Lecture 1 – Basics

SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

Recall

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

Functional Programming

We will use **functional programming** (FP) for secure programming 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, Either, and Try for error handling

Contents

Contents

Basic Data Types – Integers and Doubles

PLRG

`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)
```

Basic Data Types – 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 type represents a **unit value** `()` (similar to `void` in Java).

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

Strings

`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

variable name **initial value**

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 b: 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.

```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
```

String Interpolation

Scala supports **string interpolation** using `s` prefix.

```
val firstName = "Jihyeok"
val lastName = "Park"
s"Name: ($firstName / $lastName)"      // "Name: (Jihyeok / Park)"
```

String Interpolation

Scala supports **string interpolation** using `s` prefix.

```
val firstName = "Jihyeok"
val lastName = "Park"
s"Name: ($firstName / $lastName)"      // "Name: (Jihyeok / Park)"
```

You can use `${...}` for more complex expressions.

```
val x = 1
val y = 2
s"$x + $y = ${x + y}"                  // "1 + 2 = 3"
```

If you want to use `printf`-style formatting, you can use `f` prefix.

```
val pi = 3.14159
f"3 * PI: ${3 * pi}%.2f"               // "3 * PI: 9.42"
```

Contents

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`
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 parameter values
add(1)            // Too few parameter values
add(1, 2, 3)      // Too many parameter values

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

Default and Named Parameters

We can assign **default parameter values** to a method.

```scala
def foo(x: String, y: Int = 42, z: Boolean = true): String =
  s"x = $x | y = $y | z = $z"

foo("abc")                          // "x = abc | y = 42 | z = true"
```

Default and Named Parameters

We can assign **default parameter values** to a method.

```
def foo(x: String, y: Int = 42, z: Boolean = true): String =
  s"x = $x | y = $y | z = $z"

foo("abc")                          // "x = abc | y = 42 | z = true"
```

We can use **named parameters** to specify the parameter names.

```
foo(x = "abc", y = 7, z = false)  // "x = abc | y = 7 | z = false"
```

Default and Named Parameters

We can assign **default parameter values** to a method.

```
def foo(x: String, y: Int = 42, z: Boolean = true): String =
  s"x = $x | y = $y | z = $z"

foo("abc")                          // "x = abc | y = 42 | z = true"
```

We can use **named parameters** to specify the parameter names.

```
foo(x = "abc", y = 7, z = false)  // "x = abc | y = 7 | z = false"
```

We can freely change the order of the named parameters.

```
foo(z = false, x = "abc", y = 7)  // "x = abc | y = 7 | z = false"
```

Default and Named Parameters

We can assign **default parameter values** to a method.

```
def foo(x: String, y: Int = 42, z: Boolean = true): String =
  s"x = $x | y = $y | z = $z"

foo("abc")                          // "x = abc | y = 42 | z = true"
```

We can use **named parameters** to specify the parameter names.

```
foo(x = "abc", y = 7, z = false)  // "x = abc | y = 7 | z = false"
```

We can freely change the order of the named parameters.

```
foo(z = false, x = "abc", y = 7)  // "x = abc | y = 7 | z = false"
```

We can skip the default parameters when using named parameters.

```
foo("abc", z = false)           // "x = abc | y = 42 | z = false"
```

Recursions

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

Recursions

```scala
// 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
sum(100)      // 5050     : Int
```

$$\texttt{if } (\overbrace{\boxed{\texttt{x < 0}}}^{\text{condition}})\ \overbrace{\boxed{\texttt{-x}}}^{\text{then body}}\ \texttt{else}\ \overbrace{\boxed{\texttt{x}}}^{\text{else body}}$$

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

```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
```

Recursions

```scala
// 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
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.

```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 it is a conditional **expression** not a **statement**.

Recursions

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

```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) {
    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**

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

field name

```
// 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

PLRG

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

type name

variants

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

Algebraic Data Types (ADTs) – Enumerations

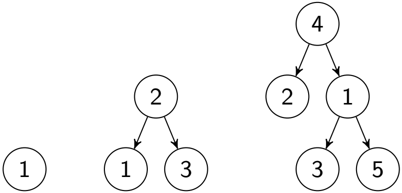
An **algebraic data type (ADT)** is a sum of product types, and you can define it using **enumerations** (`enum`) in Scala.

type name

variants

```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).

```
// 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
```

Pattern Matching

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

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

// A function converts a natural number to an integer
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
```

Contents

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
```

Functions

parameter name　　function body

```scala
(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 incF1: Int => Int = (x: Int) => x + 1
incF1(3)                                   // 3 + 1 = 4 : Int
val incF2: Int => Int = x => x + 1         // Type Inference: `x` is `Int`
incF2(3)                                   // 3 + 1 = 4 : Int
val incF3: Int => Int = _ + 1              // Placeholder Syntax
incF3(3)                                   // 3 + 1 = 4 : Int
```

Functions

We can **pass** a function to a function as an argument.

```
// 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

// 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
```

Functions

We can **pass** a function to a function as an argument.

```
// 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

// 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 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`.

```
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`.

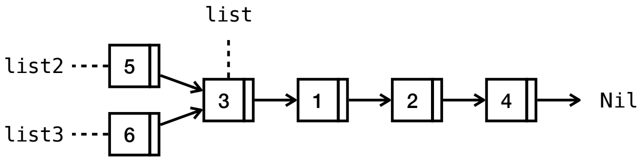
```
val list: List[Int] = List(3, 1, 2, 4)
```

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

// 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

```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]
```

Summary

1. Basic Data Types and Variables
   - Basic Data Types
   - Variables
   - String Interpolation
2. Methods
   - Default and Named Parameters
   - Recursions
3. Algebraic Data Types (ADTs)
   - Product Types – Case Classes
   - Algebraic Data Types (ADTs) – Enumerations
   - Pattern Matching
4. Functions
5. Lists

Next Lecture

- Testing and Documentation

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