Section-3

Programming Language 3

T.A :Shereen El-gazzar

Data Types Overview:

- **Basic Types :** int, float, String, bool.
- **Compound Types:** List, Array, Tuple, Set, … .
- **Printing:** Using printfn "%A“ *to print any type*

❑ *Note:* printfn *or* printf

❑ printfn → n in the end refer to new line.

Arrays :

- **Definition:** Fixed-size collection of elements of the same type.
- **Example:** `let numbers = [| 1; 2; 3; 4 |]`

Lists :

- **Definition:** Flexible-size collection, similar to a linked list.
- **Example:** `let numbers = [1; 2; 3; 4]`

Sets :

Definition: Unordered collection of unique elements.

```
let mySet = set [1; 2; 3; 4]
```

```
let mySetDirect = set {1; 2; 3; 4}
```

Accessing Values:

```
let mySet = set [1; 2; 3; 4]
let updatedSet = mySet.Add(5)
```

```
let mySet = set {1; 2; 3; 4}
let smallerSet = mySet.Remove(3)
```

Sets :

```
let mySet = set [1; 2; 3; 4]
let hasTwo = mySet.Contains(2)
let hasSeven = mySet.Contains(7)

printfn "Does the set contain the number 2? %b" hasTwo
printfn "Does the set contain the number 7? %b" hasSeven
```

Output:

```
Does the set contain the number 2? true
Does the set contain the number 7? false
```

Records :

- **Description**: A Record is a data structure that holds a set of named fields, with each field potentially being of a different type.
- **Advantages**: It's ideal for organizing complex data since each field is clearly named, making it easy to understand and use.

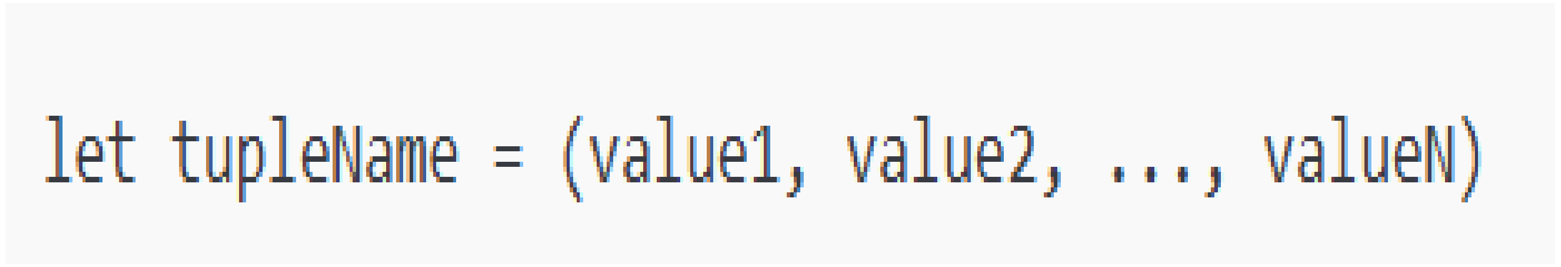
```
type Person = { Name: string; Age: int; Job: string }

let person = { Name = "Ali"; Age = 25; Job = "Engineer" }
```

```
printfn "%A" person
```

Tuples :

- A **tuple** is an ordered collection of values, and the number of elements in a tuple is fixed *when it's created*.

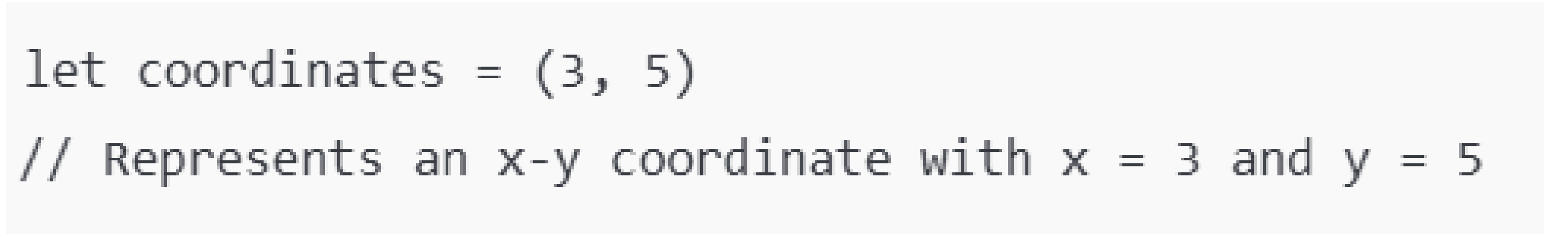
  Here's the general syntax:



```
let tupleName = (value1, value2, ..., valueN)
```

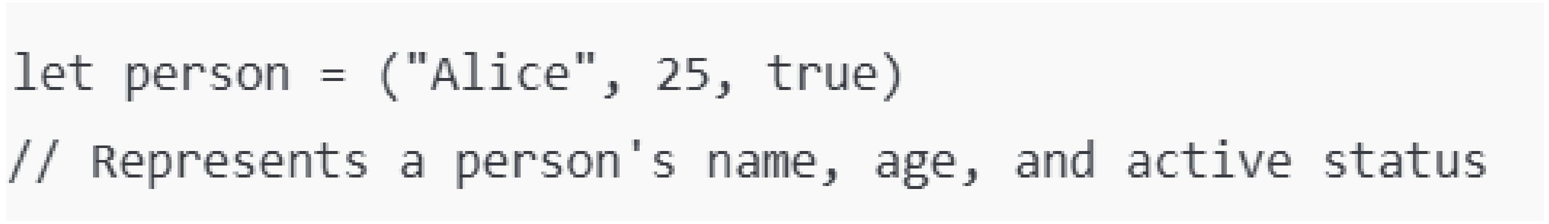
Types of Tuples:

1. **Simple Tuple**: Contains two values of same or different types.



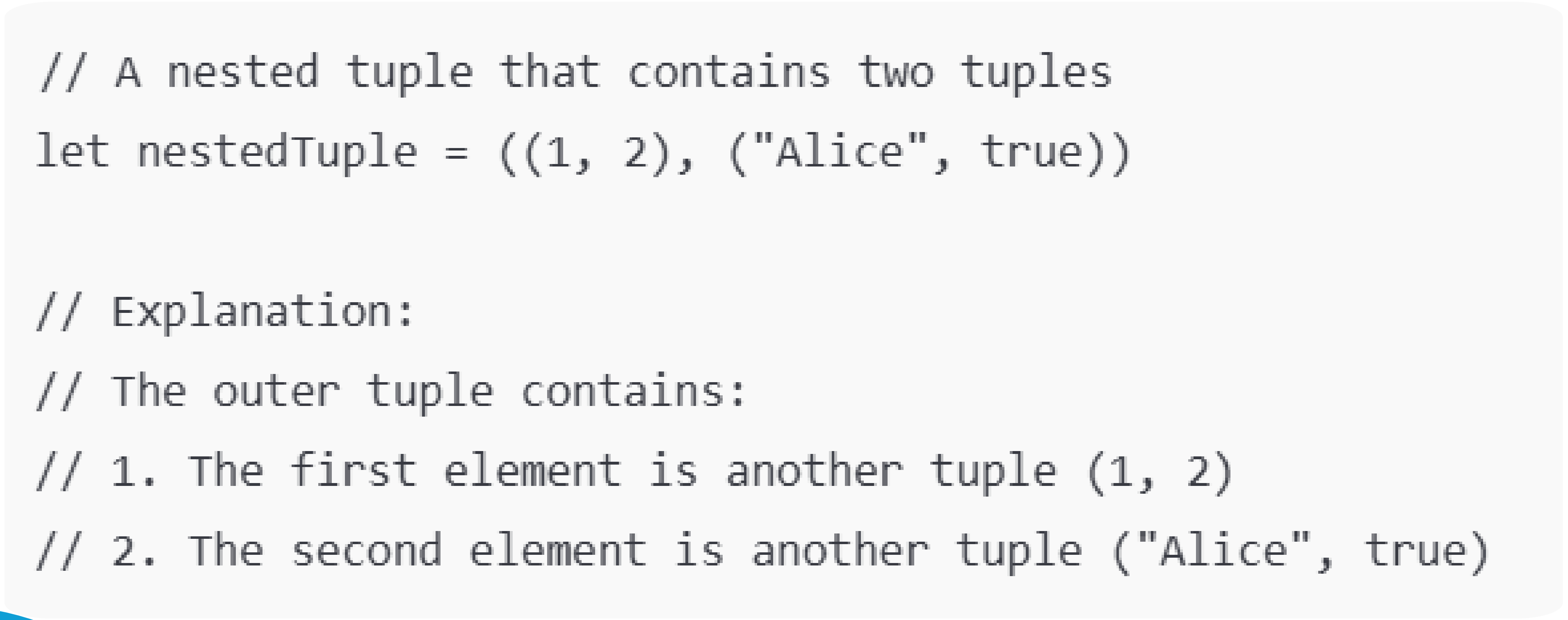
```
let coordinates = (3, 5)
// Represents an x-y coordinate with x = 3 and y = 5
```

Mixed Types: Tuple with different types of values.



```
let person = ("Alice", 25, true)
// Represents a person's name, age, and active status
```

2. **Nested Tuples:** You can even nest tuples within each other.



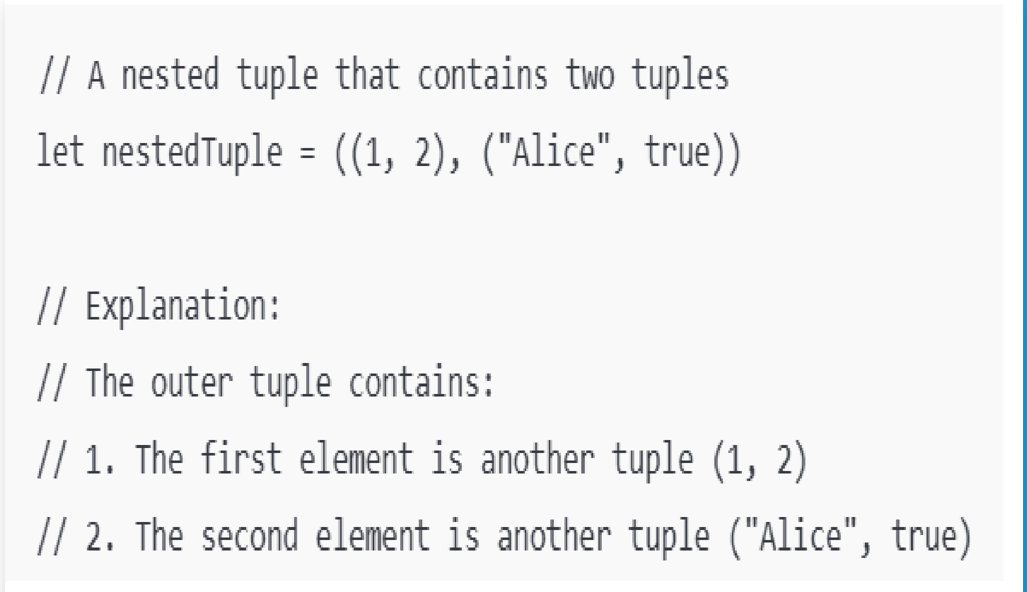
```
// A nested tuple that contains two tuples
let nestedTuple = ((1, 2), ("Alice", true))


// Explanation:
// The outer tuple contains:
// 1. The first element is another tuple (1, 2)
// 2. The second element is another tuple ("Alice", true)
```

Handling Paired Data: When values have a natural order or pairing, like coordinates (x, y) or date parts (day, month, year), tuples are a good fit.

```
let coordinates = (3, 5)
// Represents an x-y coordinate with x = 3 and y = 5
```

```
let date = (22, 10, 2024)  // (day, month, year)
let (day, month, year) = date
printfn "Date: %02d-%02d-%04d" day month year
```

Example of Nested Tuples in F#:

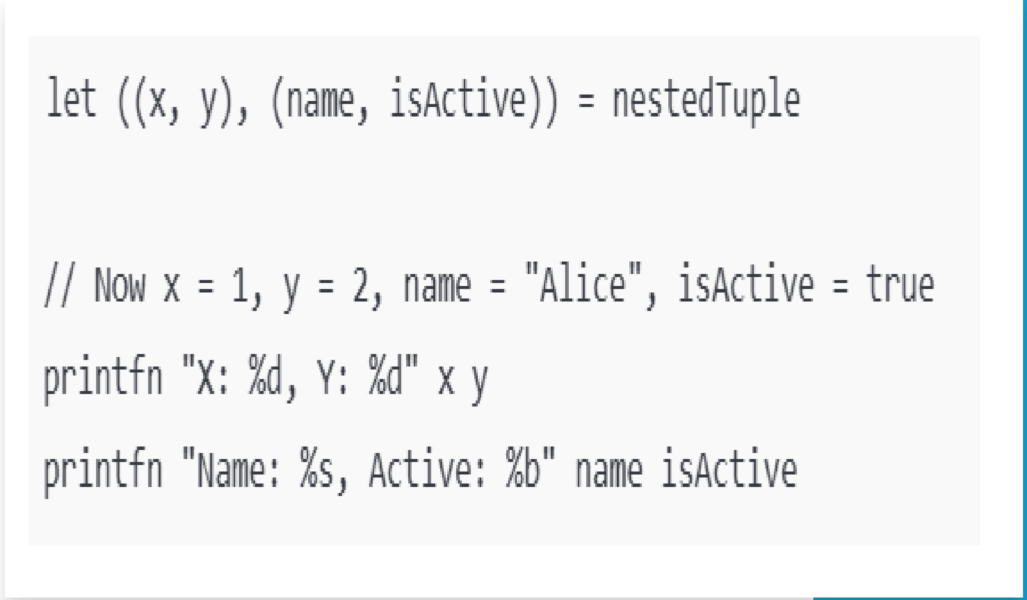


```
// A nested tuple that contains two tuples
let nestedTuple = ((1, 2), ("Alice", true))

// Explanation:
// The outer tuple contains:
// 1. The first element is another tuple (1, 2)
// 2. The second element is another tuple ("Alice", true)
```

To access the values within a nested tuple, you can use **pattern matching**.

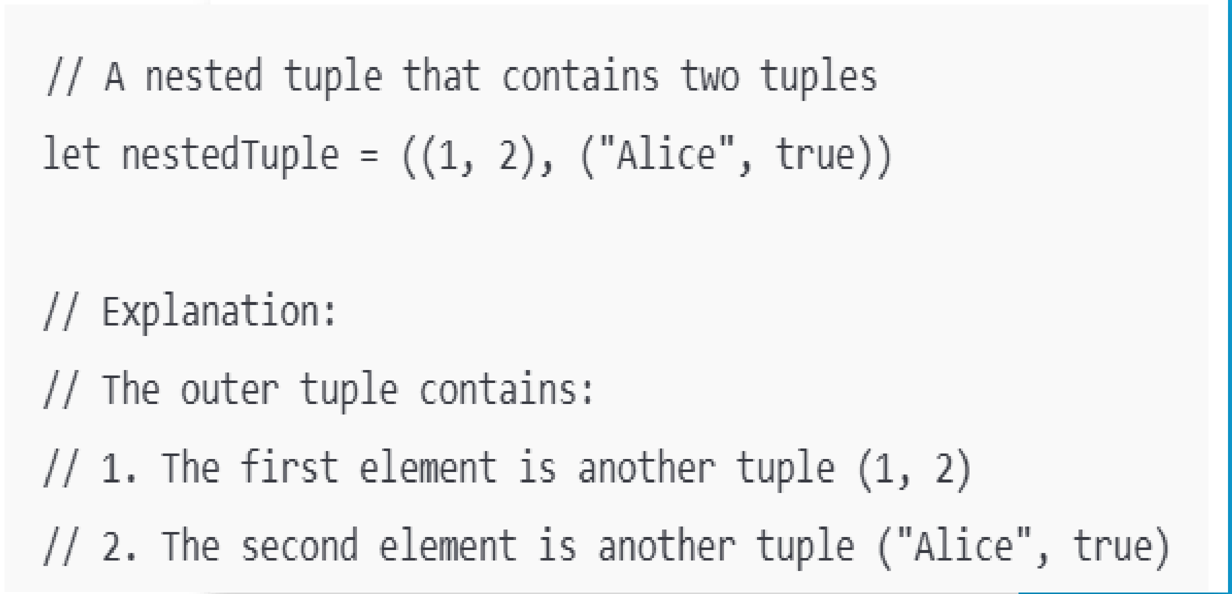
Accessing Values in a Nested Tuple

→



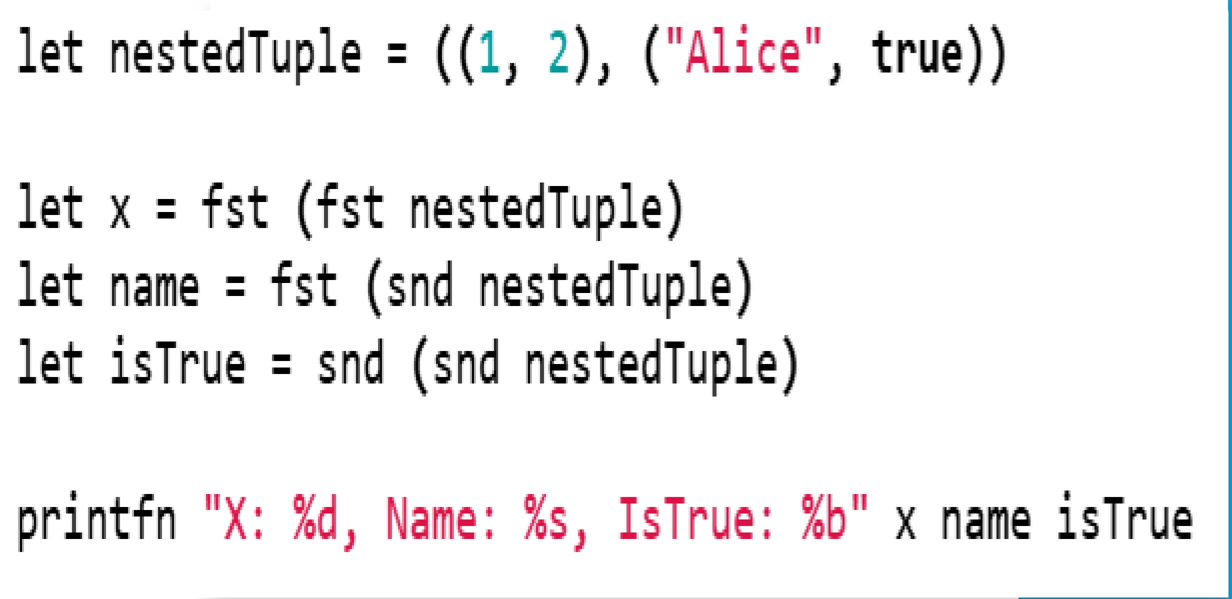
```
let ((x, y), (name, isActive)) = nestedTuple

// Now x = 1, y = 2, name = "Alice", isActive = true
printfn "X: %d, Y: %d" x y
printfn "Name: %s, Active: %b" name isActive
```

Accessing Nested Tuples in F# using manual access:



```
// A nested tuple that contains two tuples
let nestedTuple = ((1, 2), ("Alice", true))

// Explanation:
// The outer tuple contains:
// 1. The first element is another tuple (1, 2)
// 2. The second element is another tuple ("Alice", true)
```

Accessing Values in a Nested Tuple →



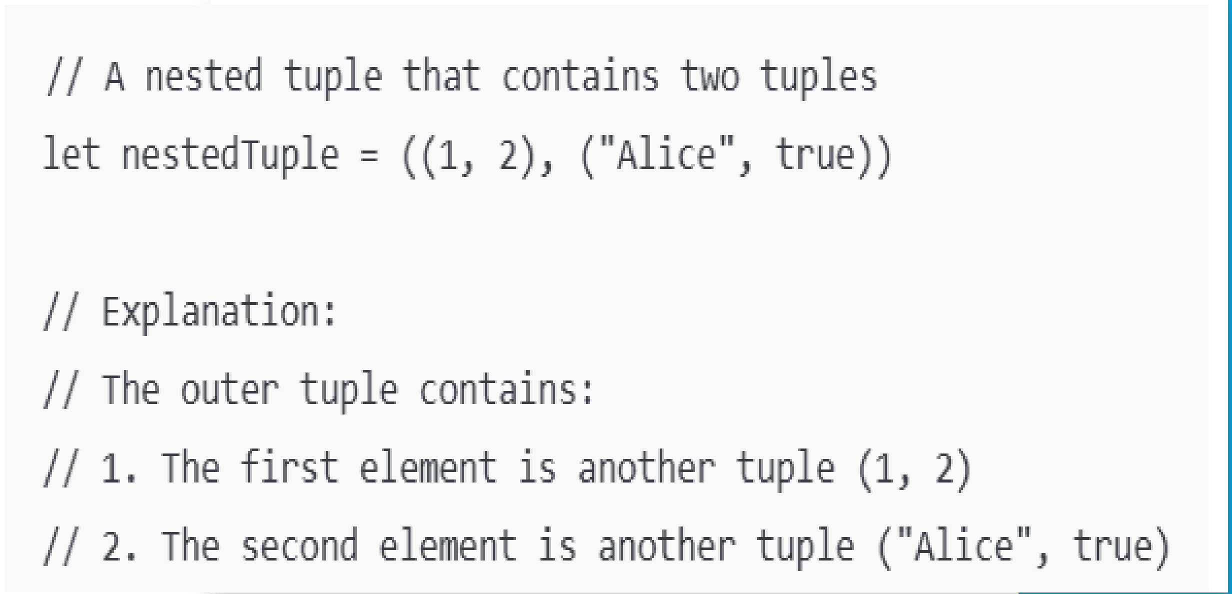
```
let nestedTuple = ((1, 2), ("Alice", true))

let x = fst (fst nestedTuple)
let name = fst (snd nestedTuple)
let isTrue = snd (snd nestedTuple)

printfn "X: %d, Name: %s, IsTrue: %b" x name isTrue
```

Accessing Nested Tuples in F# using manual access:

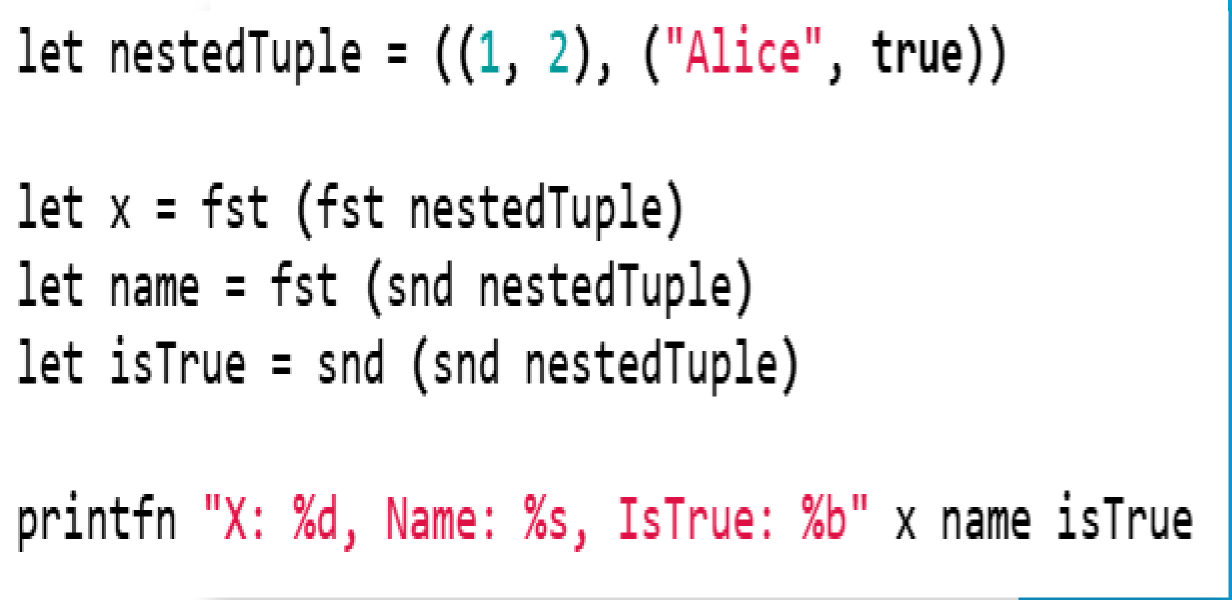
Note : fst and snd only works with tuples containing 2 elements

pattern matching is the most common and clear way to access values within nested tuples.

Accessing Values in a Nested Tuple →

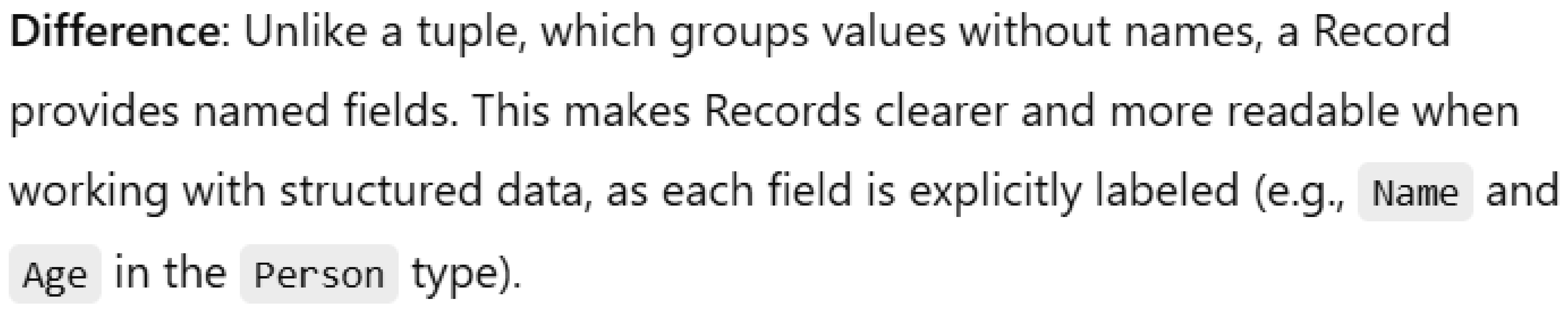


```
// A nested tuple that contains two tuples
let nestedTuple = ((1, 2), ("Alice", true))


// Explanation:
// The outer tuple contains:
// 1. The first element is another tuple (1, 2)
// 2. The second element is another tuple ("Alice", true)
```



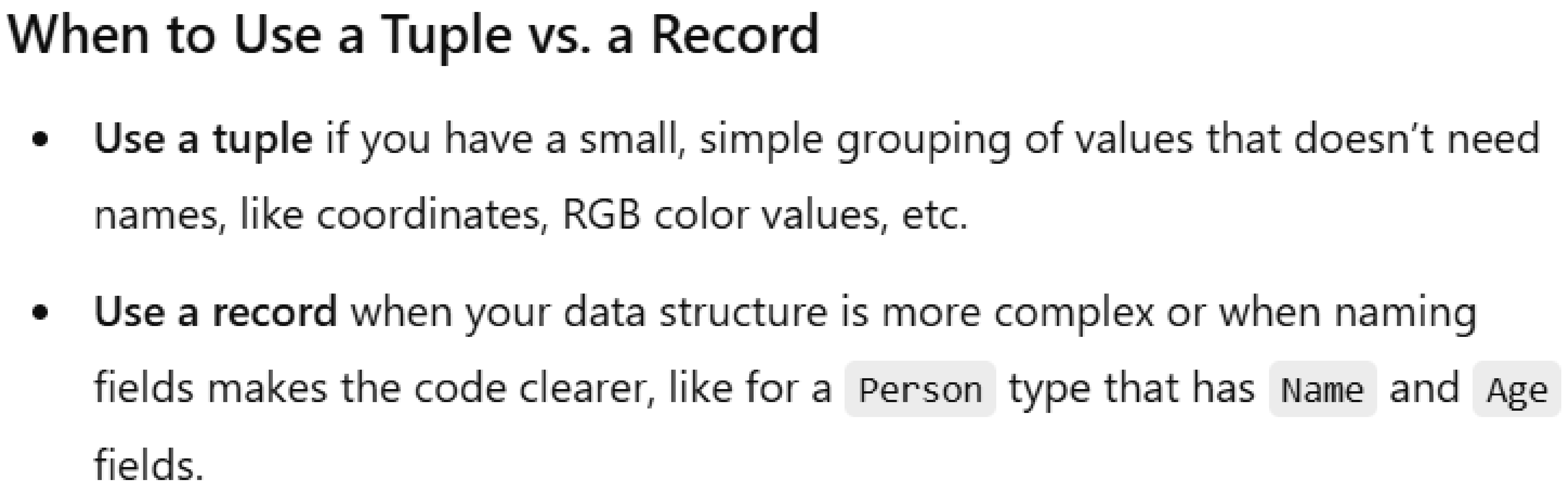
```
let nestedTuple = ((1, 2), ("Alice", true))

let x = fst (fst nestedTuple)
let name = fst (snd nestedTuple)
let isTrue = snd (snd nestedTuple)

printfn "X: %d, Name: %s, IsTrue: %b" x name isTrue
```

Difference between Tuples and Records:



Difference: Unlike a tuple, which groups values without names, a Record provides named fields. This makes Records clearer and more readable when working with structured data, as each field is explicitly labeled (e.g., `Name` and `Age` in the `Person` type).

Difference between Tuples and Records:



When to Use a Tuple vs. a Record

- **Use a tuple** if you have a small, simple grouping of values that doesn't need names, like coordinates, RGB color values, etc.
- **Use a record** when your data structure is more complex or when naming fields makes the code clearer, like for a `Person` type that has `Name` and `Age` fields.

Maps :

Maps (or dictionaries) are collections of key-value pairs, where each key is unique.

```
let myMap = map [("a", 1); ("b", 2); ("c", 3)]
```

```
let myMap = Map.empty.Add("a", 1).Add("b", 2).Add("c", 3)
```

Accessing Values Using a Key

You can retrieve a value in the map using the key like this:

```
let value = myMap.["a"]  // Returns 1
```

Checking if a Key Exists Using `ContainsKey`

To check if a specific key exists in the map:

```
let hasKey = myMap.ContainsKey("a")  // Returns true
```

Adding or Removing Elements

- **Adding an element** (creates a new map):

```
let updatedMap = myMap.Add("d", 4)
```

- **Removing an element** (creates a new map):

```
let smallerMap = myMap.Remove("b")
```

Iterating Over Map Elements

You can use a `for` loop to iterate over all keys and values in a map:

```
for (key, value) in myMap do
    printfn "Key: %s, Value: %d" key value
```

A **Map** in F# is a data structure for storing key-value pairs.

Maps are immutable, so any modification results in a new map.

They are stored in an ordered fashion, based on the keys.

Maps allow for efficient lookup by key, making them useful for fast data retrieval

Discriminated Unions: (الاتحادات المميزة)

```
type ApiResponse =
    | Success of data: string
    | Error of message: string
    | Timeout

let handleApiResponse (response: ApiResponse) =
    match response with
    | Success data -> printfn "Request succeeded with data: %s" data
    | Error message -> printfn "Request failed with error: %s" message
    | Timeout -> printfn "Request timed out"
let response1 = Success "User data loaded"
handleApiResponse response1  // Output: Request succeeded with data: User data loaded
```

cont.

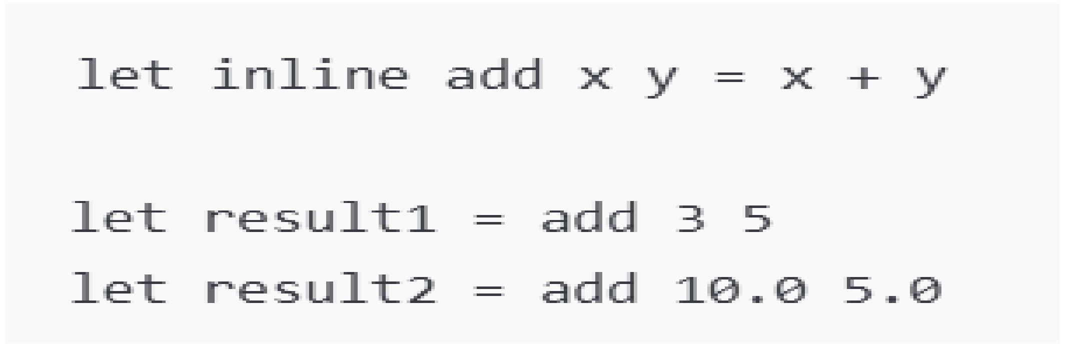
```
type ApiResponse =
    | Success of data: string
    | Error of message: string
    | Timeout

let handleApiResponse response =
    match response with
    | Success data -> printfn "Request succeeded with data: %s" data
    | Error message -> printfn "Request failed with error: %s" message
    | Timeout -> printfn "Request timed out"

let response1 = Success "User data loaded"
let response2 = Error "Network error occurred"
let response3 = Timeout
```

```
handleApiResponse response1
handleApiResponse response2
handleApiResponse response3
```

Functions

Inline functions :

allowing the compiler to replace the function call with the function body at compile time. This can lead to performance improvements, especially in cases where the function is small and called multiple times, as it avoids the overhead of a function call.



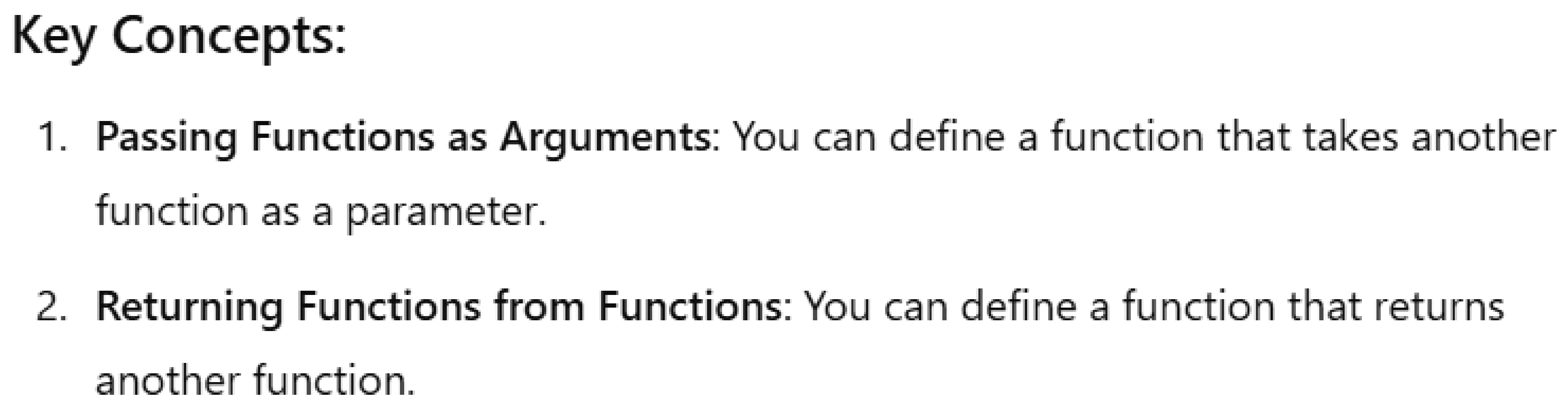
```
let inline add x y = x + y

let result1 = add 3 5
let result2 = add 10.0 5.0
```

Activities:

- Create inline functions for basic arithmetic operations.

Higher-Order Functions :

- ➢ are functions that can take other functions as parameters or return functions as their results. This is a powerful feature of functional programming that allows for more abstract and reusable code.



Key Concepts:

1. **Passing Functions as Arguments**: You can define a function that takes another function as a parameter.
2. **Returning Functions from Functions**: You can define a function that returns another function.

1. Passing Functions as Arguments:

Let's create a higher-order function called `applyTwice`, which takes a function and a value, applies the function to the value twice, and returns the result.

```
let applyTwice f x = f (f x)

// A simple function to increment a number
let increment x = x + 1

// Using the higher-order function
let result = applyTwice increment 5
printfn "Result of applying increment twice: %d" result  // Output: 7
```

2. Returning functions from functions:

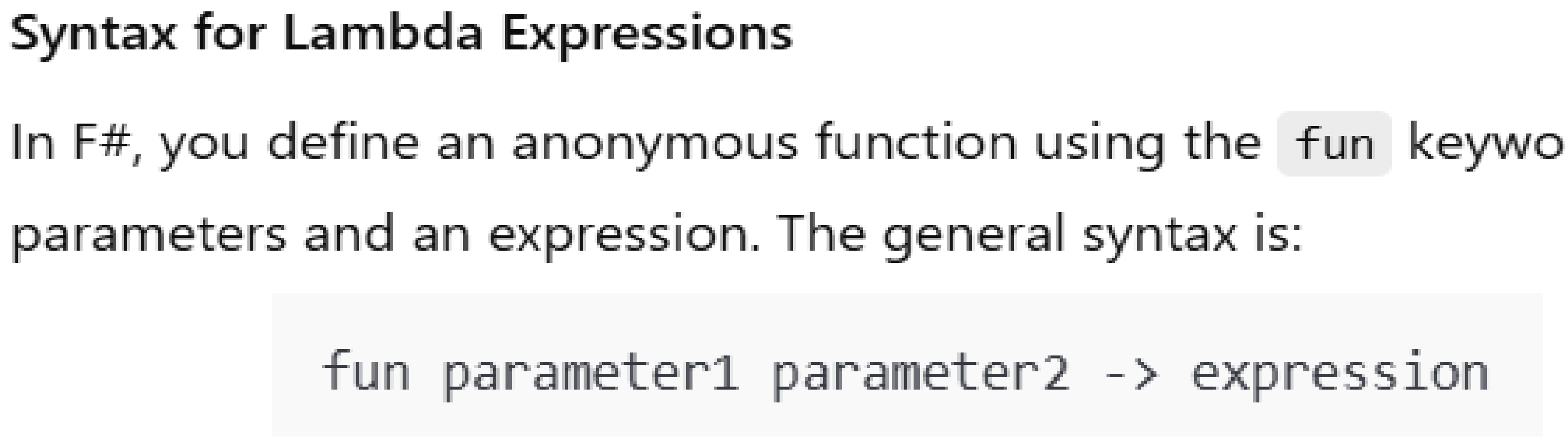
Now, let's create a function called `makeMultiplier` that returns another function.

This returned function will multiply its input by a specified factor.

```
let makeMultiplier factor =
    fun x -> x * factor

// Create a function that doubles its input
let double = makeMultiplier 2

// Using the returned function
let result = double 10
printfn "Result of doubling 10: %d" result  // Output: 20
```

Anonymous Functions and Lambda Expressions in F#:

Anonymous functions, also known as **lambda expressions**, are functions that are defined without a name. They are often used for short-term tasks and can be defined inline where they are needed. This makes them convenient for situations where you need a function only once.



Syntax for Lambda Expressions

In F#, you define an anonymous function using the `fun` keyword followed by parameters and an expression. The general syntax is:

```
fun parameter1 parameter2 -> expression
```

Example:

```
// Defining an anonymous function to add two numbers
let add = fun x y -> x + y

// Using the anonymous function
let result = add 5 7
printfn "Result of adding 5 and 7: %d" result  // Output: 12
```

Use Cases in Collections

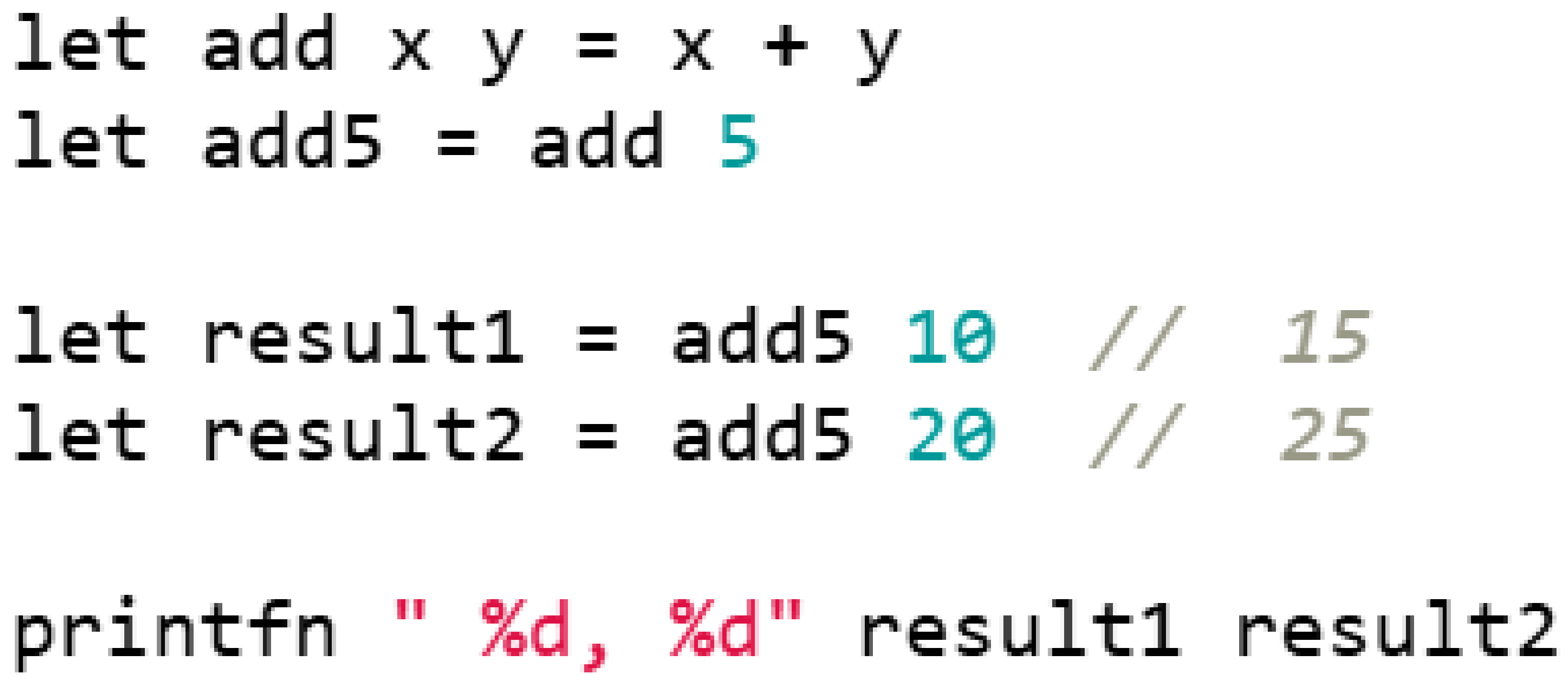
Anonymous functions are particularly useful when working with collections. For example, you can use them with functions like `List.map`, `List.filter`, or `List.fold` to perform operations on lists.

Example: Using an Anonymous Function with `List.map`

Let's say you want to square each number in a list:

```
let numbers = [1; 2; 3; 4; 5]

// Using an anonymous function with List.map
let squares = List.map (fun x -> x * x) numbers

printfn "Squares: %A" squares  // Output: Squares: [1; 4; 9; 16; 25]
```

Partial Application and Currying:



```
let add x y = x + y
let add5 = add 5

let result1 = add5 10  //  15
let result2 = add5 20  //  25

printfn " %d, %d" result1 result2
```

Resources can help YOU :

- https://www.youtube.com/watch?v=yGzu0iDuMNQ&list=PLdo4fOcmZ0oUFghYOp89baYFBTGxUkC7Z&pp=iAQB

- https://www.youtube.com/watch?v=Teak30_pXHk&list=PLEoMzSkcN8oNiJ67Hd7oRGgD1d4YBxYGC&pp=iAQB

- https://learn.microsoft.com/en-us/dotnet/fsharp/

Any Question ?!

Thank You!