Chương I. Tìm hiểu Play framework (Văn Cảnh)

1. Giới thiệu
2. Play framework là gì?
3. Lịch sử phát triển
4. Khác nhau giữa Play framework version 1 và version 2
5. Đặc điểm Play framework (Ngọc Dân)
6. Mô hình MVC
7. HTTP Routing
8. Controller
9. View
10. Model
11. Validation
12. Testing
13. Đánh giá – Nhận xét (Văn Cảnh)
14. So sánh với Spring framework
15. Ưu điểm – Nhược điểm
16. Hướng dẫn cài đặt - Ứng dụng Helloworld

Chương II. Ngôn ngữ Scala

1. Giới thiệu (Ngọc Dân)
2. Cơ bản về Scala
3. Basic Syntax
4. Data Type
5. Variable
6. Access Modifier
7. Operators
8. If-else
9. Loop Type
10. Function (Văn Cảnh)

* Function được hiểu là một khối code thực hiện một logic nào đó và có thể được sử dụng lại.
* Function trong Scala được mô tả bằng hai tính chất:
* Được đặt tên
* Là một hay nhiều biểu thức có thể tái sử dụng
* Function trong Scala mang nhiều đặc điểm của Fuction programming.
  1. **Cách khai báo Function trong Scala**

**Input-less Function (không có input)**

Syntax: *def<identifier> = <expression>*

Vd:

scala> def hello = "Hello"  
hello: String  
scala> hello  
res0: String = Hello

Một cách khác để khai báo một function không có input đó là dùng một cặp dấu ngoặc đơn () – Empty parenthese

scala> def helloWorld() = "Hello World"  
helloWorld: ()String

Điểm khác biệt giữa hai cách khai báo này là:

scala> helloWorld ()  
res1: String = Hello World  
scala> helloWorld  
res2: String = Hello World  
scala> hello()  
<console>:12: error: not enough arguments for method apply: (index: Int)Char in class StringOps.  
Unspecified value parameter index.  
hello()

**Function with a return type (trả về kiểu cố định)**

Như bạn thấy ở trên, nếu bạn không set kiểu trả về cho function, Scala sẽ tự tính toán và set cho bạn, trong thực tế bạn nên tự set kiểu cho function mình viết, điều đấy khiến cho code của bạn rõ ràng hơn và tránh được nhiều sai sót ngoài ý muốn.

Syntax:*def <identifier>: <type> = <expression>*

Vd:

scala> def kieuTraVeLaString : String = "đây là 1 String"  
kieuTraVeLaString: String  
scala> def kieuTraVeLaInt : Int = 100  
kieuTraVeLaInt: Int

**Định nghĩa đầy đủ 1 Function**  
Syntax: *def <identifier>(<identifier>: <type>[, ... ]): <type> = <expression>*

Vd:

scala> def cong2So(soThuNhat: Int , soThuHai: Int) : Int = soThuNhat + soThuHai  
cong2So: (soThuNhat: Int, soThuHai: Int)Int  
scala> cong2So(2,3)  
res4: Int = 5

Scala cho phép bạn gọi function với các input với tên được định nghĩa trước đó,ta thử gọi lại function trên với thứ tự input đảo ngược:

scala> cong2So(soThuHai = 3,soThuNhat = 2)  
res5: Int = 5

**Vararg Parameters**

Đôi khi bạn định nghĩa 1 function với với input cùng kiểu và không biết rõ số lượng , Scala đồng ý bạn viết như sau :

scala> def tinhTongNhieuSo(cacSo: Int\*): Int = {  
        | var tong = 0  
        | for (i <- cacSo) tong += i  
        | tong  
        | }  
tinhTongNhieuSo: (cacSo: Int\*)Int  
scala> tinhTongNhieuSo(1,2,3)  
res6: Int = 6  
scala> tinhTongNhieuSo(5,10,15,20)  
res7: Int = 50

**Parameter Groups**

Ngoài ra ta có 1 cách khai báo biến khác trong scala, đó là tách thành nhiều group parameter.

scala> def soNaoLon(soThuNhat: Int)(soThuHai: Int) = if(soThuNhat > soThuHai) soThuNhat

else soThuHai  
soNaoLon: (soThuNhat: Int)(soThuHai: Int)Int  
scala> soNaoLon(3)(1)  
res8: Int = 3  
scala> soNaoLon(3)(5)  
res9: Int = 5

**Function Invocation with Expression Blocks**

Theo cách thông thường ,bạn sẽ chuẩn bị trc input và sau đó gọi function .  
Vd: viết hàm kiểm tra 1 số là chẵn hay lẻ :

scala> def chanHayLe(x : Int) {  
         |    if (x % 2 == 0) println("so chan")  
         |    else println("so le")  
         | }  
chanHayLe: (x: Int)Unit

Giờ tôi sẽ cộng 2 số bất kì và kiểm tra xem  số đó là chẵn hay lẻ:

scala> val tong2SoBatKy = 5 + 9  
tong2SoBatKy: Int = 14  
scala> chanHayLe(tong2SoBatKy)  
so chan

Với Scala mọi thứ đơn giản hơn 1 bước :

scala> chanHayLe { 5 + 9}  
so chan

**Function’s Type Parameters**

Đôi khi bạn cần viết 1 function mà kiểu trả về là động , ví dụ bạn viết 1 function mà kiểu trả về tưởng ứng với kiểu input.

Cách 1: Sử dụng Any vì Any là root của mọi data trong Scala.

scala> def anyThings(x : Any) : Any = x  
anyThings: (x: Any)Any

scala> val soInt: Int = anyThings(1)

<console>:11: error: type mismatch;

 found   : Any

required: Int

val soInt: Int = anyThings(1)

Ở đây anyThings của 1 không trả về 1 số Int vì qua hàm anyThings nó đã biến thành kiểu Any

Cách 2: Scala cung cấp cho bạn 1 giải pháp cho tình huống này:

scala> def anyThings[A](x : A) : A = x

anyThings: [A](x: A)A  
scala> val soInt: Int = anyThings(1)  
soInt: Int = 1

**Nested Functions**

Scala cho phép bạn khai báo function trong function.

scala> def timSoLonTrong3So(soI: Int, soII: Int, soIII: Int) = {  
            |    def timSoLonTrong2So(soI: Int, soII: Int) = if(soI > soII) soI else soII  
            |    timSoLonTrong2So(soI,timSoLonTrong2So(soII,soIII))  
            | }  
timSoLonTrong3So: (soI: Int, soII: Int, soIII: Int)Int  
scala> timSoLonTrong3So(5,15,10)  
res1: Int = 15

**Procedures**

Procedures là function mà không trả về giá trị gì cả hay đúng hơn nó trả về 1 Unit.

scala> def procedure(ten: String) = println("Hello "+ ten)  
procedure: (ten: String)Unit  
scala> procedure("UIT")  
Hello UIT

1. First Class Function – Closures

* Một trong những điều cốt lõi của functional programming là function nên là fisrt-class, đơn giản bạn có thể hiểu nó không chỉ đơn giản được định nghĩa và được invoked mà nó được sử dụng như một phần của ngôn ngữ lập trình như là một data type.
* Vậy *first-class function* :
* Có thể được coi như các loại data type khác.
* Có thể được tạo từ một literal.
* Được lưu trữ như một value, một variable hay một data structure.
* Được sử dụng như một parameter cho một function khác – gọi là *higher-order functions.*
  1. Kiểu dữ liệu Function

Kiểu dữ liệu của function được diễn tả bằng một cặp [] bao gồm một nhóm input type và output type được nối với nhau bằng một mũi tên “=>”

Syntax: ([<type>, . . .]) => <type>

* 1. Function Literals

Function literals có đặc điểm:

* + - Syntax: ([<identifier>: <type>, ... ]) => <expression>
    - Không có định danh (Anonymous Function – là khi ta viết một hàm mà hàm đó chỉ sử dụng một lần duy nhất)
    - Vd:

scala> val maximize = (a: Int, b: Int) => if (a > b) a else b  
maximize: (Int, Int) => Int = <function2>

Ở đây (a: Int, b: Int) => if (a > b) a else b là một Literal Function

* 1. Higher-Order-Function

Là một function mà có đối số là một function khác hoặc có kết quả trả về là một function.

Vd:

scala> def dollar2Vnd(number: Int): Int = number \* 21750  
dollar2Vnd: (number: Int)Int

scala> def soDu(soTienDo : Int,f : (Int) => Int){  
             | println(f(soTienDo)+" VND")  
             | }  
soDu: (soTienDo: Int, f: Int => Int)Unit

scala> soDu(5,dollar2Vnd)  
108750 VND

* 1. By-Name Parameter

Là function mà ta khai báo với hệ thống là ta chỉ quan tâm đến output – giá trị trả về của function. Hàm được khai báo với input đầu vào là kiểu dữ liệu Unit hoặc (), kiểu dữ liệu output được khai báo để trả về giá trị của hàm.

Vd:

scala> def nhanDoi (x : => Int) : Int = x \* 2  
nhanDoi: (x: => Int)Int

scala> nhanDoi(4)  
res20: Int = 8

* 1. Partial Function

- Một partial function là một trait mà khi thi hành có thể sử dụng để xây dựng chương trình để giải quyết vấn đề.

- Một trait PartialFunction yêu cầu phương thức “isDefineAt” và “apply” được thực thi.

Vd:

val doubleEvens: PartialFunction[Int, Int] = new PartialFunction[Int, Int] {

//partial function đảm nhận nhiệm vụ

def isDefinedAt(x: Int) = x % 2 == 0

//nếu nhiệm vụ phù hợp thì sẽ thực hiện chức năng này

def apply(v1: Int) = v1 \* 2

}

doubleEvens: PartialFunction[Int,Int] = <function1>

val tripleOdds: PartialFunction[Int, Int] = new PartialFunction[Int, Int] {

def isDefinedAt(x: Int) = x % 2 != 0

def apply(v1: Int) = v1 \* 3

}

tripleOdds: PartialFunction[Int,Int] = <function1>

val whatToDo = doubleEvens orElse tripleOdds

whatToDo: PartialFunction[Int,Int] = <function1>

whatToDo(3)

res3: Int = 9

whatToDo(4)

res4: Int = 8

* + Câu lệnh “case” là một cách nhanh chóng để tạo ra Partial Function. Khi tạo câu lệnh “case” thì “apply” và “isDefineAt” được tạo cho bạn.
  + Vd:

val doubleEvens: PartialFunction[Int, Int] = {

case x if (x % 2) == 0 => x \* 2 //Case statement

}

doubleEvens: PartialFunction[Int,Int] = <function1>

val tripleOdds: PartialFunction[Int, Int] = {

case x if (x % 2) != 0 => x \* 3//Case statement

}

tripleOdds: PartialFunction[Int,Int] = <function1>

val whatToDo = doubleEvens orElse tripleOdds

whatToDo: PartialFunction[Int,Int] = <function1>

* + Từ khóa “andThen” để kết nối các funtion.

val doubleEvens: PartialFunction[Int, Int] = {

case x if (x % 2) == 0 => x \* 2

}

val tripleOdds: PartialFunction[Int, Int] = {

case x if (x % 2) != 0 => x \* 3

}

val printEven: PartialFunction[Int, String] = {

case x if (x % 2) == 0 => "Even"

}

val printOdd: PartialFunction[Int, String] = {

case x if (x % 2) != 0 => "Odd"

}

val whatToDo = doubleEvens orElse tripleOdds andThen (printEven orElse printOdd)

* 1. **Partially Applied Functions**
  + **Placeholder Syntax**

Placeholder (\_) có thể được dùng vào 2 mục đích sau :

* Thay thể cho những function type được định nghĩa rõ ràng
* Thay thế cho những parameter được sử dụng không quá 1 lần.
* Có lẽ vẫn hơi khó hiểu,ta cùng xem ví dụ :

scala> val nhanDoi: Int => Int = \_ \* 2  
nhanDoi: Int => Int = <function1>

scala> nhanDoi(5)  
res7: Int = 10

* + **Partially Applied Functions**

Một partially Applied function là một hàm không có một vài tham số hoặc không có tham số nào để tạo ra hàm khác.

Ví dụ một partially applied function không có đối số.

def sum(a: Int, b: Int, c: Int) = a + b + c  
//sum: sum[](val a: Int,val b: Int,val c: Int) => Int  
val sum3 = sum \_  
//sum3: (Int, Int, Int) => Int = <function3>  
sum3(1, 9, 7)  
//res8: Int = 17  
sum(4, 5, 6)  
//res8: Int = 17

Ví dụ partial applied function thay thế một số tham số:

def sum(a: Int, b: Int, c: Int) = a + b + c  
//sum: sum[](val a: Int,val b: Int,val c: Int) => Int  
val sumC = sum(1, 10, \_: Int)  
//sumC: Int => Int = <function1>  
sumC(4)  
//res9: Int = 15  
sum(4, 5, 6)  
//res10: Int = 15

* + **Currying Function**

Currying là một kỹ thuật để chuyển đổi chức năng với nhiều tham số vào nhiều chức năng có một tham số.

def multiply(x: Int, y: Int) = x \* y  
//multiply: multiply[](val x: Int,val y: Int) => Int  
(multiply \_).isInstanceOf[Function2[\_, \_, \_]]  
//res10: Boolean = true  
val multiplyCurried = (multiply \_).curried  
//multiplyCurried: Int => (Int => Int) = <function1>  
multiply(4, 5)  
//res11: Int = 20  
multiplyCurried(3)(2)  
//res12: Int = 6  
val multiplyCurriedFour=multiplyCurried(4)  
//multiplyCurriedFour: Int => Int = <function1>  
multiplyCurriedFour(2)  
//res13: Int = 8  
multiplyCurriedFour(4)  
//res13: Int = 16

Currying cho phép bạn tạo ra các phiên bản riêng của chức năng tổng quát:

def customFilter(f: Int => Boolean)(xs: List[Int]) = {  
 xs filter f  
}  
//customFilter: customFilter[](val f: Int => Boolean)(val xs: List[Int]) => List[Int]  
def onlyEven(x: Int) = x % 2 == 0  
onlyEven: onlyEven[](val x: Int) => Boolean  
  
val xs = **List**(12, 11, 5, 20, 3, 13, 2)  
//xs: List[Int] = List(12, 11, 5, 20, 3, 13, 2)  
  
customFilter(onlyEven)(xs)  
//res0: List[Int] = List(12, 20, 2)  
  
val onlyEvenFilter = customFilter(onlyEven) \_  
//onlyEvenFilter: List[Int] => List[Int] = <function1>  
onlyEvenFilter(xs)  
//res1: List[Int] = List(12, 20, 2)

* 1. **Cloure**

Closure là 1 funtion mà giá trị trả về của nó thay đổi bởi 1 hoặc nhiều variables được định nghĩa ở bên ngoài function  
ví dụ :  
đây đơn thuần là 1 function literal:

val multiplier = (i:Int) => i \* 10  
//multiplier: Int => Int = <function1>

Ta thay 10 bằng 1 biến được khai báo trước đó,bên ngoài function :

val multiplier = (i:Int) => i \* factor  
//multiplier: Int => Int = <function1>

giá trị trả về bây giờ phụ thuộc vào giá trị của factor nên nó là 1 Closure.

1. Strings
2. Arrays
3. Class – Object
   1. Class

Class trong Scala có ý nghĩa giống với class trong Java.

Class trong Scala là thành phần static có thể được khởi tạo vào nhiều đối tượng tại runtime.

Vd:

class Point(xc: Int, yc: Int) {  
 var **x**: Int = xc  
 var **y**: Int = yc  
 def move(dx: Int, dy: Int) {  
 **x** = **x** + dx  
 **y** = **y** + dy  
 }  
 override def toString(): String = "(" + **x** + ", " + **y** + ")";  
}

Class “Point” có 2 biến x và y, có 2 phương thức “move” và “toString”

Phương thức “move” có 2 tham số nhưng không trả về giá trị nào (ngầm trả về kiểu Unit). Phương thức “toString” không có tham số, trả về String kế thức phương thức “toString” với từ khóa “override”

Khi khai báo class trong Scala thì bạn có thể viết tất cả parameter của constructor vào câu khai báo và bên trong thì không cần viết lại constructor nữa. Với mỗi class về cơ bản chỉ được có một constructor duy nhất, Mặc dù về mặt cú pháp bạn vẫn có thể định nghĩa nhiều constructor nhưng thực tế sử dụng thì chỉ trong những trường hợp rất đặc biệt. Thêm nữa, constructor đầu tiên có một ý nghĩa đặc biệt hơn các constructor khác (nếu được định nghĩa) và được gọi là **primary constructor**.

Class trong Scala được khởi tạo bởi từ khóa “new”:

object Main {  
 def main(args: Array[String]) {  
 val pt = new Point(1, 2)  
 println(pt)  
 pt.move(10, 10)  
 println(pt)  
 }  
}

Chương trình định nghĩa class “Main” thực thi dưới dạng một đối tượng top-level singleton.

Parameter được định nghĩa với từ khóa val hoặc var như cách 2 là công khai với phạm vi ngoài class, và tất nhiên cả với các method bên trong class nữa.

Vd:

class Saiyan(val name: String, val age: Int) {  
 def +(p: Saiyan): Saiyan = {  
 new Saiyan(name + p.name, age + p.age)  
 }  
}  
val goku = new Saiyan("Goku", 100)  
val vegeta = new Saiyan("Vegeta",100)  
val vegeto = goku + vegeta  
vegeto.name  
// res0: String = GokuVegeta  
vegeto.age  
// res1: Int = 200

**Property**

Property là một phần không thể thiếu của class. Property có thể là private hoặc protected. Nếu không viết gì đằng trước thì property là public.  
Trước property trong class có thể dùng từ khóa val cho trường hợp không thể thay đổi giá trị, và var cho trường hợp có thể thay đổi giá trị.

**Method**

Method là những function thuộc về class ta đang xét. Mỗi loại dữ liệu định nghĩa như một class và có nhiều function đi kèm class đó.

Một số cách định nghĩa method:

**class Person**(**val** name: String, **val** age: Int) {  
 **def** +(p: **Person**): **Person** = **new** Person(name + p.name, age + p.age)  
}  
  
**class Person**(**val** name: String, **val** age: Int) {  
 **def** +(p: **Person**): **Person** = {  
 **new** Person(name + p.name, age + p.age)  
 }  
}

* + Trước mỗi method có thể được thêm private, protected, private[this], protected[package\_name]
* private: method chỉ có thể truy cập từ bên trong class.
* protected: method chỉ có thể truy cập từ bên trong class và các class con kế thừa nó.
* private[this]: method chỉ có thể truy cập từ bên trong object (cái này sẽ nói sau trong phần object).
* protected[package\_name]: method có thể truy cập từ bên trong package có tên là package\_name.
* Method không có gì ở đầu thì mặc định là public, tức là có thể truy cập ở bất cứ đâu.

**Curry method**

Method trong Scala có thể có nhiều dãy parameter, mỗi dãy được viết trong ngoặc đơn (). Method kiểu này gọi là **curry method**, là một kỹ thuật thông dụng trong các ngôn ngữ lập trình hàm và cần thiết cho phần **implicit parameter**. Method “union” dưới đây là một curry method.

**class Saiyan**(**val** *name*: *String*, **val** *age*: Int) {  
 **def** +(*p*: **Saiyan**): **Saiyan** = {  
 **new** Saiyan(*name* + *p*.*name*, *age* + *p*.*age*)  
 }  
}

//defined class Saiyan

**val** goku = **new** Saiyan(**"Goku"**, **100**)  
//goku: Saiyan = Saiyan@559507b5  
**val** vegeta = **new** Saiyan(**"Vegeta"**,**100**)  
//vegeta: Saiyan = Saiyan@3a75489c  
  
**class Fusion** {  
 **def** union(*left*: **Saiyan**)(*right*: **Saiyan**): **Saiyan** = *left* + *right*}  
//defined class Fusion  
  
**var** magic = **new** Fusion()  
// magic: Fusion = Fusion@447a020  
**val** vegeto = magic.union(goku)(vegeta)  
// magic: Fusion = Fusion@447a020  
magic.union(goku) \_  
// res3: Saiyan => Saiyan = <function1>

Ở dòng cuối cùng, khi chỉ gọi một dãy parameter thì curry method sẽ trả về một function.

**Kế thừa**

Tính chất quan trọng nhất của class là tính kế thừa. Kế thừa trong Java có 2 loại: kế thừa class cha (từ khóa extends) và kế thừa interface (từ khóa implements). Mỗi class chỉ có thể kế thừa duy nhất một class cha duy nhất.  
Trong Scala tính kế thừa đã được phát triển thêm. Một class có thể kế thừa nhiều trait khác nhau. Trait sẽ được nói đến trong một phần khác. Ở đây chúng ta sẽ chỉ nói đến kế thừa class thông dụng. Vd:

**class Saiyan**(**val** *strength*: Int) {  
 **def** recover(): Unit = {  
 **println**(*strength*\***10**)  
 }  
}  
//defined class Saiyan  
  
**val** kakarot = **new** Saiyan(**10**)  
//kakarot: Saiyan = Saiyan@2706f37b  
  
kakarot.recover()  
// 100  
//res0: Unit = ()  
  
**class SuperSaiyan**(*strength*: Int) **extends** Saiyan(*strength*) {  
 **override def** recover(): Unit = {  
 **println**(*strength*\***100**)  
 }  
}  
//defined class SuperSaiyan  
  
**val** goku = **new** SuperSaiyan(**10**)  
//goku: SuperSaiyan = SuperSaiyan@5860c0fa  
  
goku.recover()  
// 1000  
//res1: Unit = ()

* 1. **Object**

Trong Scala, tất cả mọi giá trị đều là một object, tất cả mọi method đều trực thuộc một object nào đó.   
Khi trong Java bạn thường thấy các property và method được gán nhãn static trong một class thì trong Scala việc đó trở nên không cần thiết. Trong Scala chúng ta có thể dùng từ khóa object để định nghĩa một khoảng không gian với một tên gọi nhất định giống như là một "singleton object". Tất cả những thứ nằm trong phần không gian nói trên cũng là một object, vì vậy hoàn toàn có thể định nghĩa method và property ở trong này.

Một Object được thể hiện bằng từ khóa “object” có đặc điểm sau:

* Có thể đặt các utility method hoặc global property trong này (giống khoảng bên trong static ở Java).
* Có thể đặt các factory method cho object.
* Sử dụng như là Singleton Pattern (Cái này là một design pattern cơ bản nhé).

Cách khai báo object:

**object obj\_name extends** class\_name **with** trait\_name1 **with** trait\_name2 {

}

Scala có một object tên là **Predef** được định nghĩa là import sẵn. Ví dụ hàm “println()” thực ra là một method của object **Predef**.

Sau đây mình sẽ đưa ra một ví dụ tạo object trùng tên với một class:

**class Saiyan**(**val** name: String, **val** age: Int)  
  
**object Saiyan** {  
 **def** apply(name: String, age: Int): **Saiyan** = **new** Saiyan(name, age)  
}

apply là một hàm đặc biệt. Sau khi định nghĩa như trên thì 2 dòng dưới đây là như nhau:

Saiyan.apply("goku", 100)  
Saiyan("goku",100)

So với cách gọi trực tiếp new Saiyan("goku", 100) thì viết thông qua object thế này có 2 cái lợi:

* + - * Ẩn được nội dung cụ thể của class Saiyan
      * Có thể trả về một instance của một subclass của Saiyan.

Cách viết như trên còn có thể viết gọn nữa như sau:

**case class Saiyan**(name: String, age: Int)  
*//defined class Saiyan*

**Case Class**

Case class là một đối tượng có hai cách dùng. Cách dùng thứ nhất mình sẽ trình bày sau đây và cách dùng thứ hai là trong **Pattern Matching** mà chúng ta sẽ gặp trong một phần khác.

***Case class*** mang bên trong mình một class cùng tên với các constructor parameter *đều là công khai (public)*. Ngoài ra mang thêm *những method cơ bản của một object*.

Thế nào là *những method cơ bản của một object* ?

* equals()
* hashCode()
* toString()

**case class Saiyan**(name: String, age: Int)  
*//defined class Saiyan  
Saiyan*("goku",100).equals(*Saiyan*("goku",100)) *// true  
//res0: Boolean = true*

**Companion Object**

Object trùng tên với class như bên trên được gọi là **Companion Object**. Companion Object khác biệt với Object bình thường ở chỗ: có thể truy cập những property hay method là *private.*

**class Saiyan**(**val** name: String, **private val** age: Int)  
*//defined class Saiyan***object Saiyan** {*//defined module Saiyan* **def** apply(name: String, age: Int): **Saiyan** = {  
 **val** s = **new** Saiyan(name, age)  
 *println*(s.age) *// OK* **return** s  
 }  
}

Nếu object mang một cái tên khác, nó sẽ không còn được coi là companion object và đoạn code trên sẽ ko chạy.

Đến đây có thể bạn sẽ nghĩ, nếu đến *private* của class mà Companion Object còn đọc được thì có gì mà Companion Object không đọc được nữa đâu ?

Câu trả lời là *private[this]*. Nếu gắn *private[this]* vào trước một *property/method* thì nó sẽ chỉ được nhìn thấy duy nhất bên trong class đó.

1. Trait

Trait trong Scala có những điểm khác biệt chủ yếu với class như sau:

+ Một Trait hoặc một class có thể kế thừa nhiều trait khác nhau (tính năng này gọi là **Mixin** hoặc **Multiple inheritance**)

+ Không thể khởi tạo trực tiếp instance từ Trait.

+ Không thể nhận parameter (constructor parameter) như class.

**Multible Inheritance**

Vd:

**trait** Saiyan  
**trait** Namek  
**class Human  
class** God{}  
  
*// OK***class SuperHero extends** Human **with** Saiyan **with** Namek  
  
*// Compile error***class SemiGod extends** Human **with** God

một class không thể kế thừa 2 class khác cùng một lúc. Tuy nhiên với trait thì không có giới hạn.

**Không thể tạo instance trực tiếp**

**trait** Saiyan  
**object** {  
 **val** bardock = **new** Saiyan  
 *// Compile error  
 // trait Saiyan is abstract; cannot be instantiated*}

Khi gặp trường hợp thế này thì bạn có 2 lựa chọn:

* Tạo một class kế thừa trait nói trên và tạo instance của class đó
* Tạo instance thông qua việc thêm implement

**trait** Saiyan  
**class SaiyanSoldier extends** Saiyan  
  
**object SaiyanArmy** {  
 **val** *bardock* = **new** SaiyanSoldier *// OK* **val** *kingVegeta* = **new** Saiyan {} *// OK*}

**Không thể nhận constructor như một class**

Vì không thể tạo instance nên Trait cũng không có khả năng nhận constructor parameter như class.

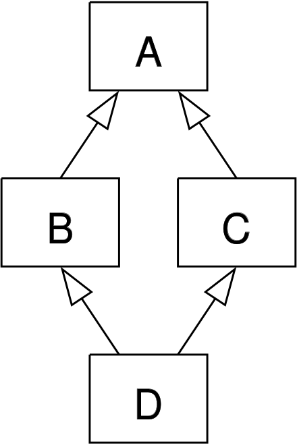
**trait** Saiyan(name: String) {}  
*// Compile error  
// traits or objects may not have parameters*

Để giải quyết tình huống này cũng có 2 cách tương tự như trên:

* Tạo class kế thừa và gọi new với class đó
* truyền thẳng paramter vào implement của trait

**trait** Saiyan {  
 **val** name: String  
}  
**class SaiyanSoldier**(**val** name: String) **extends** Saiyan  
  
**object SaiyanArmy** {  
 **val** *bardock* = **new** SaiyanSoldier("bardock")  
 **val** *kingVegeta* = **new** Saiyan { **val** *name* = "kingVegeta" }  
}

**Diamond Problem**

**Diamond Problem** là một danh từ khá nổi tiếng chỉ về một vấn đề trong cấu trúc kế thừa của lập trình hướng đối tượng. Với lý thuyết về Trait bên trên, bạn thử tưởng tượng nếu 2 trait cùng kế thừa một trait khởi điểm và override cùng một method, vậy khi class mới kế thừa 2 trait nói trên thì chuyện gì sẽ xảy ra ?

**trait** A {  
 **def** greet(): Unit  
}  
  
**trait** B **extends** A {  
 **def** greet(): Unit = *println*("Hello!")  
}  
  
**trait** C **extends** A {  
 **def** greet(): Unit = *println*("Hi!")  
}  
  
**class D extends** B **with** C  
*// Copile error  
/\*\*  
error: class D inherits conflicting members:  
 method greet in trait B of type ()Unit and  
 method greet in trait C of type ()Unit  
(Note: this can be resolved by declaring an override in class D.)  
 class D extends B with C  
 \*/*

Cách giải quyết y như thông báo lỗi đã gợi ý, chúng ta sẽ override method trong class D. Khi overide bạn có thể gọi lại method từ trait nào đã kế thừa tùy thích hoặc thậm chí gọi cả 2 luôn cũng được.

**class D extends** B **with** C {  
 **override def** greet(): Unit = {  
 **super**[B].greet()  
 **super**[C].greet()  
 }  
}

**Stackable Trait**

Giờ mình sẽ nhắc đến một trường hợp đặc biệt hơn. Nếu định nghĩa hàm greet trong cả trait B và C đều có từ khóa override thì **thứ tự kế thừa sau cùng** trong câu khai báo class sẽ quyết định hàm nào được sử dụng. Điều này gọi là **Linearization**. Các trait ở trường hợp này gọi là **Stackable Trait**.

**trait** A {  
 **def** greet(): Unit = *print*(" Borned! ")  
}  
  
**trait** B **extends** A {  
 **super**.greet()  
 **override def** greet(): Unit = *println*("Hello!")  
}  
  
**trait** C **extends** A {  
 **super**.greet()  
 **override def** greet(): Unit = *println*("Hi!")  
}  
  
**class D extends** B **with** C  
**class E extends** C **with** B  
(**new** D).greet()  
*// Borned!  
// Hello  
// Hi*(**new** E).greet()  
*// Borned!  
// Hi  
// Hello*

Ở trường hợp trên, nếu method greet ở trait A không được định nghĩ thì bạn không thể gọi super.greet(). Khi này định nghĩa cho greet sẽ cần thêm từ khóa abstract.

**trait** A {  
 **def** greet(): Unit  
}  
  
**trait** B **extends** A {  
 **abstract override def** greet(): Unit = {  
 **super**.greet()  
 **override def** greet(): Unit = *println*("Hello!")  
 }  
}

Và với kiểu khai báo trên thì bạn không thể tạo ra một class kế thừa trực tiếp trait B được. Lý do là, method greet chưa được định nghĩa? Muốn thoát khỏi tình huống này thì cần phải implement method greet thông qua một trait khác và kế thừa trait đó

**trait** C **extends** A {  
 **def** greet(): Unit = *println*("Hello!")  
}  
  
*// Compile error  
// class D needs to be a mixin, since method greet in trait B of type ()Unit is marked `abstract' and `override', but no concrete implementation could be found in a base***class D extends** B  
  
*// Ok***class ClassB extends** C **with** B

**Seft Types**

**Self types** cho phép định nghĩa trước một trait trừu tượng và thêm implement về sau. **Self types** thường được dùng khi muốn ứng dụng **Dependency Injection** hay gọi tắt là **DI**. Mỗi khi nào bạn nhìn thấy kiểu self types thế này thì có thể chuẩn bị tinh thần rằng "chuẩn bị có DI xảy ra đây".

**Vd:**

**trait** Saiyan {  
 **def** kamehameha(): Unit  
}  
*//defined trait Saiyan***trait** SuperSaiyan {  
 self: Saiyan =>  
  
 **def** damage(): Unit = kamehameha()  
}  
*//defined trait SuperSaiyan***trait** SaiyanSoldier **extends** Saiyan {  
 **def** kamehameha(): Unit = *println*("KaaaaMeeeeHaaaMeeeeHaaaa!")  
}  
*//defined trait SaiyanSoldier***val** vegeta = **new** SuperSaiyan **with** SaiyanSoldier  
*//vegeta: SuperSaiyan with SaiyanSoldier = $anon$1@6587414c*vegeta.damage()  
*//KaaaaMeeeeHaaaMeeeeHaaaa!  
//res0: Unit = ()*

Khi định nghĩa SuperSaiyan với từ khóa self: Saiyan như trên, trait SuperSaiyan có thể sử dụng method kamehameha trong khi bản thân không extends trực tiếp *Saiyan*. Khi tạo một trait SaiyanSoldier để implement thì có thể tạo được instance của SuperSaiyan.   
Tạo sao lại phải tạo trait để implement thì lý do khá giống với phần **abstract override** bên trên, Scala không cho phép tạo instance với trait có method chưa được implement.

Vậy **self types** khác với extends trực tiếp thế nào? SuperSaiyanGod dưới đây có hành vi gì khác với SuperSaiyan?

**trait** Saiyan {  
 **def** kamehameha(): Unit  
}  
*//defined trait Saiyan***trait** SuperSaiyanGod **extends** Saiyan {  
 **def** damage(): Unit = kamehameha()  
}  
*//defined trait SuperSaiyanGod*

Câu trả lời là khi khởi tạo instance và chỉ rõ kiểu *SuperSaiyan* hay *SuperSaiyanGod* thì cách nhìn của instance đó với method gốc *kamehameha*() sẽ khác nhau.

**val** vegeta: SuperSaiyan = **new** SuperSaiyan **with** SaiyanSoldier  
vegeta.kamehameha()

*// error: value kamehameha is not a member of SuperSaiyan*

**val** goku: SuperSaiyanGod = **new** SuperSaiyanGod **with** SaiyanSoldier  
goku.kamehameha()

*// KaaaaMeeeeHaaaMeeeeHaaaa*

1. Pattern Matching

Case class và matching là hai thành phần rất hữu dụng trong scala. Ở phần này chúng ta sẽ tìm hiểu về chúng.

**Case class**

Ví dụ khai báo các case class:

**abstract class Expr***//defined class Expr***case class Var**(name: String) **extends** Expr  
*//defined class Var***case class Number**(num: Double) **extends** Expr  
*//defined class Number***case class UnOp**(operator: String, arg: **Expr**) **extends** Expr  
*//defined class UnOp***case class BinOp**(operator: String, left: **Expr**, right: **Expr**) **extends** Expr  
*//defined class UnOp*

Việc khai báo case class rất đơn giản. Nó giống như khai báo class nhưng thêm tiền tố **case** vào đầu. Việc khởi tạo case class cũng không phải khởi tạo với **new** mà có thể dẽ dàng thực hiện như sau:

**val** v = Var("x")

*//v: Var = Var(x)***val** op = BinOp("+", Number(1), v)

*//op: BinOp = BinOp(+,Number(1.0),Var(x))*

Các parameter trong case class còn mặc định được gán là val.

**Pattern Matching**

Xét hàmsimplifyTop:

**def** simplifyTop(expr: **Expr**): **Expr** = expr **match** {  
 **case** *UnOp*("-", *UnOp*("-", e)) => e *// Double negation* **case** *BinOp*("+", e, *Number*(0)) => e *// Adding zero* **case** *BinOp*("\*", e, *Number*(1)) => e *// Multiplying by one* **case** \_ => expr  
}  
*//simplifyTop: simplifyTop[](val expr: Expr) => Expr*

simplifyTop(*UnOp*("-", *UnOp*("-", *Var*("x"))))  
*//res0: Expr = Var(x)*

Như vậy **match** trong scala giống với **switch** trong java. Việc sử dụng case \_ => expr giống với default trong java.

**Kinds Pattern**

Để sử dụng **match** một cách tốt nhất ta nên hiểu các pattern của nó. Dưới đây là các pattern quan trọng:

* **Wildcard patterns (mẫu ký tự đại diện)**

Trong scala wildcard pattern được sử dụng là (\_) nó đại diện cho tất cả các object. Nó được sử dụng như default trong java.   
Ta xem ví dụ sau:

expr **match** {  
 **case** *BinOp*(op, left, right) =>  
 *println*(expr +"is a binary operation")  
 **case** \_ =>  
}

Ở đây ta thấy rằng trong trường hợp expr không phải là kiểu BinOp thì sẽ không sử lý gì cả. Nếu trong trường hợp bạn không cần đến dữ liệu op, left, right trong BinOp(op, left, right) thì chúng ta cũng có thể viêt bằng cách khác như sau:

expr **match** {  
 **case** *BinOp*(\_, \_, \_) => *println*(expr +"is a binary operation")  
 **case** \_ => *println*("It's something else")  
}

* **Constant Pattern**

Sử dụng Constant patterns khi muốn chỉ rõ một giá trị nào đó ví dụ như 5, true, "hello"…

**def** describe(x: **Any**) = x **match** {  
 **case** 5 => "five"  
 **case true** => "truth"  
 **case** "hello" => "hi!"  
 **case** *Nil* => "the empty list"  
 **case** \_ => "something else"  
}

* **Variable Pattern**

Một Variable patterns sẽ nhận 1 giá trị bất kỳ giống như wildcard. Nhưng khác biệt ở đây là ta có thể sử dụng giá trị đó.

expr **match** {  
 **case** 0 => "zero"  
 **case** somethingElse => "not zero: "+ somethingElse  
}

Khi ta sử dụng đồng thời cả Variable patterns và Wildcard patterns thì việc gì sẽ sảy ra?

**import Math**.{*E*, *PI*}  
  
**val** pi = **Math**.*PI //pi: Double = 3.141592653589793  
  
E* **match** {  
 **case** pi => "strange math? E = "+ pi  
}  
*//res9: java.lang.String = strange math? Pi = 2.7182818...*

pi ở trong case pi là một Variable patterns vì vậy nó sẽ nhận giá trị = E = 2.7182818...

Nếu bạn thực hiện code như sau:

*E* **match** {  
 **case** pi => "strange math? Pi = "+ pi  
 **case** \_ => "OK"  
}  
<**console**>:10: error: unreachable code  
 case \_ => "OK"

Có lỗi sảy ra vì pi ở đây là 1 Variable patterns nó sẽ nhận tất cả các giá trị khi đó case \_ sẽ trở nên vô nghĩa.

vậy làm sao để để so sánh E với pi (Constant patterns) được khai báo ở val pi = Math.Pi? Rất đơn giản ta chỉ cần thêm ` vào trước và sau pi.

*E* **match** {  
 **case** `pi` => "strange math? Pi = "+ pi  
 **case** \_ => "OK"  
}  
*//res13: java.lang.String = OK*

* **Constructor pattern**

Constructor thực sự làm cho pattern matching chở nên mạnh mẽ. Một constructor pattern trông giống như là "BinOp("+", e, Number(0))". Nó bao gồm kiểu là "BinOp" và các patterns nhận giá trị là "+", e, Number(0)

expr **match** {  
 **case** BinOp("+", e, Number(0)) => *println*("a deep match")  
 **case** \_ =>  
}

* **Sequence pattern**

Chúng ta xem ví dụ với việc match 1 list có đội dài là 3 và có phần tử đầu tiên là 0

expr **match** {  
 **case** List(0, \_, \_) => *println*("found it")  
 **case** \_ =>  
}

Khi ban muốn match với 1 list không giới hạn độ dài ban có thể sử dụng (\_\*)

expr **match** {  
 **case** List(0, \_\*) => *println*("found it")  
 **case** \_ =>  
}

* **Tuple pattern**

Cũng như List hoặc Array việc match tuple khá đơn giản.

**def** tupleDemo(expr: **Any**) =  
 expr **match** {  
 **case** (a, b, c) => *println*("matched "+ a + b + c)  
 **case** \_ =>  
 }

* **Typed pattern**

Chúng ta cũng có thể match theo type.

**def** generalSize(x: **Any**) = x **match** {  
 **case** s: String => s.length  
 **case** m: Map[\_, \_] => m.size  
 **case** \_ => -1  
}  
generalSize("abc")  
*//res14: Int = 3*generalSize(*Map*(1 -> 'a', 2 -> 'b'))  
*//res15: Int = 2*generalSize(**Math**.Pi)  
*//res16: Int = -1*

**Variable Binding**

Khi muốn tạo rằng buộc khi match ta sủ dung thêm ký tự @ để tạo rằng buộc.

expr **match** {  
 **case** UnOp("abs", e @ UnOp("abs", \_)) => e  
 **case** \_ =>  
}

Ở đây bắt buộc e phải có kiểu là UnOp và parameter đầu tiên phải là "abs" Khi muốn tạo thêm rằng buộc ta có thể sử dụng toán tử if ngay trong câu lệnh case.

**def** simplifyAdd(e: Expr) = e **match** {  
 **case** BinOp("+", x, y) **if** x == y => BinOp("\*", x, Number(2))  
 **case** \_ => e  
}

Ở ví dụ trên ta thêm rằng buộc chỉ chấp nhận BinOp có x = y

* **Sealed Classes**

Khi bạn thực hiện match một đối tượng case class thuộc 1 lớp nào đó mà không thực hiện hết việc match bao phủ hết các trường hợp của nó thì scala sẽ đưa ra thông báo cho chúng ta.

**sealed abstract class Expr  
case class Var**(name: String) **extends** Expr  
**case class Number**(num: Double) **extends** Expr  
**case class UnOp**(operator: String, arg: **Expr**) **extends** Expr  
**case class BinOp**(operator: String,  
 left: **Expr**, right: **Expr**) **extends** Expr  
  
**def** describe(e: **Expr**): String = e **match** {  
 **case** *Number*(\_) => "a number"  
 **case** *Var*(\_) => "a variable"  
}

Khi compile đoạn code trên scala sẽ đưa ra cho ta thông báo

warning: **match** is not exhaustive!  
 missing combination UnOp  
missing combination BinOp

Để loại bỏ vấn đề này ta có 2 cách. Cách thứ 1 là bắt hết các case

**def** describe(e: Expr): String = e **match** {  
 **case** Number(\_) => "a number"  
 **case** Var(\_) => "a variable"  
 **case** \_ => **throw new** RuntimeException *// Should not happen*}

Cách thứ 2 là tắt chế độ check của scala

**def** describe(e: Expr): String = (e: *@unchecked*) **match** {  
 **case** Number(\_) => "a number"  
 **case** Var(\_) => "a variable"  
}

1. Collections

Collections là 1 cấu trúc data tập hợp của một hoặc nhiều giá trị như là arrays, lists, maps, sets, và trees. Hầu hết các ngôn ngữ lập trình đều có những cấu trúc collections của riêng mình bởi vì công việc lập trình hiện đại cần đến những cấu trúc dữ liệu như vậy.

* 1. List

Hãy tạo ra một immutable **list**, ta có thể tạo ra một list bằng cách khởi gọi function List và truyền vào đó những value trong list như những parameter.

**val** listSo = *List*(1,5,7,9)  
*//listSo: List[Int] = List(1, 5, 7, 9)***val** listHoaQua = *List*("Bưởi","Bòng","Cam","Quýt")  
*//listHoaQua: List[String] = List(Bưởi, Bòng, Cam, Quýt)***val** listLinhTinh = *List*(22,"ba b")  
*//listLinhTinh: List[Any] = List(22, ba b)***val** listRong = *List*()  
*//listRong: List[Nothing] = List()*

* 1. Set
  2. Map

1. Regular Expression
2. Exception Handing
3. Extractor
4. File I/O
5. Hướng dẫn cài đặt
6. Cài đặt môi trường
7. Cài đặt với Eclipse IDE
8. Chương trình Helloworld