First Class or Higer Order Functions

🔹 1. First-Class Functions in Scala

Scala treats **functions as first-class citizens**.
That means:
✅ You can

- assign a function to a variable,
- pass it as an argument, and
- return it from another function.

👉 In short: **Functions are values**.

✅ Example: Assigning a Function to a Variable

```
// Define a normal function
def square(x: Int): Int = x * x
// Assign it to a variable
val f = square _   // underscore converts to function value
println(f(5))   // Output: 25
```

🧠 Explanation:

- The underscore _ turns the method square into a **function value**.
- f is now a reference to that function and can be passed around like data.

✅ Example: Anonymous (Lambda) Function

```
val cube = (x: Int) => x * x * x
println(cube(3))  // Output: 27
```

- (x: Int) => x * x * x is an **anonymous function** (no name).
- Assigned to a variable cube.

🔹 2. Higher-Order Functions (HOF)

A **Higher-Order Function** is a function that either:

- takes another function as a **parameter**, or
- returns another function as a **result**, or
- does both.

This is what makes Scala truly *functional*.

✅ Example 1: Function as Parameter

```
def applyFunc(f: Int => Int, value: Int): Int = f(value)
val double = (x: Int) => x * 2
val result = applyFunc(double, 10)
println(result)   // Output: 20
```

🧠 Explanation:

- applyFunc accepts another function f as an argument.
- It calls f(value) inside.
- double is passed in — that's **higher-order** behavior.

✅ Example 2: Function Returning Another Function

```
def multiplier(factor: Int): Int => Int = {
  (x: Int) => x * factor
}
val times3 = multiplier(3)
println(times3(5))   // Output: 15
```

🧠 Explanation:

- multiplier returns a new function (x: Int) => x * factor.
- You can store it and call it later.

✅ Example 3: Real-world Use – ETL Data Transformation

Imagine you have a list of sales amounts and you want to apply different transformations dynamically:

```
val sales = List(1000, 2000, 3000)
def transformData(list: List[Int], func: Int => Int): List[Int] = list.map(func)
// Define transformations
val addTax = (x: Int) => (x * 1.18).toInt
val giveDiscount = (x: Int) => (x * 0.9).toInt
println(transformData(sales, addTax))       // Add GST → List(1180, 2360, 3540)
println(transformData(sales, giveDiscount)) // Discount → List(900, 1800, 2700)
```

🎯 **ETL Perspective:**

- transformData is a reusable higher-order function.
- You can plug in different transformation logics (tax, discount, currency conversion, etc.) dynamically.

✅ Example 4: Function Composition

Combine functions for cleaner ETL logic:

```
val add10 = (x: Int) => x + 10
val multiply2 = (x: Int) => x * 2
val combined = add10.andThen(multiply2)
println(combined(5)) // ((5 + 10) * 2) = 30
```

🔹 Summary

Concept	Description	Example
First-class function	Function is treated as a value	val f = (x:Int) => x+1
Higher-order function	Takes/returns another function	list.map(x => x*2)
Anonymous function	No name, inline defined	(x:Int) => x*x
Function composition	Combine multiple functions	f.andThen(g)