

# Introduction to Scala

<https://gitter.im/ScalaTaiwan/ScalaTaiwan>

Jimin Hsieh - Speaker - <https://tw.linkedin.com/in/jiminhsieh>

Pishen Tsai - TA - <https://github.com/pishen>

Vito Jeng - TA - [vito@is-land.com.tw](mailto:vito@is-land.com.tw)

Walter Chang - TA - <https://twitter.com/weihsiu>

Hackpad - <https://goo.gl/SIfDk7>



# Agenda

- ❖ Why Scala?
- ❖ Scala concept
- ❖ Program with Scala
- ❖ Imperative
- ❖ Object-Oriented
- ❖ Functional
- ❖ Collections
- ❖ Summary of Scala
- ❖ Further Reading & Reference
- ❖ Special Thanks

# 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

# 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/author/11531-Alan\\_Perlis](http://www.azquotes.com/author/11531-Alan_Perlis)



# Why Scala?

 CHANGEMAKERS.COM / PAULGRAHAM

NEXT QUOTE 

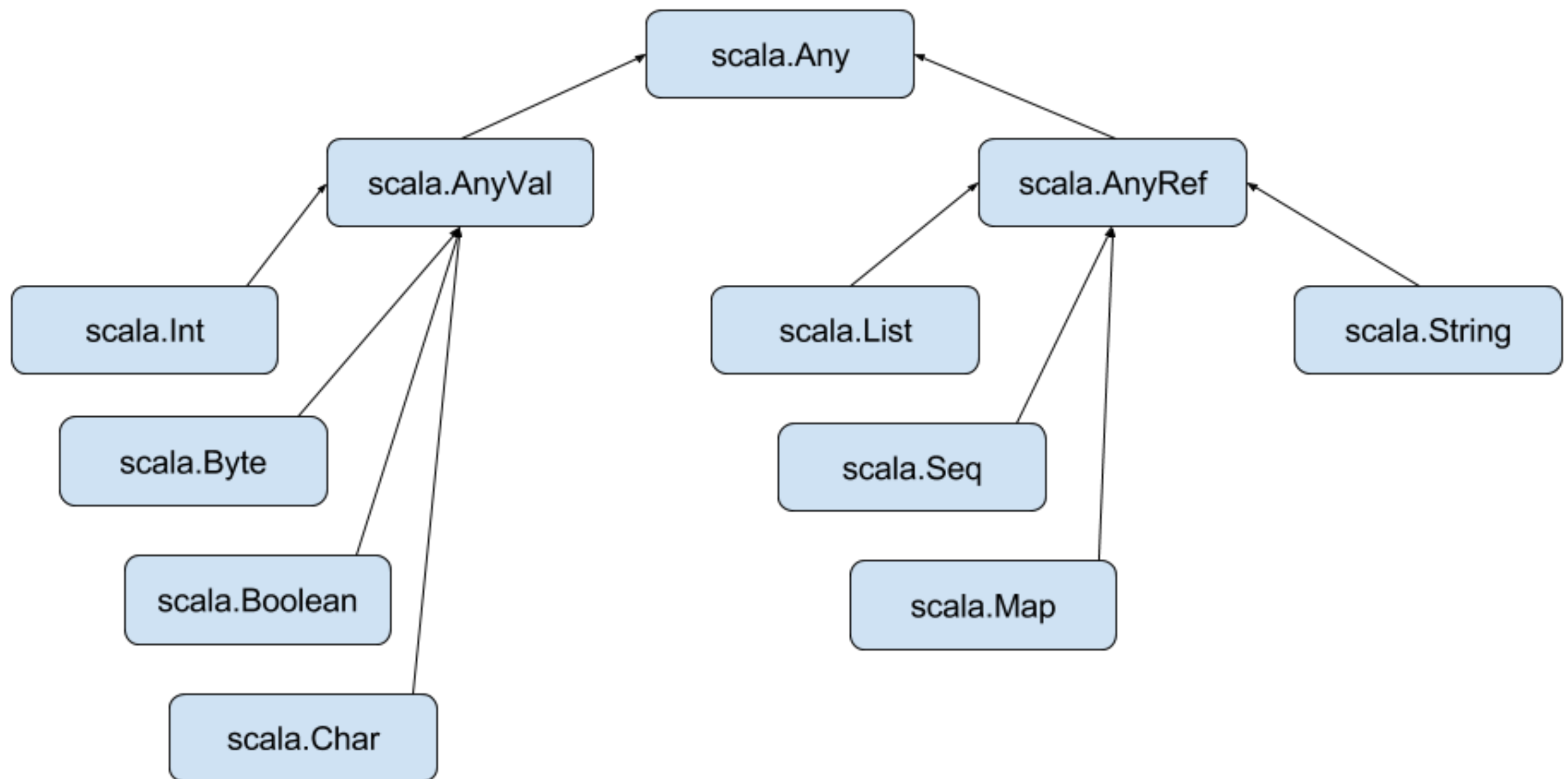
“A programming language is for thinking about programs, not for expressing programs you've already thought of. It should be a pencil, not a pen.”

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

# Scala concept

- ❖ Everything is an object.
- ❖ 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

- ❖ 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)

```
object Demo1 {  
  val todayEvent = "JCConf"  
  val workshop = "Introduction to Scala"  
  lazy val fun = (0 to 4).map(x => "fun").mkString(" ")
```

```
  def main(args: Array[String]): Unit = {  
    println("Hello everybody!")  
    print("Welcome to " + todayEvent + "!\n")  
    println("I hope you can enjoy this workshop - "  
      + workshop + ". :P")  
    print("Scala is so much " + fun + "!!")  
  }  
}
```

Object with main method.

# Program with Scala(App)

Object with App trait.

```
object Demo2 extends App {  
  val todayEvent = "JCConf"  
  val workshop = "Introduction to Scala"  
  lazy val fun =  
    (0 to 4).map(x => "fun").mkString(" ")  
  
  println("Hello everybody!")  
  print("Welcome to " + todayEvent + "!\n")  
  println("I hope you can enjoy this workshop - "  
    + workshop + ". :P")  
  print("Scala is so much " + fun + "!!")  
}
```



# Program with Scala(REPL)

## ❖ REPL - **R**ead-**E**valuate-**P**rint **L**oop

```
Welcome to Scala 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_102).  
Type in expressions for evaluation. Or try :help.
```

```
scala> val language = "scala"  
language: String = scala
```

```
scala> language.toUpperCase  
res0: String = SCALA
```

```
scala> █
```

# Program with Scala(REPL)

❖ \$ scala -Dscala.color

```
Welcome to Scala 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_102).  
Type in expressions for evaluation. Or try :help.
```

```
scala> val language = "scala"  
language: String = scala
```

```
scala> language.toUpperCase  
res0: String = SCALA
```

```
scala> █
```

# Program with Scala(Worksheet)

- ❖ Work with worksheet.
- ❖ **IntelliJ**
  - ❖ <https://www.jetbrains.com/help/idea/2016.2/working-with-scala-worksheet.html>
- ❖ 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 is 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.

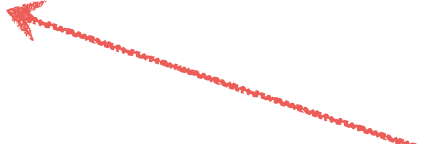


# Expression with semicolon?

```
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 has return value.(expression)
  - ❖ 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

```
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

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

def

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

No function's result type





def

```
def max(x: Int, y: Int) =
```

```
    if (x > y)
```

```
        x
```

```
    else
```

```
        y
```

No curly brackets

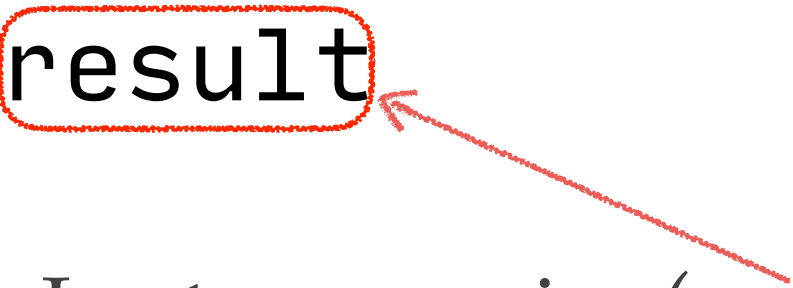


# Summary of def

- ❖ 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 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

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



# For-Loop

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

# For-Loop

```
val n = 5
```

```
for { i <- 0 to n  
      j <- 0 to n } {
```

```
  print("*")
```

```
  if (j == n)
```

```
    println("")
```

```
}
```

With curly bracket - "{",  
you can use newlines without semicolon - ";".

# Exercise of For-Loop

- ❖ Print out something like below.

```
*  
**  
***  
****  
*****  
*****
```

# Nested Function

```
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.

# Recursion vs Tail-recursion

- ❖ Factorial number

- ❖  $6! = 6 * 5 * 4 * 3 * 2 * 1$

# Recursion vs Tail-recursion

*// Recursion*

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

```
factorial(15000)
```

java.lang.StackOverflowError

# Recursion vs Tail-recursion

```
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**.

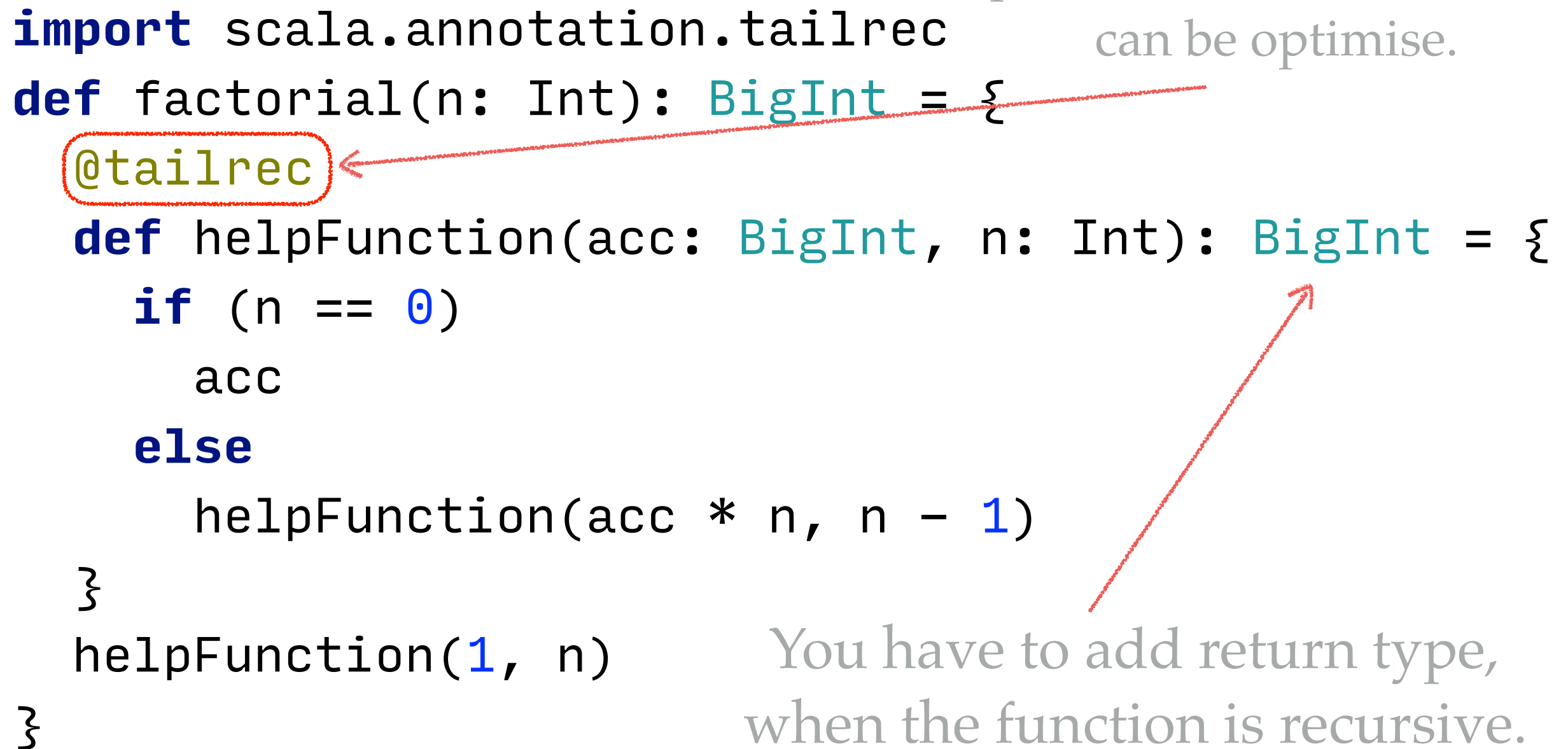


# Recursion vs Tail-recursion

Add annotation is a good habit.

Compiler can check whether or not  
can be optimise.

```
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.

# 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

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

matchString("Dog")

matchString("Human")

# 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](http://www.leaningtowerofpisa.net/fibonacci.html)

# Object-oriented

- ❖ Class
- ❖ Extends, With, Override
- ❖ Abstract Class
- ❖ Object, Companion Object
- ❖ Trait
- ❖ Access Modifiers
- ❖ Case class
- ❖ Pattern matching v2

# Class

Primary Constructor

```
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)  
}
```

val in constructor  
will give you getter

var in constructor will  
give you setter and getter

Auxiliary Constructor

```
def getAddress = address  
}
```

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

# 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

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

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

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

Sealed classes

1. Subclasses should be in the same file.

2. Compiler can check you have covered all of cases in pattern matching.

# Object

- ❖ Object is a singleton.
- ❖ 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?
  - ❖ <http://programmers.stackexchange.com/a/179402>

# Object

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

# 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**.

# 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

```
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)  
  }
```

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

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

<https://scalafiddle.io/sf/i3KWOf/0>

# 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 **optimised for use in pattern matching**.
- ❖ Each of the constructor parameters becomes a **val**.

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”**. (Factory pattern)
- ❖ It contain an **unapply** method work as **extractor** to extract values from apply method.
- ❖ Each instance contains `toString`, `equals(==)`, `hashCode`, and `copy`.

Scala for the Impatient by Cay Horstmann

# Pattern Matching v2

```
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

```
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))))
```

# 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
- ❖ 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



# 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

```
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 functions

```
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 functions

```
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

```
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)
```

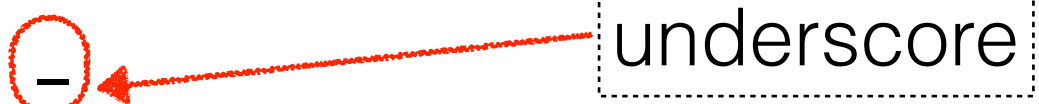
```
usaTax(413202)
twTax(100)
```

# High-order functions - Partially Applied Function

*// 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
) 
```

*// Second way of Partially Applied Function*

```
def twTax: BigDecimal => BigDecimal = calculateTax(_ * 0.25)
```

```
usaTax(413202)
```

```
twTax(100)
```

# Exercise of High-order functions

```
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

# Exercise of High-order functions

*// hint: nest function*

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

```
def sumNumber = ???
```

```
def sumCubes = ???
```

```
def sumFactorials = ???
```

Functional Programming Principles in Scala, Martin Odersky



# Exercise of Curry

*// 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

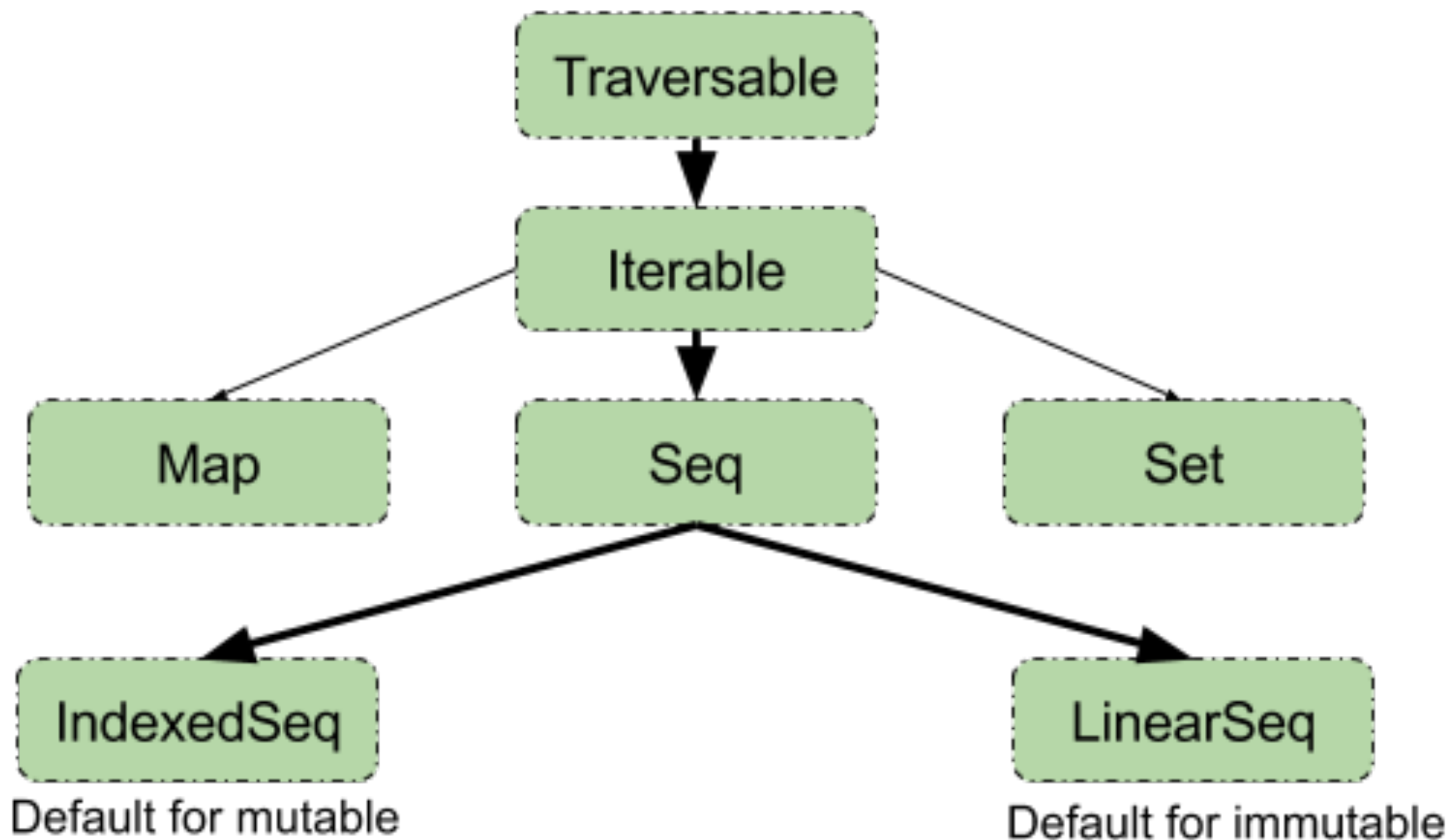
- ❖ In most programming language, they only support mutable collections.
- ❖ Scala supports mutable collections and **immutable** collections.
  - ❖ It means you **cannot change the data** that inside of collections.
  - ❖ When you operate on a collection, it will **return a new collection** than **keep 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	<del>Very Danger</del>

The Mutability Matrix of Pain by Jamie Allen

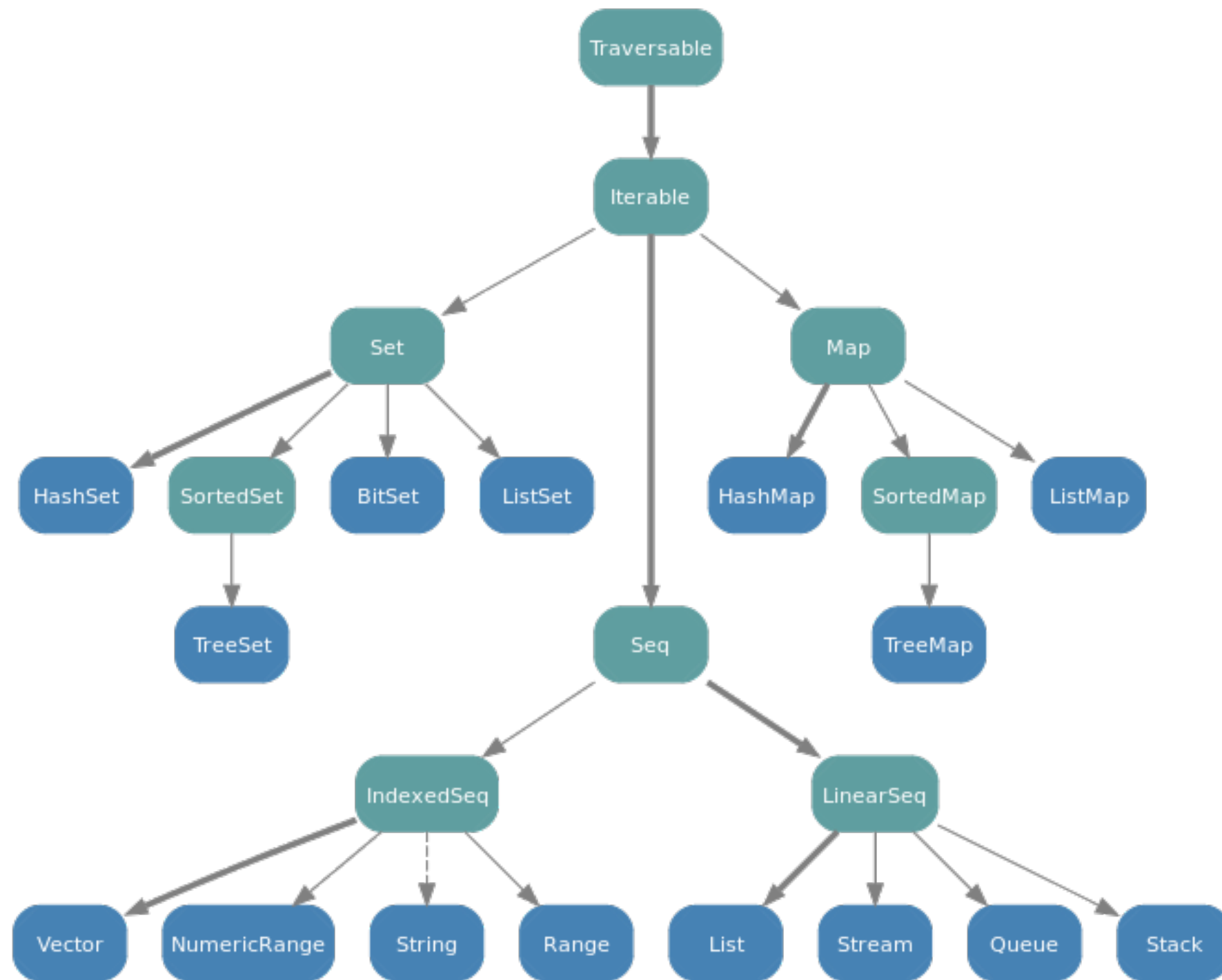
# Hierarchy of Collections



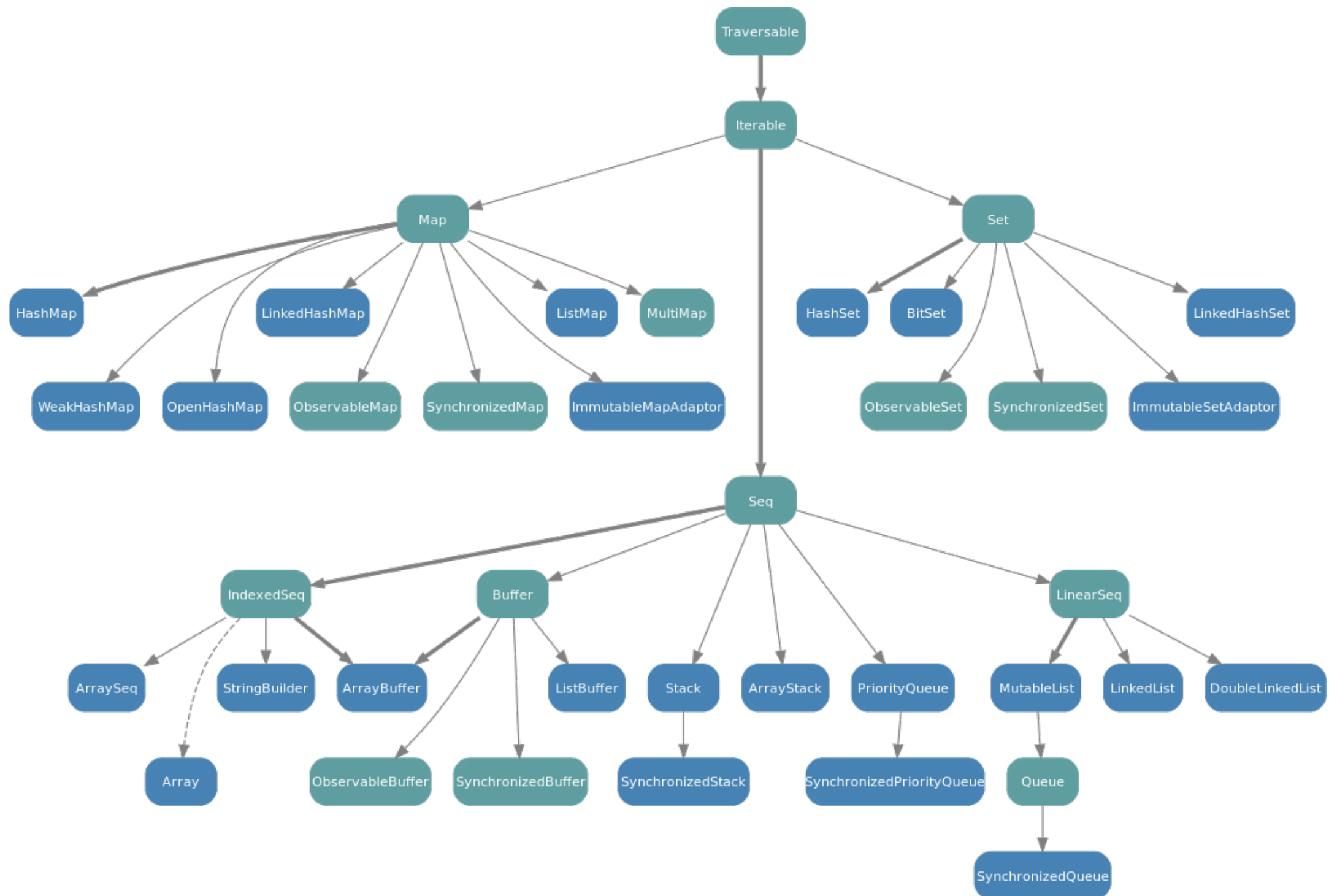
# Collection Trait

- ❖ 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



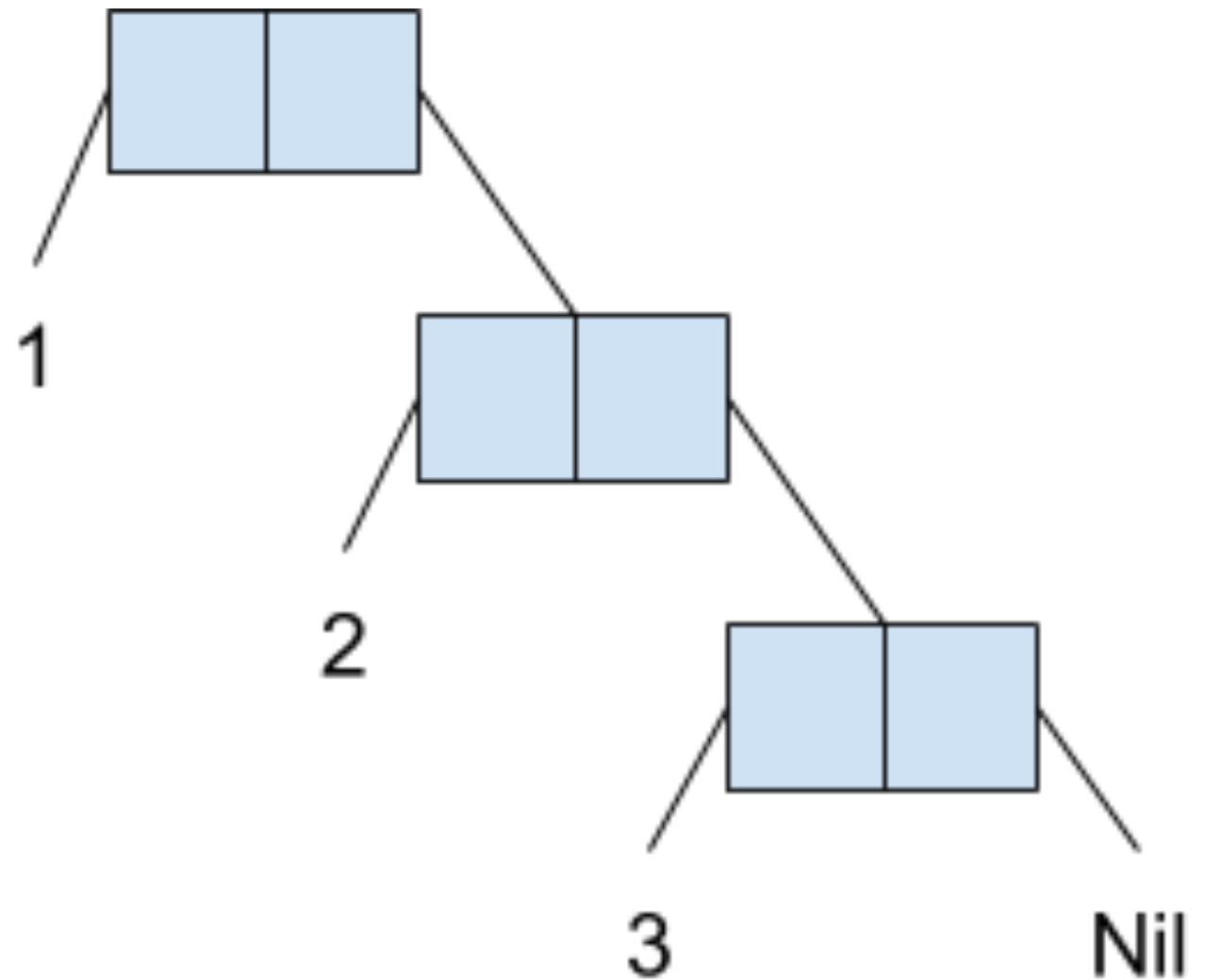
# Hierarchy of Mutable Collection





# Concept of Immutable List

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



# Work with List

*// Construct List*

```
val list1 = List(1, 2, 3)
```

*// Element prepends to List*

```
val list2 = 1 :: 2 :: 3 :: Nil
```

pronounced “cons”

```
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)
```

# API of List

```
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

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

```
val twoDim = list.map(_ + 1) ::  
  list.map(_ - 1) ::  
  list.map(_ * 2) ::  
  Nil
```

```
twoDim.flatten
```

```
// map + flatten
```

```
twoDim.flatMap(x => x.filter(_ % 2 == 1))
```

# Concept of Immutable HashMap

- ❖ Tuple
- ❖ Option
  - ❖ Some
  - ❖ None

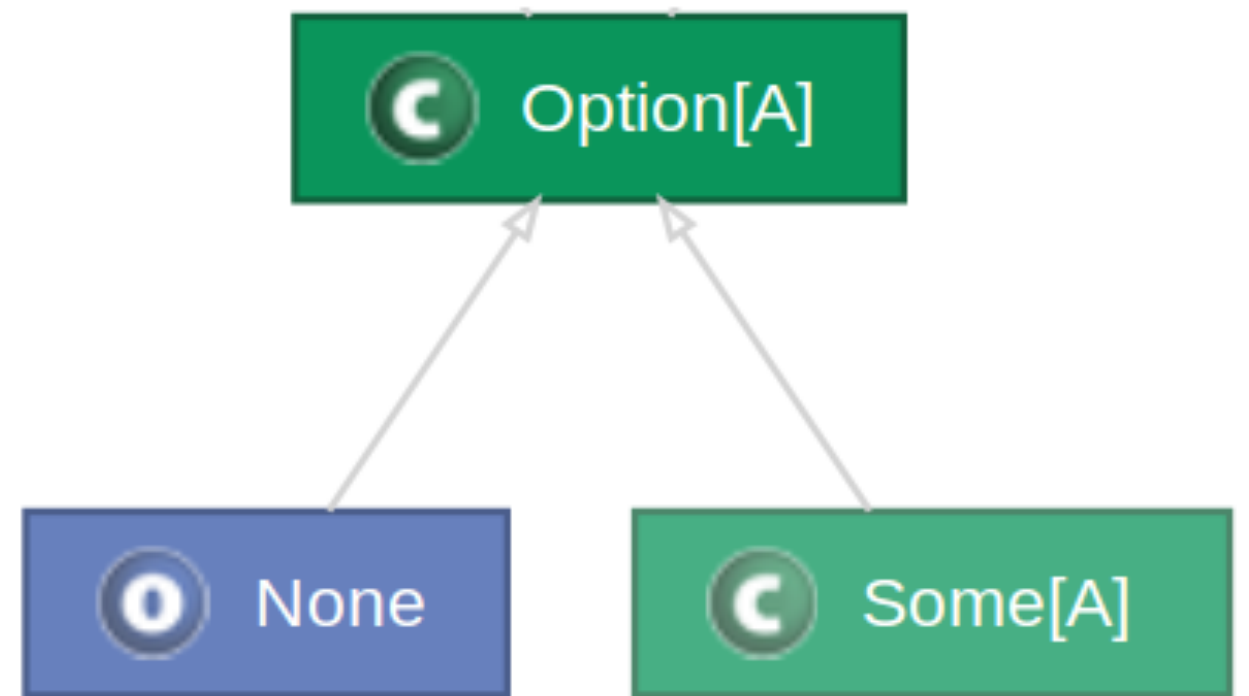
# Tuple

- ❖ 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.
- ❖ Your code needs to check whether or not it is null.
- ❖ Option type will be Some **or** None.



<http://beteheess.github.io/talks/2014/06/06-cata-visitor-adt#14>

# Work with HashMap

```
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")
```

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



# More Option

```
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 list1 =
```

```
  List(Some(1), None, Some(2), None, Some(3))
```

```
Some(5566).map(x => x * 2)
```

```
list1.flatMap(x => x.map(y => y * 2))
```

```
list1.flatMap(_.map(_ * 2))
```

# Advanced Option

```
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  
}
```

```
// 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

# For-comprehensions

*// 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
```

# Exercise of Collection

*// 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

```
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

```
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)
```

# 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	<b>L</b>	aC	L
ListBuffer	C	L	L	L	<b>C</b>	<b>C</b>	L
StringBuilder	C	L	C	C	<b>L</b>	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
<b>immutable</b>				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC*
ListMap	L	L	L	L
<b>mutable</b>				
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

# 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](#)
- ❖ [Scala Exercises](#)

## ❖ Other

- ❖ [Scala 2.11.8 API Documentation](#)

# Further Reading & Reference

## ❖ Book

- ❖ Jason Swartz. Learning Scala. Available from <http://shop.oreilly.com/product/0636920030287.do>
- ❖ Cay Horstmann. **Scala for the Impatient**. Available from <http://www.informit.com/store/scala-for-the-impatient-9780321774095>
  - ❖ (Beginning)
- ❖ Martin Odersky, Lex Spoon, and Bill Venners. **Programming in Scala, 3rd**. Available from [http://www.artima.com/shop/programming\\_in\\_scala\\_3ed](http://www.artima.com/shop/programming_in_scala_3ed)
  - ❖ (Beginning, Intermediate, Advance)
- ❖ Joshua D. Suereth. Scala in Depth. Available from <https://www.manning.com/books/scala-in-depth>
  - ❖ (Advance)

# 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.

# Special Thanks

- ❖ (By Alphabet)

- ❖ Pishen Tsai

- ❖ <https://github.com/pishen>

- ❖ Vito Jeng

- ❖ <https://github.com/vitojeng>

- ❖ Walter Chang

- ❖ <https://twitter.com/weihsiu>

- ❖ **Thanks for the most pull requests!!!**

- ❖ Zanyking

- ❖ <http://zanyking.blogspot.tw>