Lecture 2 – Basic Introduction of Scala

COSE215: Theory of Computation

Jihyeok Park

PLRG

2024 Spring

Recall

1. Mathematical Notations
   - Notations in Logics
   - Notations in Set Theory
2. Inductive Proofs
   - Inductions on Integers
   - Structural Inductions
   - Mutual Inductions
3. Notations in Languages
   - Symbols & Words
   - Languages

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
~$ scala
Welcome to Scala 3.3.0 (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

Integers and Doubles

`Int` type represents a **32-bit signed integer** (-2^{31} to $2^{31} - 1$).

```
42                  // 42   : Int

// Operations for integers
1 + 2               // 3    : Int        (integer addition)
1 - 2               // -1   : Int        (integer subtraction)
3 * 4               // 12   : Int        (integer multiplication)
5 / 2               // 2    : Int        (integer division)
5 % 2               // 1    : Int        (integer modulus)
```

Integers and Doubles

Int type represents a **32-bit signed integer** (-2^{31} to $2^{31} - 1$).

```
42                  // 42   : Int

// Operations for integers
1 + 2               // 3    : Int       (integer addition)
1 - 2               // -1   : Int       (integer subtraction)
3 * 4               // 12   : Int       (integer multiplication)
5 / 2               // 2    : Int       (integer division)
5 % 2               // 1    : Int       (integer modulus)
```

Double type represents a **64-bit double-precision floating-point**.

```
3.7                 // 3.7  : Double

// Operations for doubles
1.1 + 2.2           // 3.3  : Double    (double addition)
1.1 - 2.2           // -1.1 : Double    (double subtraction)
3.3 * 4.4           // 14.52: Double    (double multiplication)
5.5 / 2.2           // 2.5  : Double    (double division)
```

Booleans and Unit

Boolean type represents a `true` or `false` value.

```
true                // true : Boolean
false               // false: Boolean

// Operations for booleans
true && false       // false: Boolean  (logical AND)
true || false       // true : Boolean  (logical OR)
!true               // false: Boolean  (logical NOT)

// Numerical comparison operations producing booleans
1 < 2               // true : Boolean  (less than)
1 <= 2              // true : Boolean  (less than or equal to)
1 == 2              // false: Boolean  (equal to)
1 != 2              // true : Boolean  (not equal to)
```

Booleans and Unit

Boolean type represents a `true` or `false` value.

```
true                    // true : Boolean
false                   // false: Boolean

// Operations for booleans
true && false           // false: Boolean  (logical AND)
true || false           // true : Boolean  (logical OR)
!true                   // false: Boolean  (logical NOT)

// Numerical comparison operations producing booleans
1 < 2                   // true : Boolean  (less than)
1 <= 2                  // true : Boolean  (less than or equal to)
1 == 2                  // false: Boolean  (equal to)
1 != 2                  // true : Boolean  (not equal to)
```

Unit indicates **no meaningful information** and has one instance `()`.

```
()                      // ()    : Unit
println("Hello")        // ()    : Unit      (side effect: printing "Hello")
```

Strings

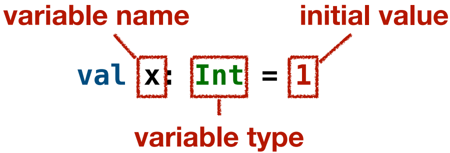
PLRG

`Char` represents a **16-bit Unicode character**,
and `String` represents an **immutable sequence of characters** (`Char`).

```
'c'                 // 'c'         : Char
"abc"               // "abc"       : String

// Operations for strings
"abc"(1)            // 'b'         : Char     (unsafe indexing)
"abc" + "def"       // "abcdef"    : String   (string concatenation)
"abc" * 3           // "abcabcabc": String   (string repetition)
"abc".length        // 3           : Int      (string length)
"abc".reverse       // "cba"       : String   (string reverse)
"abc".take(2)       // "ab"        : String   (take first two characters)
"abc".drop(2)       // "c"         : String   (drop first two characters)
"abc".toUpperCase   // "ABC"       : String   (convert to upper case)
"ABC".toLowerCase   // "abc"       : String   (convert to lower case)
```

Immutable Variables




```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

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

parameter name return type

```scala
// 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")
```

Recursion

You can **recursively** invoke a method.

```
// 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
```

Recursion

You can **recursively** invoke a method.

```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
```

condition   then body   else body

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

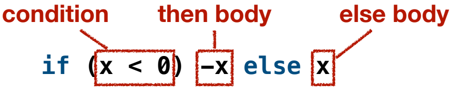
where **conditional expressions** (`if-else`) control the flow of execution.

Recursion

You can **recursively** invoke a method.

```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
```



where **conditional expressions** (`if-else`) control the flow of execution. Note that it is a conditional **expression** not a **statement** similar to the ternary operator (`x ? y : z`) in other languages.

```scala
"01" * (if (true) 3 else 7)   // "01" * 3 == "010101" : String
```

Recursion

PLRG

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 Classes

A `case class` defines a **product type** with named fields.

type name · field type · field name

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

```scala
// A case class `Point` having `x`, `y`, and `color` fields
// whose types are `Int`, `Int`, and `String`, respectively
case class Point(x: Int, y: Int, color: String)

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

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

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

Algebraic Data Types (ADTs) – Enumerations

An **algebraic data type (ADT)** is a sum of product types, and you can define it using **enumerations** (`enum`) in Scala.

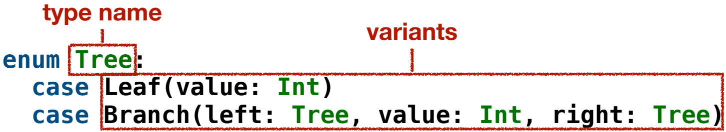
type name

variants

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

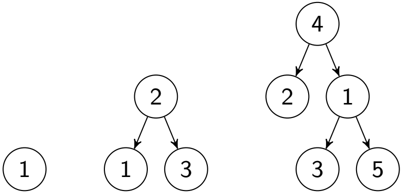
Algebraic Data Types (ADTs) – Enumerations

An **algebraic data type (ADT)** is a sum of product types, and you can define it using **enumerations** (`enum`) in Scala.




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

```scala
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).

```
// 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
```

Pattern Matching

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

```
// 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
```

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

```
// A recursive method counts the number of the given integer in a tree
def count(t: Tree, x: Int): Int = t match
  case Leaf(n)          if n == x  => 1
  case Leaf(_)                     => 0
  case Branch(l, n, r) if n == x  => count(l, x) + 1 + count(r, x)
  case Branch(l, _, r)             => count(l, x) + count(r, x)

count(Branch(Leaf(2), 3, Branch(Leaf(2), 2, Leaf(1))), 2) // 3 : Int
```

Pattern Matching

Here is another example of pattern matching on ADTs.

```
// An ADT for natural numbers
enum Nat:
  case Zero
  case Succ(n: Nat)

import Nat.* // Import all constructors for variants of `Nat`
```

Pattern Matching

Here is another example of pattern matching on ADTs.

```
// An ADT for natural numbers
enum Nat:
  case Zero
  case Succ(n: Nat)

import Nat.* // Import all constructors 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 Zero          => true
  case Succ(Succ(y)) => isEven(y)   // nested pattern matching
  case _             => false

isEven(Zero)                        // true  : Boolean
isEven(Succ(Zero))                  // false : Boolean
isEven(Succ(Succ(Zero)))            // true  : Boolean
```

Methods

You can define methods inside `case class` or enumerations (`enum`).

```scala
case class Point(x: Int, y: Int, color: String):
  // A method that returns a new point moved by (dx, dy)
  def move(dx: Int, dy: Int): Point = Point(x + dx, y + dy, color)

Point(3, 4, "RED").move(1, -2)     // Point(4, 2, "RED"): Point
```

Methods

You can define methods inside `case class` or enumerations (`enum`).

```
case class Point(x: Int, y: Int, color: String):
  // A method that returns a new point moved by (dx, dy)
  def move(dx: Int, dy: Int): Point = Point(x + dx, y + dy, color)

Point(3, 4, "RED").move(1, -2)     // Point(4, 2, "RED"): Point
```

The keyword `this` refers to the current instance.

```
enum Tree:
  ...
  // A recursive method counts the number of the given integer in a tree
  def count(x: Int): Int = this match
    case Leaf(n)          if n == x  => 1
    case Leaf(_)                     => 0
    case Branch(l, n, r) if n == x  => l.count(x) + 1 + r.count(x)
    case Branch(l, _, r)             => l.count(x) + r.count(x)

import Tree.* // Import all constructors for variants of `Nat`
Branch(Leaf(2), 3, Branch(Leaf(2), 2, Leaf(1))).count(2) // 3 : Int
```

Contents

First-Class Functions

parameter name — function body

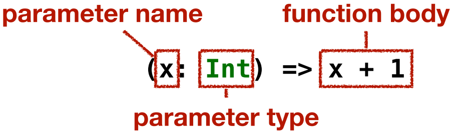
```
(x: Int) => x + 1
```

parameter type

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

```
// 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



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
```

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

```
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

PLRG

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**.

```
// 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).

```
// 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

PLRG

`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

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

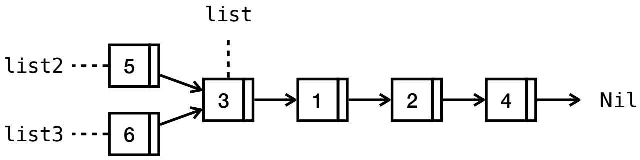
```
val list: List[Int] = List(3, 1, 2, 4)
```

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

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

Lists are immutable.

```
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

```
// 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 ++ List(5, 6, 7)          // List(3, 1, 2, 4, 5, 6, 7) : List[Int]
list.reverse                   // List(4, 2, 1, 3)          : List[Int]
list.count(_ % 2 == 1)         // 2                           : Int
list.foldLeft(0)(_ + _)        // 0 + 3 + 1 + 2 + 4 = 10    : Int
list.sorted                    // List(1, 2, 3, 4)          : List[Int]
list.map(_ * 2)                // List(6, 2, 4, 8)          : List[Int]
list.flatMap(x => List(x, -x)) // List(3, -3, ..., 4, -4)   : List[Int]
list.filter(_ % 2 == 1)        // List(3, 1)                : List[Int]

// 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.keySet       // Set("a", "b")                      : Set[String]
```

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]
set1.diff(set2)       // Set(1)          : Set[Int]
set1.subsetOf(set2)   // false           : Boolean
```

For Comprehensions

A **for comprehension**[1] is a syntactic sugar for nested `map`, `flatMap`, and `filter` operations:

```
val list = List(1, 2, 3)

// Using `map`, `flatMap`, and `filter`
list.flatMap(x => List(x, -x)) // List(1, -1, 2, -2, 3, -3)  : List[Int]
    .map(y => y * 3 + 1)       // List(4, -2, 7, -5, 10, -8) : List[Int]
    .filter(z => z % 5 == 0)   // List(-5, 10)               : List[Int]

// Using a for comprehension
for {
  x <- list
  y <- List(x, -x)
  z = y * 3 + 1
  if z % 5 == 0
} yield z                      // List(-5, 10)               : List[Int]
```

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

Exercise #1

- Please see this document for the exercise.

  `https://plrg.korea.ac.kr/courses/cose215/2024_1/ex/ex01.pdf`

- It is just an exercise, and it is **NOT** included in your grade.

- If you want to get feedback about your solution, you can send your answers to TA(`kimjg1119@korea.ac.kr`) by 23:59 on March 17.

Homework #1

- Please see this document on GitHub:

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

- The due date is 23:59 on Mar. 25 (Mon.).

- Please only submit `Implementation.scala` file to **Blackboard**.

Summary

1. Basic Features
   - Basic Data Types
   - Variables
   - Methods
   - Recursion
2. Algebraic Data Types (ADTs)
   - Product Types – Case Classes
   - Algebraic Data Types (ADTs) – Enumerations
   - Pattern Matching
   - Methods
3. First-Class Functions
4. Immutable Collections
   - Lists
   - Options and Pairs
   - Maps and Sets
   - For Comprehensions

Next Lecture

- Deterministic Finite Automata (DFA)

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