# Scala

**Functional programming (FP)** is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It emphasizes the use of functions as first-class citizens, meaning functions can be assigned to variables, passed as arguments, and returned from other functions. Here are the key principles and features of functional programming:

### Key Principles

**1. **Immutability****
- **Definition**: Once a value is created, it cannot be changed.

- **Benefit**: Reduces side effects and makes code easier to reason about.

**2. **Pure Functions****
- **Definition**: Functions that, given the same input, will always return the same output and have no side effects (they don't alter any state or perform I/O operations).

- **Benefit**: Makes functions predictable and easier to test.

**3. **First-Class and Higher-Order Functions****

- **First-Class Functions**: Functions are treated as values that can be assigned to variables, passed as arguments, and returned from other functions.

- **Higher-Order Functions**: Functions that take other functions as parameters or return functions as results.

- **Benefit**: Encourages code reuse and abstraction.

**4. **Function Composition****

- **Definition**: Combining simple functions to build more complex ones.

- **Benefit**: Promotes modularity and code reuse.

**5. **Declarative Programming****
- **Definition**: Expresses logic without explicitly describing the control flow.

- **Benefit**: Code is more concise and easier to understand.

**6. **Referential Transparency****
- **Definition**: An expression that can be replaced with its value without changing the program's behavior.

- **Benefit**: Enhances predictability and reliability of code.

### Key Features in Practice

**1. **Immutable Data Structures****
- **Example**:

  Using immutable lists, sets, and maps instead of their mutable counterparts.

**2. **Pure Functions****
- **Example**: A pure function in Scala:

```
def add(a: Int, b: Int): Int = a + b
```

**3. **First-Class and Higher-Order Functions****
- **Example**: Using functions as arguments in Scala:

```
def applyOperation(a: Int, b: Int, operation: (Int, Int) => Int): Int = operation(a, b)
val sum = applyOperation(3, 4, _ + _)
```

**4. **Function Composition****
- **Example**: Composing functions in Scala:

```
val addOne: Int => Int = _ + 1
val double: Int => Int = _ * 2
val addOneAndDouble: Int => Int = addOne.andThen(double)
```

5. **Declarative Style**
   - **Example**: Using higher-order functions to process collections:

```
val numbers = List(1, 2, 3, 4, 5)
val doubled = numbers.map(_ * 2)
val evenNumbers = numbers.filter(_ % 2 == 0)
```

### **Benefits of Functional Programming**

- **Modularity**: Functions can be combined and reused, promoting modular code.

- **Maintainability**: Code is easier to read, understand, and maintain due to the emphasis on immutability and pure functions.

- **Concurrency**: Immutability and the absence of side effects make it easier to write concurrent and parallel programs.

- **Testability**: Pure functions are easier to test because they depend only on their inputs and have no side effects.

### Examples in Scala

Here are some examples that illustrate functional programming principles in Scala:

***Immutability and Pure Functions:****

```
val numbers = List(1, 2, 3, 4, 5)
val incrementedNumbers = numbers.map(_ + 1)        // This does not mutate the original list
println(incrementedNumbers)                        // List(2, 3, 4, 5, 6)
println(numbers)                                   // List(1, 2, 3, 4, 5)
```

****Function Composition:****

```
val addOne: Int => Int = _ + 1
val square: Int => Int = x => x * x
val addOneAndSquare: Int => Int = addOne.andThen(square)
println(addOneAndSquare(2))                // 9
```

****Declarative Style:****

```
val numbers = List(1, 2, 3, 4, 5)
val evenSquares = numbers.filter(_ % 2 == 0).map(x => x * x)
println(evenSquares)                    // List(4, 16)
```

Functional programming in Scala encourages writing clean, concise, and expressive code by leveraging these principles and features.

It helps in creating robust and maintainable software systems.

****First-Class and Higher-Order Functions:****

```
val double: Int => Int = _ * 2
val result = List(1, 2, 3).map(double)
println(result)             // List(2, 4, 6)
```