Introduction to Scala

Jimin Hsieh
https://gitter.im/jiminhsieh
https://twitter.com/JiminHsieh

https://gitter.im/ScalaTaiwan/ScalaTaiwan

Me

❖ Talk - <u>Scala & Spark in Performance Aspect</u> @ Scala Taiwan

❖ Workshop - <u>Introduction to Scala</u> @ JCConf Taiwan

❖ <u>Functional Programming in Scala</u> Specialization @ Coursera
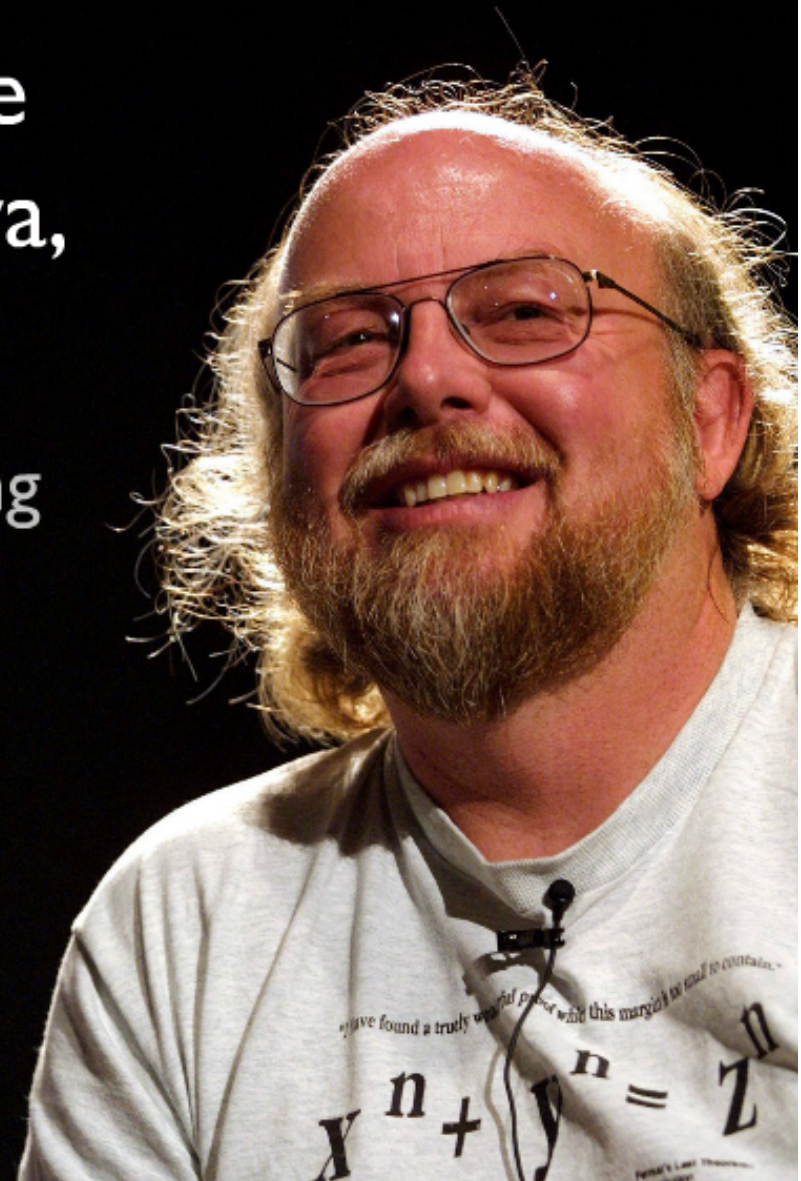
❖ <u>Contribute to some of Scala open sources</u>.

Agenda

Why Scala?

"If I were to pick a language to use today other than Java, it would be Scala"

James Gosling

Why Scala?

"I can honestly say **if someone had shown me the Programming in Scala book** by Martin Odersky, Lex Spoon & Bill Venners back in 2003 **I'd probably have never created Groovy.**"

James Strachan, creator of Groovy

http://macstrac.blogspot.tw/2009/04/scala-as-long-term-replacement-for.html

Why Scala?

A language that doesn't affect the way you think about programming is not worth knowing.

— Alan Perlis —

AZ QUOTES

http://www.azquotes.com/quote/765126

Why Scala?

"A programming language is for thinking about programs, not for expressing programs you've already thought of. It should be a pencil, not a pen."

CHANGEMAKRS.COM / PAULGRAHAM

http://thenextweb.com/shareables/2012/08/08/cant-get-enough-of-those-epic-quotes-from-y-combinators-paul-graham-youre-in-luck/
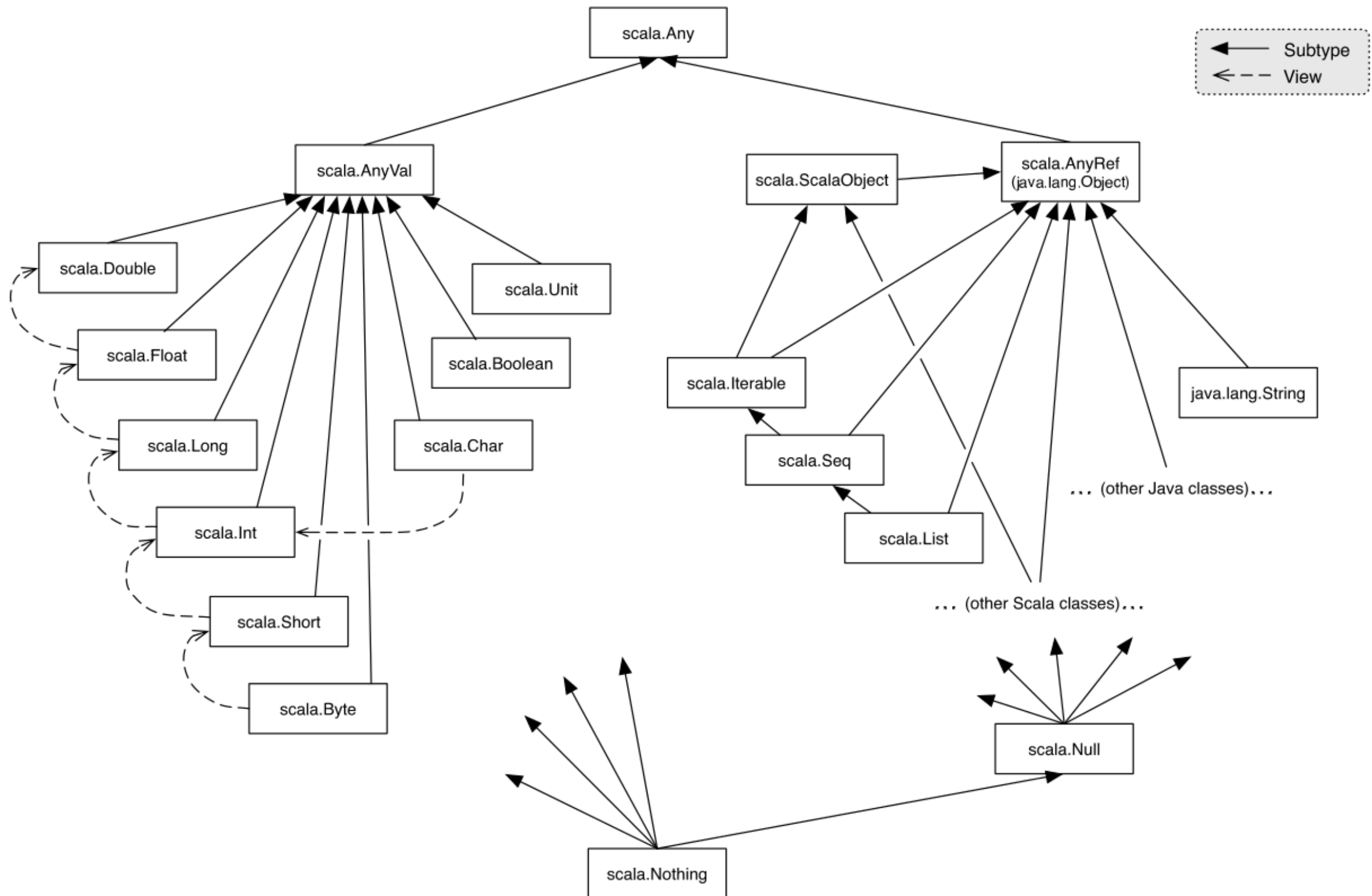
Why Scala?

High productive with **type safety**, more APIs than Java, and almost same performance with Java

Scala concept

- Everything is an <u>object</u>.

  - Not everything is an object in Java.

    - There is primitive type in Java.

      - int, float, boolean, char…etc

  - Scala is an object-oriented language in **pure** form: **every value** is an object and **every operation** is a method call.

    - Numeric types and Function are object too.

    - "**+**", "**-**", "*****", "**/**" are methods too.

Scala Class Hierarchy

Scala concept

❖ **Everything is expression.**

    ❖ Expression - an instruction to execute something that will return a value. from Wiki

    ❖ You can also say that an expression *evaluates* to a result or results in a value.

    ❖ You will hear evaluation from some of Scala geeks, it means the same thing.

Scala concept

❖ Scala is immutable by default.

   ❖ What does immutable mean?

      ❖ You **cannot reassign** object to a value or field.

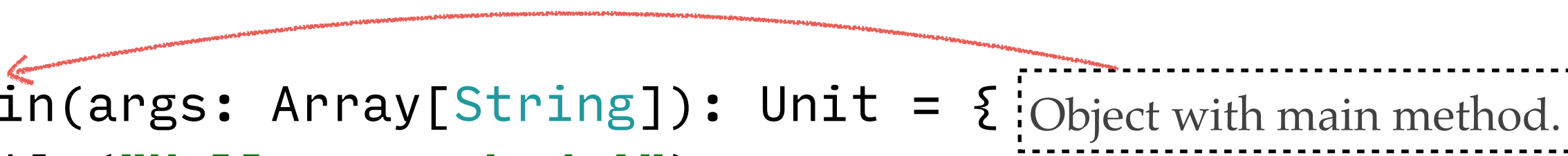   ❖ You **cannot modify the content** which is inside object or data structures.

Scala concept

❖ **Advanced type system**

   ❖ static

   ❖ strong

   ❖ inferred

   ❖ structural

Scala concept

❖ Avoid to use null.

   ❖ Less error prone.

      ❖ NullPointerException

   ❖ You don't need to use Null Object pattern.

Program with Scala (Main)

```scala
object ScalaMain {
  val todayEvent = "Workshop"
  val workshop = "Introduction to Scala"
  lazy val funStream = Stream.from(1).map(_ => "fun")

  def main(args: Array[String]): Unit = {   Object with main method.
    println("Hello everybody!")
    print("Welcome to " + todayEvent + "!\n")
    println("I hope you can enjoy – "
      + workshop + ". :)")
    print("Scala is so much " +
      funStream.take(5).mkString(" ") + "!")
  }
}
```

ScalaMain.scala

Program with Scala (App)

```scala
object ScalaApp extends App {
  val todayEvent = "Workshop"
  val workshop = "Introduction to Scala"
  lazy val fun = (0 to 4).map(_ => "fun").mkString(" ")

  println("Hello everybody!")
  print("Welcome to " + todayEvent + "!\n")
  println("I hope you can enjoy this workshop - "
    + workshop + ". :)")
  print("Scala is so much " + fun + "!")
}
```

Object with App trait.

ScalaApp.scala

Program with Scala (REPL)

❖ REPL - Read-Evaluate-Print Loop

```
Welcome to Scala 2.12.4 (OpenJDK 64-Bit Server VM, Java 1.8.0_152).
Type in expressions for evaluation. Or try :help.

scala>
```

Program with Scala (Worksheet)

❖ Work with worksheet.

   ❖ **IntelliJ**

      ❖ https://www.youtube.com/watch?v=ZyV54GSAqkI

   ❖ Scala IDE or Eclipse with Scala plugin

      ❖ https://www.youtube.com/watch?v=Forl4hpg7kA

Imperative

- ❖ var, val, semicolons

- ❖ If expressions

- ❖ def

- ❖ Block expressions

- ❖ While-Loop

- ❖ For-Loop

- ❖ Nested Function

- ❖ Recursion vs Tail-recursion

- ❖ Concept of Pattern Matching

- ❖ Pattern Matching v1

var vs. val

- var - variable
  - Something that able or likely to change or be changed. **Not always the same**. Merriam-Webster

- val - value
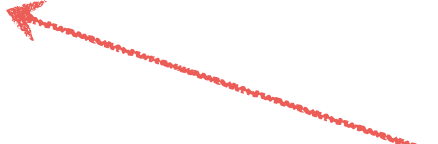 - A value is an expression which cannot be evaluated any further. Wiki
  - Opposite to var, val cannot be changed.
  - It's similar to **final** in Java.

Expression with semicolon?

```
val x = 5566
val y = 87
val java = "Java"; val scala = "scala"
```

If you have **multiple expressions** in one line,
you will need semicolon(;).
Otherwise you don't need it.

If Expressions

❖ If-expression has return value.

   ❖ Scala have no ternary operator(?:).

```
// Java version
final int value = -1;
final boolean negative = value < 0 ? true : false;


// Scala version
val value = 0
val negative = if (value < 0) true else false
```

Everything is an expression.

def

"def" starts a function definition

result type of function

function name

parameter

equals sign

```scala
def max(x: Int, y: Int): Int = {
  if (x > y)
    x
  else
    y
}
```

function body in curly braces

Programming in Scala, 3ed by Martin Odersky, Lex Spoon, and Bill Venners

Def_Expression.sc

def

```scala
def max(x: Int, y: Int): Int = {
  if (x > y)
    return x
  else
    return y
}
```

def

```scala
def max(x: Int, y: Int) = {
  if (x > y)
    x
  else
    y
}
```

No function's result type

def

```scala
def max(x: Int, y: Int) =
  if (x > y)
    x
  else
    y
```
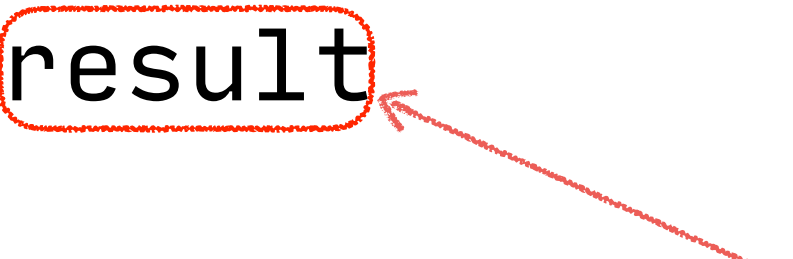
No curly brackets

Summary of def

❖ You don't need return.

   ❖ **Last expression** of block will be the return value.

❖ You don't need return type in method definition.

   ❖ Scalac will **know your return type** in most case.

   ❖ **It's a good habit to have return type, when your API is a public interface.**

❖ You don't need curly bracket.

   ❖ **If you have multiple lines of code, using curly bracket({}) is a good habit.**

Block expressions (curly brackets)

```
val n = 5
val factorial = {
  var result = 1
  for (i <- 1 to n)
    result = result * i
  result
}
```

Last expression(**result**) in code block will be the return value,
then it will assign to factorial.

While-Loop

```
var n = 10
var sum = 0

while (n > 0) {
  sum = sum + 1
  n = n - 1
}
```

For-Loop

```scala
var sum = 0
for (i <- 1 to 10) {
  sum += 1
}
println(sum)
```

For-Loop

```
for (i <- 0 until 10) {
  println(i)
}
```

For-Loop

```scala
for (_ <- 0 to n; j <- 0 to n) {
  print("*")
  if (j == n)
    println("")
}
```

Exercise of For-Loop

❖ Print out something like below.

Nested Function

```scala
def min(x: Int, y: Int): Int = {
  def max(x: Int, y: Int) = {
    if (x > y) x
    else y
  }

  if (x == max(x, y))
    y
  else
    x
}
```

Nested Function(Closure)

```
def min(x: Int, y: Int): Int = {
  def max = {
    if (x > y) x
    else y
  }

  if (x == max) y else x
}
```

If your method is zero parameters, you don't need parentheses.

Closure.sc

Recursion vs Tail-recursion

* Factorial number

  * 6! = 6 * 5 * 4 * 3 * 2 * 1

Recursion vs Tail-recursion

```scala
// Recursion
def factorial(n: Int): BigInt = {
  if (n == 0) 1
  else n * factorial(n - 1)
}


factorial(15000)
```

java.lang.StackOverflowError

Recursion vs Tail-recursion

```scala
def factorial(n: Int): BigInt = {
  // Tail-recursion
  def helpFunction(acc: BigInt, n: Int): BigInt = {
    if (n == 0)
      acc
    else
      helpFunction(acc * n, n - 1)
  }
  helpFunction(1, n)
}

factorial(15000)
```

Tail-recursion

Scalac will translate tail-recursion to **while-loop**.

Recursion vs Tail-recursion

Add annotation is a good habit.
Compiler can check whether or not
can be optimise.

```scala
import scala.annotation.tailrec
def factorial(n: Int): BigInt = {
  @tailrec
  def helpFunction(acc: BigInt, n: Int): BigInt = {
    if (n == 0)
      acc
    else
      helpFunction(acc * n, n - 1)
  }
  helpFunction(1, n)
}
```

You have to add return type,
when the function is recursive.
Or Scalac would not know the return type.

Tail_Recursion_With_Annotation.sc

Concept of Pattern Matching

❖ Pattern matching is similar to "**switch-case**", but it's **more general**.

❖ There are some differences.

  ❖ No fall-through.

  ❖ Each condition **needs to return a value**.

    ❖ Everything is a expression in Scala.

  ❖ It can match anything.

Pattern Matching v1

❖ Scala's pattern matching can match "Byte", "Character", "String", "Short" and "Integer". This is what Java's switch can do.

❖ But Scala can match **more** things than that. I will talk about this later.

Pattern Matching v1

```scala
def matchString(x: String) = {
  x match {
    case "Dog" => x
    case "Cat" => x
    case _ => "Neither Dog or Cat"
  }
}


matchString("Dog")
matchString("Human")
```

Pattern_Matching_v1.SC

Exercise of tail-recursion

❖ Fibonacci numbers (0, 1, 1, 2, 3, 5…)

$$F_n = F_{n-1} + F_{n-2},$$

with seed values

$$F_0 = 0 \quad \text{and} \quad F_1 = 1.$$

www.leaningtowerofpisa.net/fibonacci.html

Object-Oriented

❖ Class

    ❖ <u>Appendix - class and object</u>

❖ Extends, With, Override

❖ Abstract Class

❖ Object, Companion Object

❖ Trait

❖ Access Modifiers

❖ Case class

❖ Pattern matching v2

Class

Primary Constructor

without val or var,
you won't have setter and getter

val in constructor
will give you getter

var in constructor will
give you setter and getter

```scala
class Employee(id: Int,
               val name: String,
               address: String,
               var salary: BigDecimal) {
  def this(id: Int, name: String, salary: BigDecimal) {
    this(id, name, "Earth", salary)
  }

  def getAddress = address
}
```

Auxiliary Constructor;
In Java, it calls constructor overloading

When your method without parameters,
you don't need parentheses.

Class.sc

Extends, with, override

❖ Scala is single inheritance like Java.

   ❖ Scala - *extends* = Java - **extends**

   ❖ Scala - *with* = Java - *implements*

   ❖ Scala - *override* = Java - *@Override*

Abstract class

❖ Abstract class just likes normal class, but it can have **abstract methods** and **abstract fields** which means **methods and fields without implementation**.

❖ In Scala, **you don't need the keyword - ~~abstract~~** for methods and fields in Abstract class. When you leave **methods and fields without body**, it means abstract methods and fields.

Abstract class

```scala
sealed abstract class Animal(val name: String) {
  val footNumber: Integer
  val fly: Boolean
  def speak: String
}


class Dog(name: String) extends Animal(name) {
  override val footNumber: Integer = 4
  override val fly = false
  override def speak = "Spark"
}


class Bird(name: String) extends Animal(name) {
  val footNumber: Integer = 2
  val fly = true
  def speak = "chatter"
}
```

Sealed classes
1. Subclasses should be
   in the same file.
2. Compiler can check
   you have covered all of cases
   in pattern matching.

Abstract_Class.sc

Object

- Object is a <u>singleton</u>.

- It likes **normal class**.

  - You have **only one instance** in your application's life cycle.

  - Singleton objects **cannot** take **class parameters**.

- Scala **does not have static class members** like Java but has object instead.

- You don't need to worry about how to write a thread-safe singleton.

- What's advantage of Object?

  - <u>http://programmers.stackexchange.com/a/179402</u>

Object

```scala
object MathUtil {
  def doubleHalfUp(precision: Int,
                   origin: Double): Double = {
    val tmp = Math.pow(10, precision)
    Math.round(origin * tmp) / tmp
  }
}
```

Object.sc

Companion Object

❖ It's like normal object, but it **shares the same name with class name** and locates with class **in the same source file**.

❖ The class and its companion object can **access each other's private methods or fields**.

  ❖ Companion class have to import companion object explicit.

Companion Object

```scala
class CompanionObject {

    import CompanionObject._


    name


    private val address = "7788"
}


object CompanionObject {
    private val name = "5566"
}
```

Traits

❖ Traits are like interfaces in Java.

  ❖ You cannot pass class parameters.

❖ But there are differences.

  ❖ It can **contain** **fields** and **concrete methods**.

  ❖ Traits can **enrich a thin interface**(**mixin**) which can make a rich interface.

Trait - Mixin

```scala
trait Iterator[A] {
  def hasNext: Boolean
  def next: A
}

trait RichIterator[A]
  extends Iterator[A] {
  def foreach(f: A => Unit):
  Unit =
    while (hasNext)
      f(next)
}
```

```scala
class CharIterator(s: String)
  extends Iterator[Char] {
  var i = 0            → Cursor
  def hasNext = i < s.length
  def next = {
    val c = s(i)
    i += 1
    c              return
  }
}

val ci = new CharIterator("hello")
  with RichIterator[Char]
ci.foreach(println)
```

https://scalafiddle.io/sf/i3KWOvf/0 by Walter Chang

Trait.sc

Access Modifiers

* public

  * anyone

  * There is **no explicit modifiers** for public fields and methods.

* protected

  * only allow subclass access

* private

  * In the same scope of Scala file.

* protected[X] or private[X]

  * access is private or protected "up to" X

Case Classes

❖ Case classes are a special kind of classes that are **optimized for use in pattern matching**.

❖ Each of the constructor parameters becomes a **val** (immutable).

Scala for the Impatient by Cay Horstmann

Case Classes

- ❖ It creates the **companion object**.

  - ❖ It contains an **apply** method used to construct instance without **"new"**. (<u>Factory pattern</u>)

  - ❖ It contain an **unapply** method work as extractor to extract values from apply method.

  - ❖ Each instance contains toString, **equals**(==), hashCode, and copy.

<u>Scala for the Impatient by Cay Horstmann</u>

Pattern Matching v2

```scala
sealed abstract class Expr
case class Add(first: Expr, second: Expr) extends Expr
case class Sub(first: Expr, second: Expr) extends Expr
case class Multi(first: Expr, second: Expr) extends Expr
case class Div(first: Expr, second: Expr) extends Expr
case class Value(n: Int) extends Expr

def calculate(combination: Expr): Int = {
  combination match {
    case Add(first, second) => calculate(first) + calculate(second)
    case Sub(first, second) => calculate(first) - calculate(second)
    case Multi(first, second) => calculate(first) * calculate(second)
    case Div(first, second) => calculate(first) / calculate(second)
    case Value(n) => n
  }
}
// (((2 + 1) * 3) - 1) / 4
val exp = Div(Sub(Multi(
  Add(Value(1), Value(2)), Value(3)), Value(1)), Value(4))
calculate(exp)
```

Pattern Matching v2

```scala
def calculate(combination: Expr): Int = {
  combination match {
    case Add(Multi(a, x), Multi(b, y)) if (x == y) =>
      // (3 + 4) * 2
      (calculate(a) + calculate(b)) * calculate(x)
    case Add(first, second) =>
      calculate(first) + calculate(second)
    case Sub(first, second) =>
      calculate(first) - calculate(second)
    case Multi(first, second) =>
      calculate(first) * calculate(second)
    case Div(first, second) =>
      calculate(first) / calculate(second)
    case Value(n) =>
      n
  }
}
// ((3 * 2) + (4 * 2))
calculate(Add(Multi(Value(3), Value(2)), Multi(Value(4), Value(2))))
```

Pattern_Matching_v2_2.sc

Summary of Object-Oriented

❖ If it might be reused in multiple, unrelated classes, make it a trait.

❖ If you plan to distribute it in compiled form and you expect outside groups to write classes inheriting from it, you might lean towards using an abstract class.

❖ ....

❖ If you are not so sure abstract class or trait, then use trait. Trait keeps more options open.

Programming in Scala, 3ed by Martin Odersky, Lex Spoon, and Bill Venners

Functional

- ❖ Functional concept
- ❖ First-class functions
- ❖ Type of Function
- ❖ Anonymous functions
- ❖ High-order functions

Functional Concept

- ❖ Immutability (Referential transparency)

  - ❖ **No side effect**.

- ❖ Functions as values

  - ❖ **Functions as objects**

- ❖ Higher-order functions

  - ❖ Input: takes one or more **functions** as parameters

  - ❖ Output: return a **function** as result

First-class functions

❖ Functions are **first-class** values.

❖ Function is a value of **the same status as an integer or string**.

Programming in Scala, 3ed by Martin Odersky, Lex Spoon, and Bill Venners

Type of Function

- ❖ Int => String

- ❖ T => Unit

  - ❖ It's similar to return void.

- ❖ => T

  - ❖ () => T

  - ❖ <u>Call-by-name</u>

  - ❖ Execute method lazily.

- ❖ Int => Int => Int

  - ❖ Int => (Int => Int)

Anonymous Functions (Lambdas)

```
(0 until 10).map((value: Int) => value * value)
(0 until 10).map(value => value * value)

(0 until 10).map(value => value + 1)
(0 until 10).map(_ + 1)
```

Placeholder syntax

High-order functions

```scala
def calculateTax(rate: BigDecimal => BigDecimal,
                 salary: BigDecimal): BigDecimal = {
  rate(salary)
}


val usaTax = (salary: BigDecimal) => {
  if (salary > 413201)
    salary * 0.396
  else
    salary * 0.3
}


val twTax: BigDecimal => BigDecimal = _ * 0.25

calculateTax(usaTax, 413202)
calculateTax(twTax, 100)
```

High-Order_Function_1.sc

High-order functions

```scala
def calculateTax(rate: BigDecimal => BigDecimal,
                 salary: BigDecimal): BigDecimal = {
  rate(salary)
}

def usaTax(salary: BigDecimal) = {
  calculateTax(x =>
    if (x > 413201) x * 0.396
    else x * 0.3,
    salary)
}
def twTax(salary: BigDecimal) =
  calculateTax(_ * 0.25, salary)

usaTax(413202)
twTax(100)
```

High-Order_Function_2.sc

High-order functions

```scala
def calculateTax(rate: BigDecimal => BigDecimal)
: (BigDecimal) => BigDecimal = {
  rate
}


def usaTax = calculateTax(x =>
  if (x > 413201) x * 0.396
  else x * 0.3
)
def twTax = calculateTax(x => x * 0.25)


usaTax(413202)
twTax(100)


calculateTax(usaTax)(413202)
calculateTax(twTax)(100)
```

High-order functions - Curry

```scala
def calculateTax(rate: BigDecimal => BigDecimal)
                (salary: BigDecimal): BigDecimal = {
  rate(salary)
}


def usaTax(salary: BigDecimal) =
  calculateTax(x =>
    if (x > 413201) x * 0.396
    else x * 0.3)(salary)
def twTax(salary: BigDecimal) =
  calculateTax(x => x * 0.25)(salary)

usaTax(413202)
twTax(100)
```

High-order functions - Partially Applied Function

```scala
// Curry
def calculateTax(rate: BigDecimal => BigDecimal)
                (salary: BigDecimal): BigDecimal = {
  rate(salary)
}


// First way of Partially Applied Function
def usaTax = calculateTax(
  x =>
    if (x > 413201) x * 0.396
    else x * 0.3
)(_)
```

```scala
// Second way of Partially Applied Function
def twTax: BigDecimal => BigDecimal = calculateTax(_ * 0.25)

usaTax(413202)
twTax(100)
```

Partially_Applied_Function.sc

Exercise of High-order functions

```scala
def summation(fun: Int => Int, start: Int, end: Int): Int = ???

def number(n: Int): Int = ???
def cube(n: Int): Int = ???
// factorial number: 5 * 4 * 3 * 2 * 1
def factorial(n: Int): Int = ???


def sumNumber(start: Int, end: Int): Int = ???
def sumCubes(start: Int, end: Int): Int = ???
def sumFactorials(start: Int, end: Int): Int = ???
```

Functional Programming Principles in Scala, Martin Odersky

Exercise1.sc

Exercise of High-order functions

```scala
// hint: nest function
def summation(fun: Int => Int): (Int, Int) => Int = ???

def sumNumber = ???
def sumCubes = ???
def sumFactorials = ???
```

Functional Programming Principles in Scala, Martin Odersky

Exercise2.sc

Exercise of Curry

```scala
// Curry
def summation(fun: Int => Int)(start: Int, end: Int): Int = ???

def sumNumber = ???
def sumCubes = ???
def sumFactorials = ???
```

Functional Programming Principles in Scala, Martin Odersky

Collections

- Concept of Collections

- Collection Traits

- Hierarchy of Collections

  - Immutable

  - Mutable

- Immutable List

- Immutable HashMap

  - Tuple

  - Option

- Pattern Matching v3

- Time Complexity

Concept of Collections

❖ In most programming language, they only support mutable collections.

❖ Scala supports **immutable** collections and mutable collections.

  ❖ It means you **cannot change the data** that inside of collections.

  ❖ When you operate on a collection, it will **return a new collection** than **keep origin collection**.

❖ Almost every method is **left associativity**.

  ❖ Unless method that ends in a ":"(**colon**) is **right associativity**.

Concept of Collections

	val	var
Immutable Collection	Best	Not Bad
Mutable Collection	Danger	~~Very Danger~~

The Mutability Matrix of Pain by Jamie Allen

Hierarchy of Collections
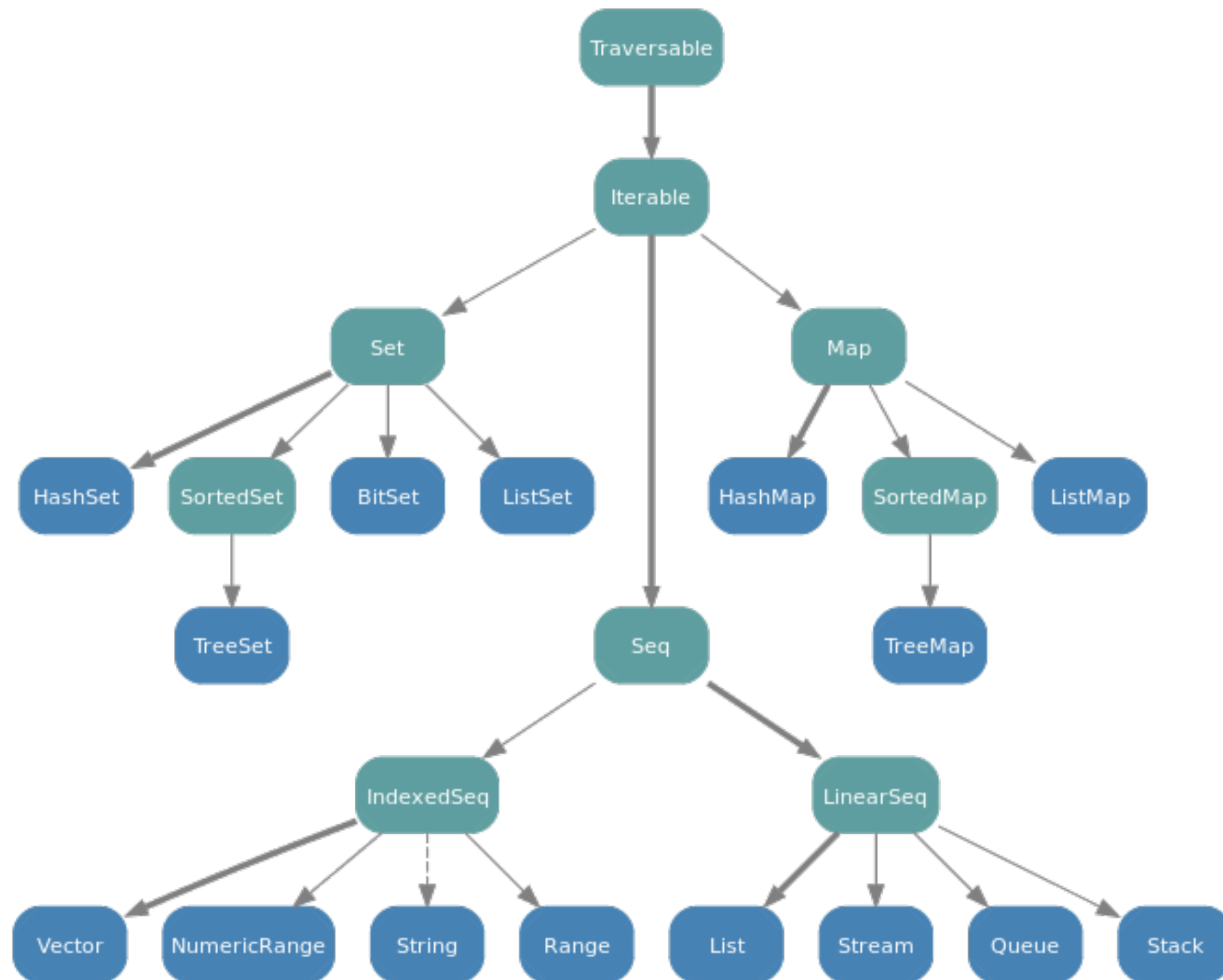
Collection Trait

❖ Traversable - traverse whole collection.

❖ Iterable - traverse whole collection **with sooner stop**.

❖ Seq

    ❖ IndexedSeq - Array

    ❖ LinearSeq - List

❖ Map - key pairs with value.

❖ Set - no duplicated item

Hierarchy of Immutable Collections

Hierarchy of Mutable Collection

Concept of Immutable List

- ❖ Nil
  - ❖ The empty list.
    - ❖ Avoid null.

Work with List

```scala
// Construct List
val list1 = List(1, 2, 3)
// Element prepends to List
val list2 = 1 :: 2 :: 3 :: Nil
val list3 = 1 :: (2 :: (3 :: Nil))
val list4 = Nil.::(3).::(2).::(1)
// List prepends to List
val list5 = List(5, 5, 6, 6) ::: List(8, 7)
val list6 = List(8, 7).:::(List(5, 5, 6, 6))
```

pronounced "cons"

Basic_List.sc

API of List

```scala
val list =
  List(4, 3, 0, 1, 2)
list.head
list.tail
list.length
list.max
list.min
list.sum
list.contains(5)
list.mkString(",")
list.drop(2)
list.reverse

list.sortWith(_ > _)
list.map(x => x * 2)
list.map(_ * 2)
list.reduce((x, y) => x + y)
list.reduce(_ + _)
list.filter(x => x % 2 == 0)
list.filter(_ % 2 == 0)
list.groupBy(x => x % 2 == 0)
list.groupBy(_ % 2 == 0)
list.indices zip list
list.zipWithIndex
```

List with high-order function

```scala
val vector = Vector(1, 2, 3)
val list = List(
  vector.map(_ + 10),
  vector.map(_ - 10),
  vector.map(_ * 100)
)

list.flatten

val flatmap =
  list.flatMap(x => x.filter(_ % 2 == 1))

val mapThenFlat =
  list.map(x => x.filter(_ % 2 == 1)).flatten

flatmap == mapThenFlat
```

List_With_High-Order_Function.sc

Concept of Immutable HashMap

❖ Tuple

❖ Option

   ❖ Some

   ❖ None

Tuple

- Immutable

- Contain **different type** of elements
  - You don't need Java-Bean

- It can contain to 22 elements.

- Tuple2 = Pair

```
val people =
  ("Martin", 58)
people._1
people._2
```

Option

- Use to avoid null.
  - Null is error prone.
  - You code need to check whether or not it is null.
- Option type will be Some **or** None.

Work with HashMap

```scala
val countryCode =
  Map("Taiwan" -> 886,
    "United States" -> 1,
    "Japan" -> 81)
val countryCode1 =
  Map(("Taiwan", 886),
    ("United States", 1),
    ("Japan", 81))

// 886
countryCode("Taiwan")
// Some(886)
countryCode.get("Taiwan")
```

```scala
// NoSuchElementException
countryCode("Canada")
// None
countryCode.get("Canada")
// add
countryCode + ("China" -> 86)
// delete
countryCode - ("Japan")
// update
countryCode + ("Taiwan" -> 5566)
countryCode.getOrElse("China", 0)
```

HashMap.sc

More Option

```scala
val list = List(0, 1, 2, 3, 4, 5)

def odd(n: Int) = {
  if (n % 2 == 1) Some(n)
  else None
}

list.map(odd(_))
list.flatMap(odd(_))

val listOption =
  List(Some(1), None, Some(2), None, Some(3))

Some(5566).map(x => x * 2)
listOption.flatMap(x => x.map(y => y * 2))
listOption.flatMap(_.map(_ * 2))
```

Option.sc

Advanced Option

```scala
case class People(
                id: String,
                firstName: String,
                lastName: String,
                age: Int,
                countries: List[String],
                gender: Option[String]
            )


object PeopleRepository {
  private val peoples = Map(
    1 -> People("1", "John", "Doe", 30,
List("TW", "USA"), Some("male")),
    2 -> People("2", "Janie", "Doe", 10,
List("Japan"), Some("female")),
    3 -> People("3", "", "Doe", 50, List("TW"),
None))

  def find(id: Int): Option[People] =
peoples.get(id)
  def getAll = peoples.values
}
```

```scala
// None
for {
  people <- PeopleRepository.find(2)
  gender <- people.gender
} yield gender
// Some(female)
for {
  people <- PeopleRepository.find(3)
  gender <- people.gender
} yield gender
// None
for {
  people <- PeopleRepository.find(4)
  gender <- people.gender
} yield gender
```

The Neophyte's Guide to Scala Part 5: The Option Type

Advanced_Option.sc

For-Comprehensions

```scala
// List of People has TW passport
for {
  people <- PeopleRepository.getAll
  if (people.countries.contains("TW"))
} yield people

(for {
  people <- PeopleRepository.getAll
  country <- people.countries
} yield country).toSet
```

For_Comprehension.sc

Exercise of Collection

```scala
// only use collection's API(combinator)

// find the maximum number of Seq with fun
def larges(fun: Int => Int,
           inputs: Seq[Int]): Int = ???


// find the maximum number of index of Seq with fun
def largesIndex(fun: Int => Int,
                inputs: Seq[Int]): Int = ???
```

Scala for the Impatient by Cay Horstmann

Pattern Matching v3

```scala
val list = (0 to 5).toList
def sum(list: List[Int]): Int = {
  list match {
    case Nil => 0
    case head :: tail => head + sum(tail)
  }
}
sum(list)
```

Pattern Matching v3

```scala
import scala.annotation.tailrec

val list = (0 until 9).toList
def sumTail(list: List[Int]): Int = {
  @tailrec
  def help(list: List[Int],
          previous: Int): Int = list match {
    case Nil => 0
    case List(onlyOne) => previous + onlyOne
    case head :: tail => help(tail, previous + head)
  }
  help(list, 0)
}
sumTail(list)
```

Pattern_Matching_With_Tail_Recursion.sc

Time complexity

- ❖ C: constant time, O(1)

- ❖ eC: effectively constant time

- ❖ aC: amortised constant time

  - ❖ Some of operation take longer. On average, it is constant time.

- ❖ Log: O(log n)

- ❖ L: linear time, O(n)

Collection Performance – Immutable Sequence

	head	tail	apply	update	prepend	append	insert
List	**C**	**C**	L	L	C	**L**	
Stream	**C**	**C**	L	L	C	**L**	
Vector	eC	eC	eC	eC	eC	eC	
Stack	C	C	L	L	C	C	L
Queue	aC	aC	L	L	L	C	
Range	C	C	C				
String	C	L	C	L	L	L	

http://docs.scala-lang.org/overviews/collections/performance-characteristics.html

Collection Performance – Mutable Sequence

	head	tail	apply	update	prepend	append	insert
ArrayBuffer	C	L	C	C	**L**	aC	L
ListBuffer	C	L	L	L	**C**	**C**	L
StringBuilder	C	L	C	C	**L**	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C			
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C			

http://docs.scala-lang.org/overviews/collections/performance-characteristics.html

Collection Performance – Set & Map

	lookup	add	remove	min
immutable				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC*
ListMap	L	L	L	L
mutable				
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC
TreeSet	Log	Log	Log	Log

http://docs.scala-lang.org/overviews/collections/performance-characteristics.html

Summary of Scala

❖ Keep it simple.

❖ **Don't pack too much in one expression.**

❖ Find meaningful names.

  ❖ If you are hard to find meaningful names, maybe you have wrong abstraction.

❖ Prefer functional.

❖ **But don't diabolise local state.**

❖ Careful with mutable objects.

Scala with Style by Martin Odersky

Appendix

- ❖ What are class and object?

- ❖ Implicit

- ❖ sbt

Class and Object

❖ Class

  ❖ Class likes template, memory layout, or memory structure.

❖ Object

  ❖ Object is an instance of class where the object contains variables, functions, data structures.

  ❖ Objects likes a actual memory block.

Implicit

- Implicit

  - Compiler will help you to do something implicitly. Opposite to you have to write it yourself explicitly.

- Implicit in IntelliJ - you shall see underline in the method.

```
10.printFiveTimes
```

Implicit

❖ Features of implicit

   ❖ Implicit parameter

   ❖ Implicit (type) conversions

      ❖ Implicit method

      ❖ Implicit class

Implicit Parameter

```scala
// declaration
def intToString(implicit n: Int) = n.toString

def max(x: Int)(implicit y: Int) = {
  x + y
}

implicit val x = 5566

// execution
intToString

max(8787)
```

Implicit Method

```scala
import scala.language.implicitConversions

class MyInteger(n: Int) {
  def printFiveTimes = (0 until 5).foreach(println(_))
}

implicit def implicitIntToString(n: Int) = new MyInteger(n)

10.printFiveTimes
```

ImplicitMethod.sc

Implicit Class

```scala
implicit class MyInteger(n: Int) {
  def printFiveTimes = (0 until 5).foreach(println(_))
}


10.printFiveTimes
```
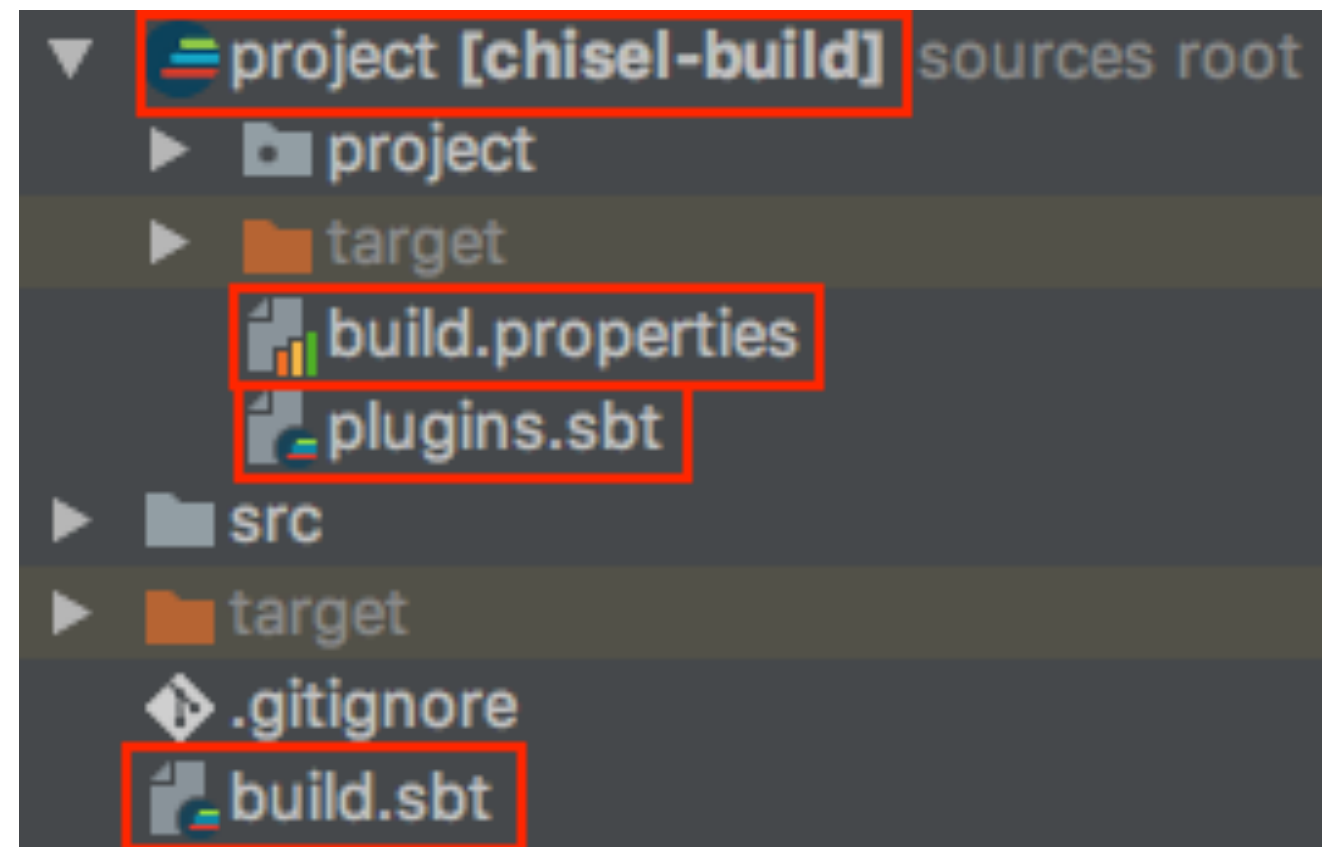
sbt

❖ What is sbt?

  ❖ sbt is the abbreviation of Simple Build Tool.

  ❖ Functionality of sbt is similar to makefile or CMake for C/C++.

  ❖ In Java, it is the same as Maven or Gradle.

    ❖ sbt has **incremental** compilation.

  ❖ But it is more powerful than makefile.

  ❖ You could write your own plugin.

sbt structure

- build.sbt

  - build definition.

- project - meta-build of root project

  - build.properties

    - sbt version

  - plugins.sbt

    - sbt plugin

sbt keys

- Build-in Keys

  - SettingKey[T]

    - name, version, scalaVersion, libraryDependencies…etc

  - TaskKey[T]

    - clean, compile, test, console…

  - InputKey[T]

    - run, runMain…etc

sbt

- ❖ Features of implicit
  - ❖ Implicit parameter
  - ❖ Implicit (type) conversions
    - ❖ Implicit method
    - ❖ Implicit class

Further Reading & Reference

- Video

  - Scala for the Intrigued by Venkat Subramaniam

  - Scala with Style by Martin Odersky

- Online Tutorial

  - Scala School by Twitter

  - Creative Scala by Underscore

  - Scala Official Tutorial

  - Scala 101 by Lightbend

  - hello-scala.com

  - Scala Exercises

- Other

  - Scala 2.12.4 API Documentation

Further Reading & Reference

- Book

  - Cay Horstmann. <u>Scala for the Impatient, 2nd</u>. Available from <u>http://www.informit.com/store/scala-for-the-impatient-9780134540634</u>

    - (Beginning)

  - Martin Odersky, Lex Spoon, and Bill Venners. **<u>Programming in Scala, 3rd</u>**. Available from <u>http://www.artima.com/shop/programming_in_scala_3ed</u>

    - (Beginning, Intermediate, Advance)

  - Joshua D. Suereth. <u>Scala in Depth</u>. Available from <u>https://www.manning.com/books/scala-in-depth</u>

    - (Advance)

  - Jason Swartz. <u>Learning Scala</u>. Available from <u>http://shop.oreilly.com/product/0636920030287.do</u>

Further Reading

- SBT

  - [SBT Basic Concepts by Pishen Tsai](#)

  - [SBT Official Document](#)

- Design

  - [Functional Patterns in Scala by Walter Chang](#)

  - [Functional Programming in Scala by Paul Chiusano and Rúnar Bjarnason](#)

- Test

  - [ScalaTest Official Guild](#)

Further Reading

* **S**tructure and **I**nterpretation of **C**omputer **P**rograms by Harold Abelson, Gerald Jay Sussman, Julie Sussman

    * Books - https://mitpress.mit.edu/sicp/full-text/book/book.html

    * Videos - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-2005/video-lectures/

* It's not teach you Scala, but it teaches **the principles of computer programming**. Even Martin's Coursera courses borrow concepts from this material.