Lecture 1 – Basic Introduction of Scala

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

The goal of this course:

Learn **Essential Concepts** of **Programming Languages**

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

Recall

The goal of this course:

Learn **Essential Concepts** of **Programming Languages**

- How?

By Implementing **Interpreters** using **Scala**

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

Recall

The goal of this course:

Learn **Essential Concepts** of **Programming Languages**

- How?

  By Implementing **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.

What is Scala?

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

- A **more concise** version of Java with **advanced features**

What is Scala?

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

- A **more concise** version of Java with **advanced features**
- A general-purpose programming language

What is Scala?

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

- A **more concise** version of Java with **advanced features**
- A general-purpose programming language
- **Java Virtual Machine (JVM)**-based language

What is Scala?

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

- A **more concise** version of Java with **advanced features**
- A general-purpose programming language
- **Java Virtual Machine (JVM)**-based language
- A **statically typed** language

What is Scala?

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

- A **more concise** version of Java with **advanced features**
- A general-purpose programming language
- **Java Virtual Machine (JVM)**-based language
- A **statically typed** language
- A **object-oriented programming (OOP)** language

What is Scala?

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

- A **more concise** version of Java with **advanced features**
- 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 3.3.0 (20.0.2, Java Java HotSpot(TM) 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala>
```

Contents

Contents

Built-in Data Types

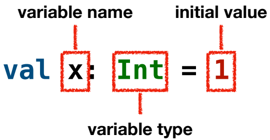
PLRG

```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

// Strings
"abc"                // "abc"        : String
"hello" + " world"   // "hello world": String
"hello".length       // 5            : Int
```

Immutable Variables (Identifiers)




```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`

// An immutable variables of other types
val b: Boolean = true
val s: String = "abc"

// 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

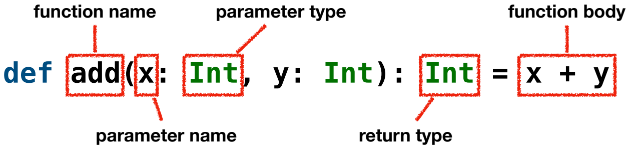
While Scala supports mutable variables (`var`), **DO NOT USE MUTABLE VARIABLES IN THIS COURSE**.

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

```scala
// 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
```

Functions

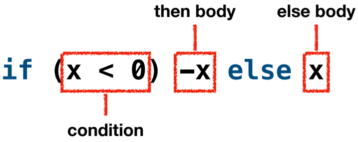



```scala
// A function `add` of type `(Int, Int) => Int`
def add(x: Int, y: Int): Int = x + y
add(1, 2)          // 1 + 2 == 3  : Int
add(5, 6)          // 5 + 6 == 11 : Int

// 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




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

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

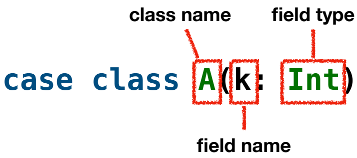
Contents

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm based on the concept of **object**, 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).[2]

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

Case Classes




```
// 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 : Int

// Fields are immutable by default
a.k = 20    // Type Error: Reassignment to val `k`
```

Contents

Algebraic Data Types (ADTs)

An **algebraic data type (ADT)** is a kind of composite type, i.e., a type formed by combining other types.

type name

variants

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

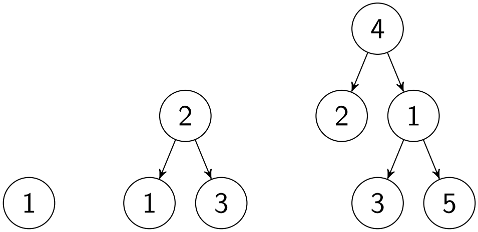
Algebraic Data Types (ADTs)

An **algebraic data type (ADT)** is a kind of composite type, i.e., a type formed by combining other types.

type name — variants

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

```
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 use **pattern matching** to match a value against a pattern.

```
def getValue(t: Tree): Int = t match
  case Leaf(v)            => v
  case Branch(_, v, _)    => v

getValue(tree1)  // 1 : Int
getValue(tree2)  // 2 : Int
getValue(tree3)  // 4 : Int
```

Pattern Matching

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

```scala
def getValue(t: Tree): Int = t match
  case Leaf(v)            => v
  case Branch(_, v, _)    => v

getValue(tree1)  // 1 : Int
getValue(tree2)  // 2 : Int
getValue(tree3)  // 4 : Int
```

```scala
enum Number:
  case Zero
  case Succ(n: Number)

def toInt(n: Number): Int = n match
  case Zero    => 0
  case Succ(n) => 1 + toInt(n)

toInt(Zero)             // 0 : Int
toInt(Succ(Succ(Zero))) // 2 : Int
```

Contents

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.[3]

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

[4] https://docs.scala-lang.org/scala3/book/fp-pure-functions.html

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.[3]

- A **pure function**[4] is a function that 1) returns the **same result** for the same input and 2) has **no side effects**.

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

[4] https://docs.scala-lang.org/scala3/book/fp-pure-functions.html

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.[3]

- A **pure function**[4] is a function that 1) returns the **same result** for the same input and 2) has **no side effects**.
- **Immutability** is a cornerstone of pure functions:

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

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

[4] https://docs.scala-lang.org/scala3/book/fp-pure-functions.html

First-Class Functions (Functions as Values)

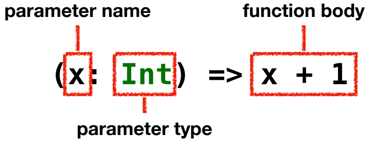
```scala
// A function `inc` that increments its input
def inc(x: Int): Int = x + 1
inc(3)                          // 3 + 1 = 4 : Int

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

First-Class Functions (Functions as Values)

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

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




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

Recursion

You can **recursively** invoke a function.

```
// Sum of all the numbers from 1 to n
def sum(n: Int): Int = if (n < 1) 0 else sum(n - 1) + n

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

Recursion

You can **recursively** invoke a function.

```
// Sum of all the numbers from 1 to n
def sum(n: Int): Int = if (n < 1) 0 else sum(n - 1) + n

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

```
// An ADT for trees
enum Tree:
  case Leaf(value: Int)
  case Branch(left: Tree, value: Int, right: Tree)
// Import all constructors for variants of `Tree`
import Tree.*
// A function recursively 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
```

Recursion

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

```
// 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

Lists

A **list** (`List[T]`) is a sequence of elements of the same type `T`:

1. `Nil` represents the empty list
2. `::` adds an element to the front of a list

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

// You can construct lists using `::` and `Nil`
3 :: 1 :: 2 :: 4 :: Nil == list  // true : Boolean

// Pattern matching on lists - filter odd integers and double them
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

PLRG

```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 ++ 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**[5] 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]
```

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

Homework #1

- Please see this document[6] on GitHub.
- The due date is Sep. 20 (Wed.).
- Please only submit `Implementation.scala` file to **Blackboard**.

[6] https://github.com/ku-plrg-classroom/docs/tree/main/scala-tutorial.

Summary

1. Basic Features
   - Built-in Data Types
   - Variables
   - Functions
   - Conditionals
2. Object-Oriented Programming (OOP)
   - Case Classes
3. Algebraic Data Types (ADTs)
   - Pattern Matching
4. Functional Programming (FP)
   - First-Class Functions
   - Recursion
5. Immutable Collections (Data Structures)
   - 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