Lecture 2 – Basic Introduction of Scala

COSE215: Theory of Computation

Jihyeok Park

PLRG

2023 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

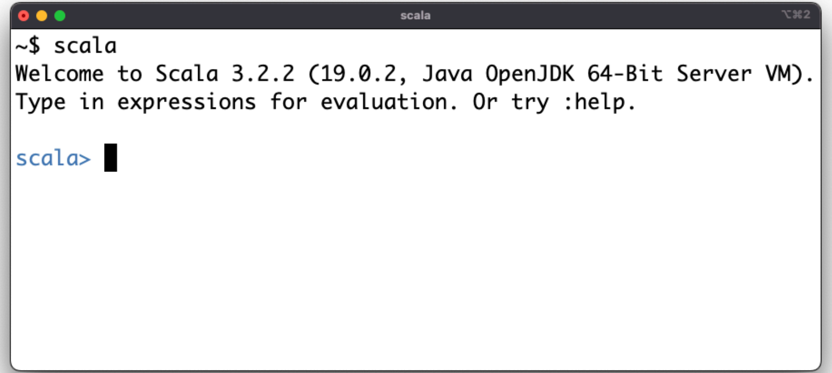
What is Scala?

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

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

Read Eval Print Loop (REPL)

- Please download Scala REPL:
  https://www.scala-lang.org/download/



```
~$ scala
Welcome to Scala 3.2.2 (19.0.2, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> 
```

Contents

1. Basic Features
   - Primitive Values
   - Immutable Variables
   - Functions
   - Conditional Branches
2. Object-Oriented Programming (OOP)
   - Case Classes
   - Traits
   - Pattern Matching
3. Functional Programming (FP)
   - Higher-Order Functions (Functions as Values)
   - Recursion
4. Immutable Collections (Data Structures)
   - Lists
   - Options and Pairs
   - Maps
   - Sets

Contents

1. Basic Features
    - Primitive Values
    - Immutable Variables
    - Functions
    - Conditional Branches
2. Object-Oriented Programming (OOP)
    - Case Classes
    - Traits
    - Pattern Matching
3. Functional Programming (FP)
    - Higher-Order Functions (Functions as Values)
    - Recursion
4. Immutable Collections (Data Structures)
    - Lists
    - Options and Pairs
    - Maps
    - Sets

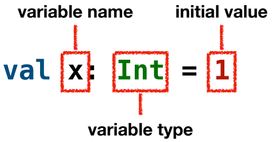
Primitive Values

```scala
// You can write comments using `// ...` or `/* ... */`
// Integers
1 + 2         // 3: Int
3 - 2         // 1: Int
2 * 3         // 6: Int

// Booleans
true && false // false: Boolean
true || false // true : Boolean
! true        // false: Boolean
1 == 2        // false: Boolean
1 < 2         // true : Boolean

// Characters (Symbols) and Strings (Words)
'a'               // 'a'          : Char
"abc"             // "abc"        : String
"hello" + "world" // "helloworld" : String
"hello".length    // 5            : Int
"hello"(0)        // 'h'          : Char
```

Immutable Variables



variable name

initial value

val x: Int = 1

variable type

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

// An immutable variable `s` of type `String` with "abc"
val s = "abc"

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

Immutable Variables

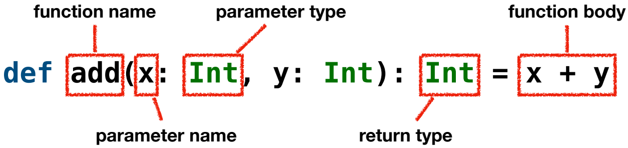
While Scala supports mutable variables (`var`), **DO NOT USE MUTABLE VARIABLES IN THIS COURSE**.

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

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

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

Functions



function name

parameter type

function body

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

parameter name

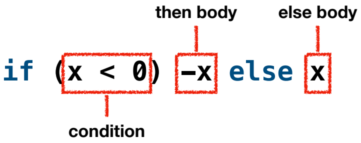
return type

```scala
// A function `add` of type `(Int, Int) => Int`
def add(x: Int, y: Int): Int = x + y
add(1, 2)           // 1 + 2 == 3
add(5, 6)           // 5 + 6 == 11

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

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

Conditional Branches



then body

else body

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

condition

```
// a function `abs` of type `Int => Int`
def abs(x: Int): Int = if (x < 0) -x else x
abs(-3)                // 3
abs(42)                // 42
```

Note that the conditional branch is an **expression**, not a **statement**.

Contents

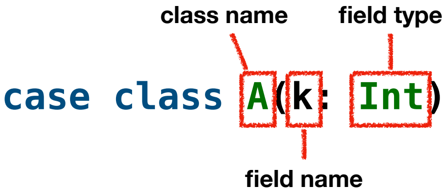
1. Basic Features
   - Primitive Values
   - Immutable Variables
   - Functions
   - Conditional Branches
2. Object-Oriented Programming (OOP)
   - Case Classes
   - Traits
   - Pattern Matching
3. Functional Programming (FP)
   - Higher-Order Functions (Functions as Values)
   - Recursion
4. Immutable Collections (Data Structures)
   - Lists
   - Options and Pairs
   - Maps
   - Sets

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm based on the concept of **"objects"**, which can contain data and code. The data is in the form of **fields** (often known as attributes or properties), and the code is in the form of **procedures** (often known as methods).[1]

[1] `https://en.wikipedia.org/wiki/Object-oriented_programming`

Case Classes



class name

field type

`case class A(k: Int)`

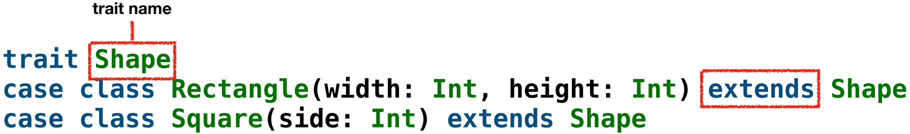
field name

```
// A case class `A` having a field `k` of type `Int`
case class A(k: Int)

// An instance object `a` of type `A` whose field `k` has 10
val a: A = A(10)

// You can access fields using the dot operator
a.k // 10
```

Traits



trait name

```scala
trait Shape
case class Rectangle(width: Int, height: Int) extends Shape
case class Square(side: Int) extends Shape
```

```scala
// A `Rectangle` type is a `Shape` type
val rectangle: Rectangle = Rectangle(20, 30)
rectangle.width    // 20
rectangle.height   // 30
val shape1: Shape = Rectangle(20, 30)
shape1.width       // `width` is not a field of `Shape`
shape1.height      // `height` is not a field of `Shape`

// A `Square` type is a `Shape` type
val square: Square = Square(10)
square.side        // 10
val shape2: Shape = Square(10)
shape2.side        // `side` is not a field of `Shape`
```

Pattern Matching

You can use **pattern matching** to match a value against a pattern.

```scala
def is42(n: Int): Boolean = n match
  case 42                => true    // exact matching for 42
  case 1 | 2 | 3         => false   // `|` denotes disjunction
  case k if k > 43       => false   // `if` denotes a guard
  case _                 => false   // `_` denotes a wildcard

is42(42)  // true
is42(1)   // false
is42(44)  // false
is42(10)  // false
```

```scala
def perimeter(sh: Shape): Int = sh match
  case Rectangle(w, h) => 2 * (w + h)
  case Square(s)       => 4 * s

perimeter(Rectangle(20, 30))  // 100
perimeter(Square(10))         // 40
```

Contents

1. Basic Features
   - Primitive Values
   - Immutable Variables
   - Functions
   - Conditional Branches
2. Object-Oriented Programming (OOP)
   - Case Classes
   - Traits
   - Pattern Matching
3. Functional Programming (FP)
   - Higher-Order Functions (Functions as Values)
   - Recursion
4. Immutable Collections (Data Structures)
   - Lists
   - Options and Pairs
   - Maps
   - Sets

Functional Programming (FP)

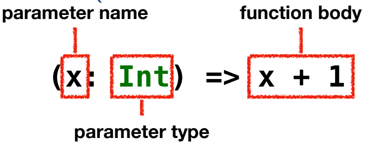
In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing **functions**. It is a **declarative programming paradigm** in which function definitions are trees of expressions that map values to other values, rather than a sequence of **imperative statements** which update the running state of the program.[2]

- If a function always returns the same result when given the same, it is a **pure function**. **PLEASE DEFINE ONLY PURE FUNCTIONS IN THIS COURSE**. How about the following function `f`?

```
var y: Int = 1
def f(x) = x + y
f(1)  // 1 + 1 = 2
y = 2
f(1)  // 1 + 2 = 3
```

[2] https://en.wikipedia.org/wiki/Functional_programming

Higher-Order Functions (Functions as Values)



parameter name

function body

(x: Int) => x + 1

parameter type

```
// An arrow function that increments its input
(x: Int) => x + 1      // Int => Int

// A function `inc` that increments its input
val inc: Int => Int = (x: Int) => x + 1
inc(1)  // 2

// A function `twice` that applies a function twice
def twice(f: Int => Int, x: Int): Int = f(f(x))
twice(inc, 5)                   // 7
twice((x: Int) => x + 1, 5)     // 7
twice(x => x + 1, 5)            // 7 - Type Inference
twice(_ + 1, 5)                 // 7 - Placeholder Syntax
```

Recursion

You can **recursively** invoke a function.

```scala
// Sum of all the numbers from 1 to n
def sum(n: Int): Int = n match
  case 0 => 0
  case k => k + sum(k - 1)
sum(10)  // 55
sum(100) // 5050
```

```scala
// A tree is either a branch or a leaf
trait Tree
case class Branch(l: Tree, n: Int, r: Tree) extends Tree
case class Leaf(n: Int) extends Tree
// Sum of all the values in a tree
def sum(t: Tree): Int = t match
  case Branch(l, n, r) => sum(l) + n + sum(r)
  case Leaf(n)         => n
sum(Branch(Leaf(1), 2, Leaf(3)))                       // 6
sum(Branch(Branch(Leaf(1), 2, Leaf(3)), 4, Leaf(5)))  // 15
```

Recursion

While Scala supports `while` loops, **PLEASE DO NOT USE WHILE LOOPS IN THIS COURSE**.

```scala
// 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) do
    s = s + k
    k = k + 1
  s
sum(10)  // 55
sum(100) // 5050
```

Contents

1. Basic Features
   - Primitive Values
   - Immutable Variables
   - Functions
   - Conditional Branches
2. Object-Oriented Programming (OOP)
   - Case Classes
   - Traits
   - Pattern Matching
3. Functional Programming (FP)
   - Higher-Order Functions (Functions as Values)
   - Recursion
4. Immutable Collections (Data Structures)
   - Lists
   - Options and Pairs
   - Maps
   - Sets

Lists

A **list** is a sequence of elements of the same type.

```scala
// A list of integers: 3, 1, 2, 5, 4
val list: List[Int] = List(3, 1, 2, 5, 4)
val list2 = 3 :: 1 :: 2 :: 5 :: 4 :: Nil
list == list2              // true
// Pattern matching on lists
def countOdd(list: List[Int]): Int = list match
  case Nil                   => 0
  case x :: xs if x % 2 == 1 => 1 + countOdd(xs)
  case _ :: xs               => countOdd(xs)
countOdd(list)             // 3 (three odd numbers: 3, 1, 5)
// Operations/functions on lists
6 :: list                  // List(6, 3, 1, 2, 5, 4)
list ++ List(6, 7, 8)      // List(3, 1, 2, 5, 4, 6, 7, 8)
list.reverse               // List(4, 5, 2, 1, 3)
list.filter(_ % 2 == 1)    // List(3, 1, 5)
list.map(_ * 2)            // List(6, 2, 4, 10, 8)
list.foldLeft(0)(_ + _)    // 15
list.sorted                // List(1, 2, 3, 4, 5)
```

Options and Pairs

An **option** is a container that may or may not contain a value. **DO NOT USE NULL IN THIS COURSE.**

```scala
val some: Option[Int] = Some(42)
val none: Option[Int] = None
// Operations/functions on options
some.map(_ + 1)      // Some(43)
none.map(_ + 1)      // None
some.getOrElse(7)    // 42
none.getOrElse(7)    // 7
some.fold(3)(_ * 2) // 84
none.fold(3)(_ * 2) // 3
```

A **pair** is a container that contains two values.

```scala
val pair: (Int, String) = (42, "foo")
// Operations/functions on options
pair(0)                // 42    - NOT RECOMMENDED
pair(1)                // "foo" - NOT RECOMMENDED
val (x, y) = pair      // x = 42, y = "foo"
```

Maps

A **map** is a mapping from keys to values.

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

// Operations/functions on maps
map + ("c" -> 3)        // Map("a" -> 1, "b" -> 2, "c" -> 3)
map + ("a" -> 3)        // Map("a" -> 3, "b" -> 2)
map - "a"               // Map("b" -> 2)
map.get("a")            // Some(1)
map.get("c")            // None
map.getOrElse("a", 42)  // 1
map.getOrElse("c", 42)  // 42
map.toList              // List(("a", 1), ("b", 2))
map.keySet              // Set("a", "b")
map.values.toList       // List(1, 2)
```

Sets

A **set** is a collection of distinct elements.

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

// Operations/functions on sets
set1 + 4               // Set(1, 2, 3, 4)
set1 + 2               // Set(1, 2, 3)
set1 - 2               // Set(1, 3)
set1.contains(2)       // true
set1 ++ set2           // Set(1, 2, 3, 5)
set1.intersect(set2)   // Set(2, 3)
set1.diff(set2)        // Set(1)
set1.subsetOf(set2)    // false
set1.toList            // List(1, 2, 3)
```

Homework #1

- Please see https://github.com/ku-plrg-classroom/docs/tree/main/scala-tutorial.
- The due date is Mar. 21 (Tue.).
- Please only submit `Implementation.scala` file to **Blackboard**.

Summary

1. Basic Features
   - Primitive Values
   - Immutable Variables
   - Functions
   - Conditional Branches
2. Object-Oriented Programming (OOP)
   - Case Classes
   - Traits
   - Pattern Matching
3. Functional Programming (FP)
   - Higher-Order Functions (Functions as Values)
   - Recursion
4. Immutable Collections (Data Structures)
   - Lists
   - Options and Pairs
   - Maps
   - Sets

Next Lecture

- Deterministic Finite Automata (DFA)

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