Lecture 1 – Basic Introduction of Scala

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

The goal of this course is:

To learn **essential concepts** of **programming languages**

- How?

[1] https://docs.scala-lang.org/scala3/book/introduction.html

Recall

The goal of this course is:

To learn **essential concepts** of **programming languages**

- How? You will learn how to:
  - **design** programming languages in a **mathematical** way.
  - **implement** their **interpreters** using **Scala**.

[1] https://docs.scala-lang.org/scala3/book/introduction.html

Recall

The goal of this course is:

To learn **essential concepts** of **programming languages**

- How? You will learn how to:
  - **design** programming languages in a **mathematical** way.
  - **implement** their **interpreters** using **Scala**.
- Before entering the world of PL,

Let's learn **Scala**

(If you interested in more details, please see Scala 3 Book.[1])

[1] https://docs.scala-lang.org/scala3/book/introduction.html

What is Scala?

Scala stands for **Sca**lable **La**nguage.

- A general-purpose programming language
- **Java Virtual Machine (JVM)**-based language
- A **statically typed** language
- An **object-oriented programming (OOP)** language
- A **functional programming (FP)** language

Read-Eval-Print-Loop (REPL)

Please download and install them using the following links.

- **JDK** – https://www.oracle.com/java/technologies/downloads/
- **sbt** – https://www.scala-sbt.org/download.html
- **Scala REPL** – https://www.scala-lang.org/download/

```
~$ scala
Welcome to Scala 3.5.0 (20.0.2, Java Java HotSpot(TM) 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala>
```

Functional Programming

We will use **functional programming** (FP) by **reducing unexpected side effects** and **increasing code readability**.

- **Immutable Variables**
  - Variables are immutable by default
- **Pure Functions**
  - Functions do not have side effects
- **First-class Functions**
  - Functions are first-class citizens (i.e., functions are values)
- **Functional Error Handling**
  - Using `Option` for error handling

Contents

Contents

Basic Data Types

```scala
42           // 42    : Int       (32-bit signed integer)
3.7          // 3.7   : Double    (64-bit double-precision floating-point)
true         // true  : Boolean
false        // false: Boolean
'c'          // 'c'   : Char      (16-bit Unicode character)
"abc"        // "abc": String     (a sequence of characters)
()           // ()    : Unit      (meaningless value - similar to `void`)
```

Basic Data Types

```
42            // 42   : Int     (32-bit signed integer)
3.7           // 3.7  : Double  (64-bit double-precision floating-point)
true          // true : Boolean
false         // false: Boolean
'c'           // 'c'  : Char    (16-bit Unicode character)
"abc"         // "abc": String  (a sequence of characters)
()            // ()   : Unit    (meaningless value - similar to `void`)
```

You can perform following operations on these data types.

```
1 + 2 * 3     // 7    : Int     (addition and multiplication)
11 / 3        // 3    : Int     (quotient of division)
11 % 3        // 2    : Int     (remainder of division)
1 < 2         // true : Boolean (comparison)
1 == 3        // false: Boolean (equality)
true && false // false: Boolean (logical AND)
true || false // true : Boolean (logical OR)
!true         // false: Boolean (logical NOT)
"abc".length  // 3    : Int     (length of a string)
println("Hi") // ()   : Unit    (side effect: printing "Hi")
```

Immutable Variables

variable name   initial value

```scala
val x: Int = 1
```

variable type

```scala
// An immutable variable `x` of type `Int` with 1
val x: Int = 1
x + 2            // 1 + 2 == 3 : Int
x = 2            // Type Error: Reassignment to val `x`

// Type Inference: `Int` is inferred from `1`
val y = 1        // y: Int

// Type Mismatch Error: `Boolean` required but `Int` found: 42
val c: Boolean = 42
```

Mutable Variables

While Scala supports mutable variables (`var`), **DO NOT USE MUTABLE VARIABLES IN THIS COURSE** because it is against the **functional programming** paradigm.

```
var x: Int = 1
```

```
// A mutable variable `x` of type `Int` with 1
var x: Int = 1
x + 2               // 1 + 2 == 3 : Int

// You can reassign a mutable variable `x`
x = 2               // x == 2
x + 2               // 2 + 2 == 4 : Int
```

Methods

method name parameter type method body

```
def add(x: Int, y: Int): Int = x + y
```

parameter name return type

```
// A method `add` of type `(Int, Int) => Int`
// It means that `add` takes two `Int` arguments and returns an `Int`
def add(x: Int, y: Int): Int = x + y
add(1, 2)        // 1 + 2 == 3  : Int
add(5, 6)        // 5 + 6 == 11 : Int

// Type Error: wrong number of arguments
add(1)           // Too few arguments
add(1, 2, 3)     // Too many arguments

// Type Mismatch Error: `Int` required but `String` found: "abc"
add(1, "abc")
```

Conditionals

condition then body else body

```
if (x < 0) -x else x
```

```scala
// A method `abs` which takes an `Int` and returns its absolute value
def abs(x: Int): Int = if (x < 0) -x else x

abs(42)           // 42 : Int
abs(-42)          // 42 : Int
```

Conditionals

condition then body else body

```scala
if (x < 0) -x else x
```

```scala
// A method `abs` which takes an `Int` and returns its absolute value
def abs(x: Int): Int = if (x < 0) -x else x

abs(42)          // 42 : Int
abs(-42)         // 42 : Int
```

Note that it is a conditional **expression** not a **statement** similar to the ternary operator (`x ? y : z`) in other languages.

```scala
2 * (if (true) 3 else 5)   // 2 * 3 == 6  : Int
```

Recursions

You can **recursively** invoke a method with a conditional expression.

```scala
// A recursive method `sum` that adds all the integers from 1 to n
def sum(n: Int): Int =
  if (n < 1) 0
  else sum(n - 1) + n

sum(10)                           // 55       : Int
sum(100)                          // 5050     : Int
```

Recursions

You can **recursively** invoke a method with a conditional expression.

```
// A recursive method `sum` that adds all the integers from 1 to n
def sum(n: Int): Int =
  if (n < 1) 0
  else sum(n - 1) + n

sum(10)                          // 55       : Int
sum(100)                         // 5050     : Int
```

You can use either **indentation** (above) or **curly braces** (below) for a block of expressions as follows.

```
def sum(n: Int): Int = {
  if (n < 1) 0
  else sum(n - 1) + n
}
```

Recursions

While Scala supports `while` loops, **DO NOT USE WHILE LOOPS IN THIS COURSE** because it is against the **functional programming** paradigm.

```
// Sum of all the numbers from 1 to n
def sum(n: Int): Int = {
  var s: Int = 0
  var k: Int = 1
  while (k <= n) {
    s = s + k
    k = k + 1
  }
  s
}

sum(10)  // 55   : Int
sum(100) // 5050 : Int
```

Contents

Product Types – `case class`

type name　　　field type

```scala
case class Point(x: Int, y: Int, color: String)
```

field name

A `case class` defines a **product type** with:

- its **type name** (e.g., `Point`)
- its **constructor** (e.g., `Point`)

Product Types – `case class`

type name   field type

```scala
case class Point(x: Int, y: Int, color: String)
```

field name

A `case class` defines a **product type** with:

- its **type name** (e.g., `Point`)
- its **constructor** (e.g., `Point`)

```scala
// A `Point` instance whose fields: x = 3, y = 4, and color = "RED"
val p: Point = Point(3, 4, "RED")

// You can access fields using the dot operator
p.x               // 3       : Int
p.color           // "RED"   : String

// Fields are immutable by default
p.x = 5           // Type Error: Reassignment to val `x`
```

Algebraic Data Types (ADTs) – enum

type name

variants

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

An enum defines an **algebraic data type (ADT)** with:

- its **type name** (e.g., Tree)
- its **constructors** of variants (e.g., Leaf, Branch)

Algebraic Data Types (ADTs) – `enum`

type name

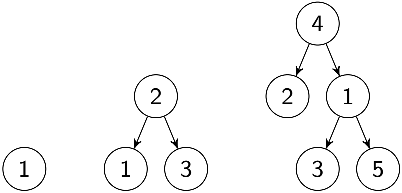
variants

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

An `enum` defines an **algebraic data type (ADT)** with:

- its **type name** (e.g., `Tree`)
- its **constructors** of variants (e.g., `Leaf`, `Branch`)

```
import Tree.* // Import all constructors for variants of `Tree`
val tree1: Tree = Leaf(1)
val tree2: Tree = Branch(Leaf(1), 2, Leaf(3))
val tree3: Tree = Branch(Leaf(2), 4, Branch(Leaf(3), 1, Leaf(5)))
```



Pattern Matching

You can **pattern match** on algebraic data types (ADTs).

```scala
// A recursive method computes the sum of all the values 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(Leaf(1), 2, Leaf(3)))                       // 6  : Int
sum(Branch(Branch(Leaf(1), 2, Leaf(3)), 4, Leaf(5)))  // 15 : Int
```

Pattern Matching

You can **pattern match** on algebraic data types (ADTs).

```
// A recursive method computes the sum of all the values 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(Leaf(1), 2, Leaf(3)))                        // 6  : Int
sum(Branch(Branch(Leaf(1), 2, Leaf(3)), 4, Leaf(5)))  // 15 : Int
```

You can **ignore** some components using an underscore (_) and use **if guards** to add conditions to patterns.

```
// A method checks whether a tree is a branch whose value is even
def isEvenBranch(t: Tree): Boolean = t match
  case Branch(_, n, _) if n % 2 == 0 => true
  case _                             => false

isEvenBranch(Leaf(1))                                 // false : Boolean
isEvenBranch(Branch(Leaf(1), 2, Leaf(3)))            // true  : Boolean
```

Pattern Matching

Here is another example of pattern matching on ADTs.

```
// An ADT for natural numbers
enum Nat:
  case Z          // Zero
  case S(n: Nat)  // Successor of a natural number

import Nat.* // Import constructors `Z` and `S` for variants of `Nat`
```

Pattern Matching

Here is another example of pattern matching on ADTs.

```
// An ADT for natural numbers
enum Nat:
  case Z          // Zero
  case S(n: Nat)  // Successor of a natural number

import Nat.* // Import constructors `Z` and `S` for variants of `Nat`
```

We can also use **nested pattern matching**.

```
// A recursive method adds two natural numbers
def isEven(x: Nat): Boolean = x match
  case Z       => true
  case S(S(y)) => isEven(y)   // nested pattern matching
  case _       => false

isEven(Z)                     // true  : Boolean
isEven(S(Z))                  // false : Boolean
isEven(S(S(Z)))               // true  : Boolean
```

Contents

First-Class Functions

parameter name **function body**

$\texttt{(}\boxed{\texttt{x}}\texttt{: }\boxed{\texttt{Int}}\texttt{) => }\boxed{\texttt{x + 1}}$

parameter type

A **function** is a **first-class citizen** (i.e., a function is a value) in Scala.

```
// A function that increments its input
(x: Int) => x + 1                        // a function `Int => Int`
((x: Int) => x + 1)(3)                   // 3 + 1 = 4 : Int
```

First-Class Functions

parameter name | function body

`(x: Int) => x + 1`

parameter type

A **function** is a **first-class citizen** (i.e., a function is a value) in Scala.

```scala
// A function that increments its input
(x: Int) => x + 1                       // a function `Int => Int`
((x: Int) => x + 1)(3)                  // 3 + 1 = 4 : Int
```

We can **store** a function in a variable.

```scala
val inc: Int => Int = (x: Int) => x + 1
inc(3)                                  // 3 + 1 = 4 : Int
val inc: Int => Int = x => x + 1        // Type Inference: `x` is `Int`
inc(3)                                  // 3 + 1 = 4 : Int
val inc: Int => Int = _ + 1             // Placeholder Syntax
inc(3)                                  // 3 + 1 = 4 : Int
```

First-Class Functions

We can **pass** a function to a method (or function) as an **argument**.

```scala
// A method `twice` that applies the function `f` twice to `x`
def twice(f: Int => Int, x: Int): Int = f(f(x))
twice(inc, 5)                  // inc(inc(5)) = 5 + 1 + 1 = 7 : Int

// You can pass a function to `twice`
twice((x: Int) => x + 1, 5)  // 7 : Int
twice(x => x + 1, 5)           // 7 : Int - Type Inference: `x` is `Int`
twice(_ + 1, 5)                // 7 : Int - Placeholder Syntax
```

First-Class Functions

We can **pass** a function to a method (or function) as an **argument**.

```scala
// A method `twice` that applies the function `f` twice to `x`
def twice(f: Int => Int, x: Int): Int = f(f(x))
twice(inc, 5)                   // inc(inc(5)) = 5 + 1 + 1 = 7 : Int

// You can pass a function to `twice`
twice((x: Int) => x + 1, 5)     // 7 : Int
twice(x => x + 1, 5)            // 7 : Int - Type Inference: `x` is `Int`
twice(_ + 1, 5)                 // 7 : Int - Placeholder Syntax
```

We can **return** a function from a method (or function).

```scala
// A function `addN` returns a function that adds `n`
val addN = (n: Int) => (x: Int) => x + n
val add2 = addN(2)              // add2:          Int => Int
add2(3)                         // 3 + 2 = 5      : Int
addN(7)(5)                      // 5 + 7 = 12     : Int
twice(add2, 5)                  // 5 + 2 + 2 = 9  : Int
twice(addN(7), 5)               // 5 + 7 + 7 = 19: Int
```

Contents

Lists

`List[T]` type is an **immutable** sequence of elements of type `T`.

```scala
val list: List[Int] = List(3, 1, 2, 4)
```

Lists

`List[T]` type is an **immutable** sequence of elements of type `T`.

```
val list: List[Int] = List(3, 1, 2, 4)
```

We can define a list using `::` (cons) and `Nil` (empty list).

```
val list = 3 :: 1 :: 2 :: 4 :: Nil
```

Lists

`List[T]` type is an **immutable** sequence of elements of type `T`.

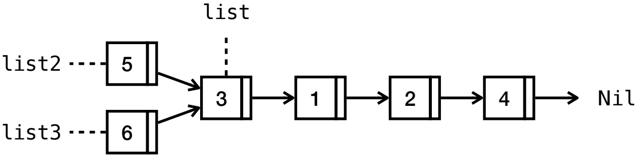
```scala
val list: List[Int] = List(3, 1, 2, 4)
```

We can define a list using `::` (cons) and `Nil` (empty list).

```scala
val list = 3 :: 1 :: 2 :: 4 :: Nil
```

Lists are immutable.

```scala
val list2 = 5 :: list   // List(5, 3, 1, 2, 4): List[Int]
val list3 = 6 :: list   // List(6, 3, 1, 2, 4): List[Int]
```



Lists

We can **pattern match** on lists.

```
val list: List[Int] = 3 :: 1 :: 2 :: 4 :: Nil

// Get the second element of the list or 0
def getSnd(list: List[Int]): Int = list match
  case _ :: x :: _ => x
  case _           => 0

getSnd(list)                  // 1 : Int

// Filter odd integers and double them in the list
def filterOddAndDouble(list: List[Int]): List[Int] = list match
  case Nil                      => Nil
  case x :: xs if x % 2 == 1   => x * 2 :: filterOddAndDouble(xs)
  case _ :: xs                  => filterOddAndDouble(xs)

filterOddAndDouble(list)   // List(6, 2) : List[Int]
```

Lists – Operations

```scala
// A list of integers: 3, 1, 2, 4
val list: List[Int] = List(3, 1, 2, 4)

// Operations/functions on lists
list.length                       // 4                       : Int
list.map(_ * 2)                   // List(6, 2, 4, 8)        : List[Int]
list.filter(_ % 2 == 1)           // List(3, 1)              : List[Int]
list.foldLeft(0)(_ + _)           // 0 + 3 + 1 + 2 + 4 = 10 : Int
list.flatMap(x => List(x, -x))    // List(3, -3, ..., 4, -4): List[Int]
list.map(x => List(x, -x)).flatten // List(3, -3, ..., 4, -4): List[Int]
```

Lists – Operations

```scala
// A list of integers: 3, 1, 2, 4
val list: List[Int] = List(3, 1, 2, 4)

// Operations/functions on lists
list.length                         // 4                      : Int
list.map(_ * 2)                     // List(6, 2, 4, 8)       : List[Int]
list.filter(_ % 2 == 1)             // List(3, 1)             : List[Int]
list.foldLeft(0)(_ + _)             // 0 + 3 + 1 + 2 + 4 = 10 : Int
list.flatMap(x => List(x, -x))      // List(3, -3, ..., 4, -4): List[Int]
list.map(x => List(x, -x)).flatten  // List(3, -3, ..., 4, -4): List[Int]
```

```scala
// Redefine `filterOddAndDouble` using `filter` and `map`
def filterOddAndDouble(list: List[Int]): List[Int] =
  list
    .filter(_ % 2 == 1)
    .map(_ * 2)

filterOddAndDouble(list)        // List(6, 2)                 : List[Int]
```

Options

While Scala supports `null` to represent the absence of a value, **DO NOT USE NULL IN THIS COURSE**.

Options

While Scala supports `null` to represent the absence of a value, **DO NOT USE NULL IN THIS COURSE**.

Instead, an **option** (`Option[T]`) is a container that may or may not contain a value of type `T`:

1. `Some(x)` represents a value `x` and
2. `None` represents the absence of a value

```
val some: Option[Int] = Some(42)
val none: Option[Int] = None

// Operations/functions on options
some.map(_ + 1)      // Some(43)        : Option[Int]
none.map(_ + 1)      // None            : Option[Int]
some.getOrElse(7)    // 42              : Int
none.getOrElse(7)    // 7               : Int
some.fold(7)(_ * 2) // 42 * 2 = 84     : Int
none.fold(7)(_ * 2) // 7               : Int
```

Pairs

A **pair** `(T, U)` is a container that contains two values of types `T` and `U`:

```
val pair: (Int, String) = (42, "foo")

// You can construct pairs using `->`
42 -> "foo" == pair  // true          : Boolean
true -> 42           // (true, 42)    : (Boolean, Int)

// Operations/functions on options
pair(0)              // 42            : Int     - NOT RECOMMENDED
pair(1)              // "foo"         : String  - NOT RECOMMENDED

// Pattern matching on pairs
val (x, y) = pair    // x == 42 and y == "foo"
```

Maps and Sets

A **map** (`Map[K, V]`) is a mapping from keys of type `K` to values of type `V`:

```
val map: Map[String, Int] = Map("a" -> 1, "b" -> 2)

map + ("c" -> 3) // Map("a" -> 1, "b" -> 2, "c" -> 3) : Map[String, Int]
map - "a"        // Map("b" -> 2)                     : Map[String, Int]
map.get("a")     // Some(1)                           : Option[Int]
map.get("c")     // None                              : Option[Int]
```

A **set** (`Set[T]`) is a collection of distinct elements of type `T`:

```
val set1: Set[Int] = Set(1, 2, 3)
val set2: Set[Int] = Set(2, 3, 5)

set1 + 4             // Set(1, 2, 3, 4) : Set[Int]
set1 + 2             // Set(1, 2, 3)    : Set[Int]
set1 - 2             // Set(1, 3)       : Set[Int]
set1.contains(2)     // true            : Boolean
set1 ++ set2         // Set(1, 2, 3, 5) : Set[Int]
set1.intersect(set2) // Set(2, 3)       : Set[Int]
```

For Comprehensions

```scala
val xlist = List(1, 2, 3)
val ylist = List(4, 5, 6)

// Filter pairs of elements whose sum is even and multiply them
xlist.flatMap(x =>
  ylist
    .filter(y => (x + y) % 2 == 0)
    .map(y => x * y)
)                            // List(5, 8, 12, 15)         : List[Int]
```

[2] https://docs.scala-lang.org/tour/for-comprehensions.html

For Comprehensions

```scala
val xlist = List(1, 2, 3)
val ylist = List(4, 5, 6)

// Filter pairs of elements whose sum is even and multiply them
xlist.flatMap(x =>
  ylist
    .filter(y => (x + y) % 2 == 0)
    .map(y => x * y)
)                           // List(5, 8, 12, 15)         : List[Int]
```

A **for comprehension**[2] is a syntactic sugar for nested `map`, `flatMap`, and `filter` operations:

```scala
for {
  x <- xlist
  y <- ylist
  if (x + y) % 2 == 0
} yield x * y               // List(5, 8, 12, 15)         : List[Int]
```

[2] https://docs.scala-lang.org/tour/for-comprehensions.html

Homework #1

- Please see this document on GitHub:

`https://github.com/ku-plrg-classroom/docs/tree/main/scala/scala-tutorial`

- The due date is 23:59 on Sep. 25 (Wed.).
- Please only submit `Implementation.scala` file to **Blackboard**.

Summary

1. Basic Features
   - Basic Data Types
   - Variables
   - Methods
   - Conditionals
   - Recursions
2. User-Defined Data Types
   - Product Types – `case class`
   - Algebraic Data Types (ADTs) – `enum`
   - Pattern Matching
3. First-Class Functions
4. Immutable Collections
   - Lists
   - Options and Pairs
   - Maps and Sets
   - For Comprehensions

Next Lecture

- Syntax and Semantics (1)

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