let fetchData = async {
    let client = new HttpClient()
    let! response = client.GetStringAsync("http://example.com")
    printfn "Fetched data: %s" response
}

Async.RunSynchronously fetchData
```

Here, **let!** stores the response from the API to use later.

## 2. Task Expressions in F#:

- This is very close to Async, but it works more with .NET programming.
- They allow you to write asynchronous code with ease and work well with tasks in the background.

```
open System.Net.Http
open System.Threading.Tasks

// Function to simulate downloading data from the internet
let downloadData (url: string) = task {
    printfn "Downloading data from %s..." url

    // Simulate a delay for downloading data
    do! Task.Delay(3000) // Delay for 3 seconds

    // Simulate data being downloaded
    printfn "Data from %s downloaded!" url
}
```

```
// Function to run multiple download tasks simultaneously
let startDownloadingData () = task {
    let url1 = "https://api.example.com/data1"
    let url2 = "https://api.example.com/data2"

    // Start downloading data using Task Expressions
    let download1 = downloadData url1
    let download2 = downloadData url2

    // Wait for all downloads to complete using Task.WhenAll
    do! Task.WhenAll([download1; download2]) // Wait for all tasks to complete
    printfn "All downloads completed!"
}
```

```
// Run the Task Expression in the background  
Task.Run(startDownloadingData) |> ignore
```

### **startDownloadingData :**

A function that launches multiple tasks (downloading data from two different sources simultaneously).

Uses **Task.WhenAll** to wait for all tasks to complete before proceeding.

### **Task.Run(startDownloadingData) :**

Runs startDownloadingData in the background using Task.Run.

The **|> ignore** ensures *we're not concerned with the return value* of the task; *we only want it to execute*.

### 3. Lazy Expressions in F#:

- **Lazy Expressions** in F# *delay* the computation of a value until it is accessed for the first time.
- Useful for improving performance and avoiding unnecessary computations.
- Particularly helpful for expensive or conditional calculations.

```
let lazyValue = lazy (printfn "Computing..."; 42)
```

```
// Accessing the lazy value
```

```
let value = lazyValue.Force() // Triggers computation
```

```
printfn "Value: %d" value
```

## 4. Option Expressions:

In F#, **Option Expressions** are used to handle values that might or might not exist. It's similar to the concept of *nullable types* in other languages, but more structured and type-safe. Options are especially helpful when you want to avoid null reference errors.



## Key Concepts of Option Expressions:

- The `Option` type has two possible values:
  - `Some(value)` : Indicates that a value exists.
  - `None` : Indicates the absence of a value.
- Commonly used in functions where a result might not always be returned.

# Example 1: Handling Optional Values

```
let divide x y =  
  if y = 0 then None  
  else Some(x / y)
```

```
let result = divide 10 2  
match result with  
| Some(value) -> printfn "Result: %d" value  
| None -> printfn "Division by zero!"
```

- If the denominator (y) is 0, the function returns None.
- Otherwise, it wraps the result in Some(value).

# Example 1: Handling Optional Values

```
let divide x y =  
  if y = 0 then None  
  else Some(x / y)
```

```
let result = divide 10 2  
match result with  
| Some(value) -> printfn "Result: %d" value  
| None -> printfn "Division by zero!"
```

- If the denominator (y) is 0, the function returns None.
- Otherwise, it wraps the result in Some(value).

## Example 2: Using Option.DefaultValue

```
let result = divide 10 0
let value = Option.defaultValue -1 result
printfn "Value: %d" value
```

- If the value is None, Option.defaultValue provides a default value (-1 in this case).

```
let result1 = Some(10)
```

```
let result2 = None
```

```
let value1 = Option.defaultValue -1 result1
```

```
let value2 = Option.defaultValue -1 result2
```

```
printfn "Value1: %d" value1
```

```
printfn "Value2: %d" value2
```

Value1: 10

Value2: -1

- **Option.defaultValue only works when the value is None.**
- **If the value is Some(value), it returns the value inside Some.**

## 5. Result Expressions:

In F#, Result Expressions are used to represent computations that may produce either:

- A successful result (`Ok(value)`), or
- An error (`Error(value)`).

```
let parseNumber input =  
    match System.Int32.TryParse(input) with  
    | (true, value) -> Ok(value)  
    | _ -> Error("Invalid number")
```

`System.Int32.TryParse` → tries to convert a string to a number.

If conversion succeeds → result is `Ok(value)`.

If conversion fails → result is `Error("Invalid number")`.

```
let result = parseNumber "123"  
match result with  
| Ok(value) -> printfn "Parsed number: %d" value  
| Error(err) -> printfn "Error: %s" err
```

- ❑ If `parseNumber` returns `Ok(value)`, we print the number.

```
Parsed number: 123
```

- ❑ If it returns `Error(err)`, we print the error message.

```
Error: Invalid number
```



# Example (Chaining Results) :

```
let validatePositive x =  
    if x > 0 then Ok(x)  
    else Error("Number must be positive")  
  
let computeSquareRoot x =  
    if x >= 0.0 then Ok(System.Math.Sqrt(x))  
    else Error("Cannot compute square root of a negative number")  
  
let result =  
    match validatePositive -5 with  
    | Ok(value) -> computeSquareRoot (float value)  
    | Error(err) -> Error(err)  
  
match result with  
| Ok(value) -> printfn "Square root: %f" value  
| Error(err) -> printfn "Error: %s" err
```

# Example (Chaining Results) :

- If the number (x) is -5 the result will be

Error: Number must be positive

**And,**

- If the number (x) is 4 the result will be

Square root: 2.000000

# Difference between Result and Option

## Option:


- Expresses only the presence or absence of a value (Some/None).

## Result:

- Expresses success or failure with details  
(Success: Ok(value), Failure: Error(reason)).

The background of the slide is a dark, textured surface covered with numerous question marks. Some question marks are in a light beige or gold color, while others are in a dark grey or black color. They are scattered across the entire frame, creating a dense, patterned effect.

**Any Questions?!**

- 
- ❑ Please, each team must fill out the form once.
  - ❑ Deadline : Next Friday → 29/11/2024.
  - ❑ The team must consist of 6 to 7 members.

<https://forms.gle/5uZA2Y9Re7Tr6Poj6>





**Thank YOU !**