

Lecture 4 – Functional Programming

SWS121: Secure Programming

Jihyeok Park



2024 Spring

- Recall: Product Types and Algebraic Data Types
- **Basic Object-Oriented Programming**
 - Constructors
 - Traits
 - Overloading and Overriding
 - Access Modifiers
- **Advanced Object-Oriented Programming**
 - Objects
 - Companion Objects
 - Operators

1. Functions

- Methods vs Functions
- Eta Expansion
- Recursive Functions
- Tail-Call Optimization
- Default Parameter Values
- Nested Methods
- Multiple Parameter Lists

2. Pattern Matching

- Sealed Types
- Regular Expression Patterns
- Extractor Objects

3. Functional Error Handling

- Option Type
- Try Type
- Either Type

1. Functions

- Methods vs Functions
- Eta Expansion
- Recursive Functions
- Tail-Call Optimization
- Default Parameter Values
- Nested Methods
- Multiple Parameter Lists

2. Pattern Matching

- Sealed Types
- Regular Expression Patterns
- Extractor Objects

3. Functional Error Handling

- Option Type
- Try Type
- Either Type

In general, a **method** is a member of a class or object, and a **function** is a value that can be assigned to a variable.

```
def isEvenMethod(n: Int) = n % 2 == 0           // a method
val isEvenFunction = (n: Int) => n % 2 == 0     // a function
```

Methods should be used by calling them with parentheses.

```
isEvenMethod(2)                               // true
```

The **function** truly is an **object**, so we can use it just like any other variable, including passing it as an argument to a method.

```
List(1, 2, 3, 4, 5).filter(isEvenFunction)     // List(2, 4)
```

However, we can also pass a **method** as a function in Scala. Why?

```
List(1, 2, 3, 4, 5).filter(isEvenMethod)       // List(2, 4)
```

Eta expansion!

To utilize a method as a function, we need to create a **function** as a wrapper around the method.

```
List(1, 2, 3, 4, 5).filter(x => isEvenMethod(x)) // List(2, 4)
```

Or, we can simplify it using the **placeholder syntax** (`_`).

```
List(1, 2, 3, 4, 5).filter(isEvenMethod(_)) // List(2, 4)
```

It is called **eta expansion**, which converts a method into a function.

Scala compiler **automatically performs** the **eta expansion** for us.

```
List(1, 2, 3, 4, 5).filter(isEvenMethod) // List(2, 4)
```

It means that we can utilize a method as a function without explicitly converting it into a function in Scala.

In **imperative programming**, we use **loops** to iterate to recursively do something. For example, we can use a **while loop** to calculate the sum of the first n numbers.

```
def sum(n: Int): Int = {  
  var k = n  
  var sum = 0  
  while (k > 0) {  
    sum += k  
    k -= 1  
  }  
  sum  
}
```

```
sum(1000)
```

However, in **functional programming**, we use **recursive functions** (or methods) to recursively do something.

Here is a Scala code example with a **recursive method**:

```
def sum(n: Int, acc: Int): Int =  
  if (n < 1) acc  
  else sum(n - 1, n + acc)  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

Here is a Scala code example with a **recursive method**:

```
def sum(n: Int, acc: Int): Int =  
  if (n < 1) acc  
  else sum(n - 1, n + acc)  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02	999	x	sum
0x03	1000	acc	
0x04			
0x05			
	⋮		
0xFE			
0xFF			

Here is a Scala code example with a **recursive method**:

```
def sum(n: Int, acc: Int): Int =
  if (n < 1) acc
  else sum(n - 1, n + acc)

sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02	999	x	sum
0x03	1000	acc	
0x04	998	x	sum
0x05	1999	acc	
	⋮	⋮	
0xFE	873	x	sum
0xFF	118999	acc	

It fails with a **stack overflow** error.

However, is it really necessary to keep **all the stack frames**? **No!**

Scala supports **tail-call optimization** (TCO).

Here is a Scala code example with a **recursive method**:

```
def sum(n: Int, acc: Int): Int =  
  if (n < 1) acc  
  else sum(n - 1, n + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	1000	x	sum
0x01	0	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

The function call is in **tail-call position**.

It means that it **directly returns** the **last function call result** without any further computation.

Thus, we can safely **discard** the current stack frame **before** calling the function, and it is called **tail-call optimization (TCO)**.

Here is a Scala code example with a **recursive method**:

```
def sum(n: Int, acc: Int): Int =  
  if (n < 1) acc  
  else sum(n - 1, n + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	999	x	sum
0x01	1000	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
0xFE			
0xFF			

The function call is in **tail-call position**.

It means that it **directly returns** the **last function call result** without any further computation.

Thus, we can safely **discard** the current stack frame **before** calling the function, and it is called **tail-call optimization (TCO)**.

Here is a Scala code example with a **recursive method**:

```
def sum(n: Int, acc: Int): Int =  
  if (n < 1) acc  
  else sum(n - 1, n + acc) // tail-call  
  
sum(1000, 0)
```

A	M		
0x00	0	x	sum
0x01	500500	acc	
0x02			
0x03			
0x04			
0x05			
	⋮		
	⋮		
0xFE			
0xFF			

The function call is in **tail-call position**.

It means that it **directly returns** the **last function call result** without any further computation.

Thus, we can safely **discard** the current stack frame **before** calling the function, and it is called **tail-call optimization (TCO)**.

To ensure that the function call is in **tail-call position**, we can utilize the `scala.annotation.tailrec` annotation in Scala.

```
import scala.annotation.tailrec

@tailrec
def sum(n: Int, acc: Int): Int =
  if (n < 1) acc
  else sum(n - 1, n + acc) // tail-call
```

Scala compiler will check if the function is in **tail-call position** and optimize it accordingly.

```
@tailrec
def sum(n: Int): Int =
  if (n < 1) 0
  else n + sum(n - 1) // not a tail-call
```

If the function is not in the **tail-call position**, the compiler will **reject** the program during compilation.

However, passing the **accumulated value** `acc` to the function is not user-friendly.

One possible solution is to use a **default parameter value** for the accumulated value.

```
@tailrec
def sum(n: Int, acc: Int = 0): Int =
  if (n < 1) acc
  else sum(n - 1, n + acc) // tail-call

sum(1000)           // 500500
sum(1000, 0)       // 500500
```

We can call it using the **named argument** syntax to improve readability.

```
sum(1000, acc = 0) // 500500
sum(acc = 0, n = 1000) // 500500
```

Another possible solution is to use a **nested method** to **hide** the accumulated value from the user.

```
def sum(n: Int): Int = {  
  
  // nested method supporting tail-call optimization  
  @tailrec  
  def aux(n: Int, acc: Int): Int =  
    if (n < 1) acc  
    else aux(n - 1, n + acc) // tail-call  
  
  aux(n, 0)  
}  
  
sum(1000)           // 500500
```

It is another way of **encapsulation** in functional programming paradigm.

We **cannot access** the **nested method** (`aux`) from outside the enclosing method (`sum`).

We learned that functions are **first-class citizens** in Scala, and we can define a function that returns another function.

```
// `add` takes two arguments and returns an integer
val add = (x: Int, y: Int) => x + y
// `addN` takes a single argument and returns a function
val addN = (x: Int) => (y: Int) => x + y
```

Currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each take a single argument.

We can use **multiple parameter lists** to define a curried method similar to the curried function.

```
def addN(x: Int)(y: Int): Int = x + y

val add3 = addN(3)
add3(5)           // 3 + 5 = 8
addN(4)(6)       // 4 + 6 = 10
```

1. Functions

- Methods vs Functions
- Eta Expansion
- Recursive Functions
- Tail-Call Optimization
- Default Parameter Values
- Nested Methods
- Multiple Parameter Lists

2. Pattern Matching

- Sealed Types
- Regular Expression Patterns
- Extractor Objects

3. Functional Error Handling

- Option Type
- Try Type
- Either Type

We learned that Scala supports **pattern matching** to match a value against a pattern. It is a **more powerful** and **safer** version of the switch statement in Java.

```
enum Tree:
  case Leaf(value: Int)
  case Branch(left: Tree, value: Int, right: Tree)

// Load all constructors of the Tree enum
import Tree.*

// A recursive method counts the number of the given integer in a tree
def sum(t: Tree): Int = t match
  case Leaf(n) => n
  case Branch(l, n, r) => sum(l) + n + sum(r)

sum(Branch(Branch(Leaf(1), 2, Leaf(3)), 4, Leaf(5))) // 15 : Int
```

We can define a function using **pattern matching**.

```
val sum: Tree => Int = {  
  case Leaf(n) => n  
  case Branch(l, n, r) => sum(l) + n + sum(r)  
}
```

We can utilize this concept with **multiple parameter lists** as follows:

```
// `List.foldLeft` has two parameter lists:  
// (1) the initial value and  
// (2) a function used to fold the list.  
List(1, 2, 3, 4, 5).foldLeft(0)((x, y) => x + y)  
List(1, 2, 3, 4, 5).foldLeft(0)(_ + _) // placeholder syntax
```

```
// Add only even numbers by using pattern matching  
List(1, 2, 3, 4, 5).foldLeft(0) {  
  case (acc, n) if n % 2 == 0 => acc + n  
  case (acc, _) => acc  
}
```

Pattern matching is more than a programming concept; it's a **paradigm shift** in how we handle data and so crucial in functional programming.

- **Simple and concise code** – Compared to the traditional `if-else` statements or `switch` statements, it supports more complex patterns with 1) nested patterns, 2) guards, and 3) matching on tuples.
- **Enhanced readability** – It makes the code more readable and maintainable.
- **Reducing bugs** – The compiler can check if all cases are covered (exhaustive pattern matching) and warn if some cases are missing.

We can perform **pattern matching** on **tuples** to consider multiple different values at once in a single pattern.

```
// Merge two lists
val merge: (List[Int], List[Int]) => List[Int] = {
  case (lh :: lt, rh :: rt) => (lh + rh) :: merge(lt, rt)
  case (Nil, right) => right
  case (left, Nil) => left
}
```

We can **safely extract** the values from a tuple using the **pattern matching** syntax and ignore some values using the underscore (`_`).

```
val triple = (1, "abc", true)

// Extract values from a tuple
val (x, y, z) = triple
// x = 1, y = "abc", z = true

// Ignore the second value
val (a, _, c) = triple
```

Similarly, we can perform **pattern matching** on **case classes** to extract the values from the case class without using the dot notation.

```
case class Person(name: String, age: Int)
val person = Person("Jihyeok Park", 32)

// Extract values from a case class
val Person(name, age) = person    // name = "Jihyeok Park", age = 32
```

We can also use the **nested pattern matching** to extract the values from the case class.

```
case class Card(person: Person, number: Int)
val card = Card(Person("Jihyeok Park", 32), 1234)

// Safely extract the all information of the card at once
val Card(Person(name, age), number) = card
// val name    = card.person.name
// val age     = card.person.age
// val number  = card.number
```

Another strong point of **pattern matching** is that it can check if all cases are covered (**exhaustive matching**).

Assume we forgot to handle the case `(_ :: _, Nil)` in merge function.

```
val merge: (List[Int], List[Int]) => List[Int] = {  
  case (lh :: lt, rh :: rt) => (lh + rh) :: merge(lt, rt)  
  case (Nil, right) => right  
}
```

Then, the compiler will warn us the pattern matching is **not exhaustive**:

```
[E029] Pattern Match Exhaustivity Warning:  
  case (lh :: lt, rh :: rt) => (lh + rh) :: merge(lt, rt)  
  ^  
  match may not be exhaustive.  
  
It would fail on pattern case: (List(_, _*), Nil)
```

Exhaustive matching is a powerful feature that can prevent many bugs.

Scala compiler also checks whether each case is **reachable** or not.

Assume we accidentally added the already covered case (Nil, Nil) in merge function.

```
val merge: (List[Int], List[Int]) => List[Int] = {  
  case (lh :: lt, rh :: rt) => (lh + rh) :: merge(lt, rt)  
  case (Nil, right) => right  
  case (left, Nil) => left  
  case (Nil, Nil) => Nil  
}
```

Then, the compiler will warn us the last case is **unreachable**:

```
[E030] Match case Unreachable Warning:  
  case (Nil, Nil) => Nil  
  ~~~~~  
  Unreachable case
```

To ensure the pattern matching is **exhaustive**, we can use the **sealed** keyword to restrict the possible subtypes of a supertype.

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape
case class Triangle(base: Double, height: Double) extends Shape

// Calculate the area of the shape
def area(shape: Shape): Double = shape match
  case Circle(r) => math.Pi * r * r
  case Rectangle(w, h) => w * h
  // Triangle(_, _) case is missing and the compiler will warn us
```

The **sealed** type can only be extended in the **same file** and prevents the creation of new subtypes outside the file.

Conceptually, **enums** is a **syntactic sugar** for a **sealed** type with a set of **case classes** (or **objects**).

```
enum Tree:  
  case Leaf(value: Int)  
  case Branch(left: Tree, value: Int, right: Tree)
```

will be conceptually desugared into:

```
sealed abstract class Tree  
object Tree:  
  case class Leaf(value: Int) extends Tree  
  case class Branch(left: Tree, value: Int, right: Tree) extends Tree
```

Thus, the **exhaustiveness** of the pattern matching is guaranteed also for the **enums** in Scala.

Scala supports **regular expressions** with a method `r` on a string.

```
// email pattern: <username>@<domain>
val emailPattern: Regex = """"^(\w+)@(\w+(\.\w+)+)$""".r
// phone pattern: <area code>-<exchange code>-<number>
val phonePattern: Regex = """"^(\d{3}-\d{3}-\d{4})$""".r
```

We can use **regular expression patterns** in Scala to match a string against a regular expression.

```
def detectContact(contact: String): String = contact match
  case emailPattern(username, domain) => s"Email: $username at $domain"
  case phonePattern(phone) => s"Phone: $phone"
  case _ => "Unknown contact"
```

```
detectContact("abc@domain.com") // Email: abc at domain.com
detectContact("123-456-7890") // Phone: 123-456-7890
detectContact("abc#domain.com") // Unknown contact
detectContact("123-45-43") // Unknown contact
detectContact("invalid") // Unknown contact
```

We can define **user-defined patterns** using **extractor objects**.

Extractor objects are objects that define a method called `unapply` or `unapplySeq` to extract values from a given object.

```
enum Tree:
  case Leaf(value: Int)
  case Branch(left: Tree, value: Int, right: Tree)

// An extractor object for the leaf node whose value is even
object EvenLeaf:
  def unapply(tree: Tree): Option[Int] = tree match
    case Leaf(n) if n % 2 == 0 => Some(n)
    case _ => None

def sumEvenLeaves(tree: Tree): Int = tree match
  case EvenLeaf(n) => n           // extract only if the leaf value is even
  case Branch(l, _, r) => sumEvenLeaves(l) + sumEvenLeaves(r)
  case Leaf(_) => 0

sumEvenLeaves(Branch(Leaf(2), 3, Branch(Leaf(4), 6, Leaf(7)))) // 6
```

The `unapplySeq` method can be used to extract a **sequence of values**.

```
// An extractor object for string sequence split by a comma
object CommaSeparated:
  def unapplySeq(s: String): Option[Seq[String]] =
    Some(s.split(',').toSeq)

// Parse a string having at most three parts separated by a comma
def parse(s: String): String = s match
  case CommaSeparated(a, b, c) => s"Three parts: $a, $b, $c"
  case CommaSeparated(a, b)   => s"Two parts: $a, $b"
  case CommaSeparated(a)     => s"One part: $a"
  case _                     => "Invalid format"

parse("a,b,c")      // Three parts: a, b, c
parse("x,y")        // Two parts: x, y
parse("k")          // One part: k
parse("a,b,c,d")    // Invalid format
```

1. Functions

- Methods vs Functions
- Eta Expansion
- Recursive Functions
- Tail-Call Optimization
- Default Parameter Values
- Nested Methods
- Multiple Parameter Lists

2. Pattern Matching

- Sealed Types
- Regular Expression Patterns
- Extractor Objects

3. Functional Error Handling

- Option Type
- Try Type
- Either Type

Functional programming supports a **safer** and **more robust** way of **handling errors** rather than just using exceptions and try-catch blocks.

Assume we want to parse an integer from a string and return 0 if it fails.

In general, we can handle the exception with the try-catch block:

```
def makeInt(s: String): Int =  
  try Integer.parseInt(s)  
  catch case _: Exception => 0
```

However, it is not really accurate. If we get 0 as a result, we don't know whether it is because the string was not a valid integer or the input is "0".

We can use the **Option** type to handle this situation more accurately.

The `Option[T]` type is a **container** that may contain a single value of type `T` or no value.

- **Some(x)** – a container that holds a single value `x` of type `T`
- **None** – a container that holds no value

```
def makeInt(s: String): Option[Int] =  
  try Some(Integer.parseInt(s))  
  catch case _: Exception => None
```

We can apply pattern matching to handle the `Option` type.

```
val x: String = ...  
  
makeInt(x) match  
  case Some(n) => println(s"Integer: $n")  
  case None => println("Not an integer")
```

We can utilize **for comprehension** to handle multiple Option values.

```
def addStrings(s1: String, s2: String, s3: String): Option[Int] = for {  
  a <- makeInt(s1)  
  b <- makeInt(s2)  
  c <- makeInt(s3)  
} yield a + b + c  
  
addStrings("1", "2", "3")      // Some(6)  
addStrings("x", "2", "3")     // None
```

Option is not the only solution for error handling in Scala.

We will learn how it works later in this course.

In addition, Scala supports other types like Try or Either types for functional error handling.

We can keep which exception occurred using the `Try[T]` type.

- **Success(x)** – a container that holds a single value `x` of type `T`
- **Failure(e)** – a container that holds an exception `e`

```
import scala.util.Try

def makeInt(s: String): Try[Int] = Try(Integer.parseInt(s))
makeInt("123")                // Success(123)
makeInt("abc")                // Failure(java.lang.NumberFormatException)
```

```
def addStrings(s1: String, s2: String, s3: String): Try[Int] = for {
  a <- makeInt(s1)
  b <- makeInt(s2)
  c <- makeInt(s3)
} yield a + b + c

addStrings("1", "2", "3") // Success(6)
addStrings("x", "2", "3") // Failure(java.lang.NumberFormatException)
```

We can keep our own data for failure cases using the `Either[L, R]` type.

- **Left(x)** – a value `x` of type `L` for failure cases
- **Right(x)** – a value `x` of type `R` for success cases

```
def divide(x: Int, y: Int): Either[String, Int] =  
  if (y == 0) Left("Cannot divide by zero")  
  else Right(x / y)
```

```
divide(10, 2)           // Right(5)  
divide(10, 0)          // Left("Cannot divide by zero")
```

```
def divideThree(x: Int, y: Int, z: Int): Either[String, Int] = for {  
  a <- divide(x, y)  
  b <- divide(a, z)  
} yield b
```

```
divideThree(10, 2, 5)   // Right(1)  
divideThree(10, 0, 5)  // Left("Cannot divide by zero")
```

1. Functions

- Methods vs Functions
- Eta Expansion
- Recursive Functions
- Tail-Call Optimization
- Default Parameter Values
- Nested Methods
- Multiple Parameter Lists

2. Pattern Matching

- Sealed Types
- Regular Expression Patterns
- Extractor Objects

3. Functional Error Handling

- Option Type
- Try Type
- Either Type

- Immutable Collections

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>