

# Introduction to Scala

Kenny Zhuo Ming Lu

School of Information Technology, Nanyang Polytechnic

September 19, 2017

By the end of this lecture, you will have some basic understanding on

- Object Oriented Programming in Scala
- Functional Programming in Scala
  - Algebraic data type and pattern matching
  - Higher order functions
  - Partial functions
  - Implicit functions
  - Type classes
  - Monads
- Data parallelism in Scala
- Concurrent programming in Scala with threads and actors

A general purpose programming language that is

- Object Oriented and Functional
- with distributed programming builtin
- with a rich type system and a huge set of libraries and eco-systems
- Widely used in academic research projects (e.g. Spark was from UC Berkeley) and industries (e.g. Scalding from Twitter, Kakfa from Linkedln, Coursera Walmart, ... )

# Hello World in a script

Let's say we have a Scala file `Script.scala`

```
println("Hello world!")
```

To execute it

```
$ scala Script.scala  
Hello world!
```

# Hello World in a console application

Compiling Scala program into an application

```
object Main extends App
{
    println("Hello world!")
}
```

object is like a singleton class.

To execute it

```
$ scalac Main.scala
```

```
$ scala Main
```

```
Hello world!
```

# Object Oriented Programming in Scala

It is very similar to Java

```
class Person(n:String,i:String) {  
    private val name:String = n  
    private val id:String    = i  
    def getName():String = name  
    def getId():String = id  
}
```

- Person is the name of the class,  
 Person(n:String,i:String) is a constructor.
- private sets the accessibility scope of the members
- val introduces a value, (immutable variable)
- def introduces a method definition.

## Class Inheritance via the extends keyword

```
class Student(n:String, i:String, g:Double)
    extends Person(n,i) {
    private var gpa = g
    def getGPA() = gpa
    def setGPA(g:Double) = {
        gpa = g
    }
}
```

- var introduces a mutable variable
- Most of the type annotations are optional and will be inferred by the type system.

# Object Oriented Programming in Scala

```
class Staff(n:String, i:String, sal:Double)
    extends Person(n,i) {
  private var salary = sal
  def getSalary() = salary
  def setSalary(sal:Double) =
  {
    salary = sal
  }
}
```



# Object Oriented Programming in Scala

Traits are like Java interfaces and abstract classes

```
trait NightOwl {  
    def stayUpLate():Unit  
}  
  
class Student(n:String, i:String, g:Double)  
    extends Person(n,i) with NightOwl {  
    private var gpa = g  
    def getGPA() = gpa  
    def setGPA(g:Double) = {  
        gpa = g  
    }  
    override def stayUpLate():Unit = {  
        println("woohoo")  
    }  
}
```

# Running our first Scala program

Scala comes with an interpreter (AKA REPL)

```
scala> :load OOP.scala
```

```
Loading OOP.scala...
```

```
defined class Person
```

```
defined trait NightOwl
```

```
defined class Student
```

```
defined class Staff
```

```
scala> val tom = new Student("Tom", "X1235", 4.0)
```

```
tom: Student = Student@601c1dfc
```

```
scala> val jerry = new Staff("Jerry", "T0001", 500000.0)
```

```
jerry: Staff = Staff@650fbe32
```

```
scala> tom.stayUpLate
```

```
woohoo
```

# Functional Programming in Scala

OOP is useful, but often less interesting, let's consider the FP features in Scala.



Image taken from <https://www.toptal.com/scala/why-should-i-learn-scala>

# Algebraic Data Type

In Scala, this is how we define a data type

```
sealed trait Exp
case class Val(v:Int) extends Exp
case class Plus(e1:Exp, e2:Exp) extends Exp

def simp(e:Exp):Exp = e match {
  case Val(v) => e
  case Plus(Val(0), e2) => e2
  case Plus(e1,e2) => Plus(simp(e1), simp(e2))
}
```

- sealed requires all extensions to the trait Exp must be declared in the same module.
- case makes a class “pattern-matchable”

# Algebraic Data Type

In Java, to achieve the above we have to use abstract class and method overriding. (For brevity, setter and getter methods are omitted.)

```
public abstract class Exp { public abstract Exp simp(); }
public class Val extends Exp {
    private int v;
    public Val(int v) { this.v = v; }
    public Exp simp() { return this; }
}
public class Plus extends Exp {
    private Exp e1; private Exp e2;
    public Plus(Exp e1, Exp e2) {
        this.e1 = e1; this.e2 = e2;
    }
    public Exp simp() {
        if (e1 instanceof Val) {
            Val v1 = (Val)e1;
            if (v1.getV() == 0) { return e2; }
        }
        return new Plus(e1.simp(), e2.simp());
    }
}
```

Algebraic data type and pattern matching are useful in case of handling recursive/nested data structures, e.g. a linked list.

```
sealed trait List[A]
case class Nil[A]() extends List[A]
case class Cons[A](x:A, xs:List[A]) extends List[A]

val x = Cons(1,Cons(2,Nil()))

def length[A](l:List[A]):Int = l match {
  case Nil() => 0
  case Cons(_,xs) => 1 + length(xs)
}
```

## Observation

- A is a type variable (AKA generics)
- Parametric polymorphism
- Nil() takes no argument, it's always (almost) singleton.

## Mixing subtyping with parametric polymorphism

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](x:A, xs:List[A]) extends List[A]

val x = Cons(1,Cons(2,Nil))

def length[A](l:List[A]):Int = l match {
  case Nil => 0
  case Cons(_,xs) => 1 + length(xs)
}
```

- case object introduces a singleton value constructor.
- + indicates that A is a covariant type parameter, hence `List[Nothing] <: List[A]` for any A .
- Nothing is the bottom type in the sub-typing hierarchy.

# List

List is a builtin data type comes with Scala library

```
val x = List(1,2)

def length[A](l:List[A]):Int = l match {
  case Nil => 0
  case (_::xs) => 1 + length(xs)
}
```

which is equivalent to the following Haskell program

```
x = [1,2]
length :: [a] -> Int
length l = case l of
  [] -> 0
  (_:xs) -> 1 + length xs
```



```
val nats:List[Int] = List(1,2,3,4)
val even_plus_1:List[Int] = for (nat <- nats
                                if nat % 2 == 0
                                ) yield (nat+1)
```

- for ... yield denotes a sequence comprehension.
- C.f. set builder notation in math,

$$\{nat + 1 \mid nat \in nats \wedge nat \bmod 2 = 0\}$$

# For loop vs comprehension

Note that `for ... yield` denotes a sequence comprehension.

```
val nats:List[Int] = List(1,2,3,4)
val even_plus_1:List[Int] = for (nat <- nats
                                if nat % 2 == 0
                                ) yield (nat+1)
```

While `for` introduces a `for` loop.

```
for (i <- 1 to 10) { println(i) }
```

prints the numbers 1 to 10.

```
for (nat <- nats if nat % 2 == 0) { println(nat) }
```

prints the numbers 2 and 4.

## Length

```
val l1 = List(1,2,3)
val len = l1.length // 3
```

## Concatenation

```
val l2 = List(-1,-2,-3)
val l3 = l1 ++ l2 // List(1,2,3,-1,-2,-3)
```

## Reverse

```
val l4 = l1.reverse // List(3,2,1)
```

## min/max

```
// recall l1 = List(1,2,3)
val min = l1.min // 1
val max = l1.max // 3
```

## sorting

```
// recall l3 = List(1,2,3,-1,-2,-3)
val sl3 = l3.sortWith(_<_) // List(-3, -2, -1, 1, 2, 3)
// we will look into _<_ in a short while.
```

## sub-list

```
val sl3a = sl3.take(3) // List(-3,-2,-1)
val sl3b = sl3.drop(3) // List(1,2,3)
```

# Map and Fold

`l.map(f)` returns a new list by applying the unary function `f` to each element in `l`

```
val l = List(1,2,3,4)
val l2 = l.map(x => x * 2) // List(2,4,6,8)
```

Note that `x => x * 2` is a lambda-expression, which can be shortened as `_ * 2`. The `.` is optional.

```
val l2 = l map (_*2)
```

which is equivalent to the following Haskell program

```
l = [1,2,3,4]
l2 = map (\x -> x*2) l
```

# Map and Fold

`l.foldLeft(b)(f)` aggregate all the elements in `l` with the binary function `f` and the base value `b`.

```
// recall l = List(1,2,3,4)
val l3 = l.foldLeft(0)((x,y) => x + y)
```

In essence the above is computing

$$(((0 + 1) + 2) + 3) + 4$$

`l.foldRight(b)(f)` aggregate all the elements in `l` with the binary function `f` and the base value `b` with right associativity.

```
val l4 = l.foldRight(0)((x,y) => x + y)
// or val l4 = l.foldRight(0)(_+_)
```

In essence the above is computing

$$1 + (2 + (3 + (4 + 0)))$$

# Higher order function

The definition of map.

```
sealed trait List[+A] {  
  def map[B](f:A=>B):List[B] = this match {  
    case Nil => Nil  
    case x::xs => f(x)::xs.map(f)  
  }  
}  
case object Nil extends List[Nothing]  
case class Cons[A](x:A, xs:List[A]) extends List[A]
```

Function  $f:A \Rightarrow B$  is used as a formal argument.

Equivalently, in Haskell, we have.

```
map :: (a -> b) -> [a] -> [b]  
map f [] = []  
map f (x:xs) = (f x):(map f xs)
```

# Higher order function

## Function composition

```
val l1 = List(1,2,3)
val l2 = l1.map ((x:Int) => x*2) compose (y=>y+1))
// List(4, 6, 8)
```

$(f \text{ compose } g)(x)$  is equivalent to  $f(g(x))$

```
val l3 = l1.map ((x:Int) => x*2) andThen (y=>y+1))
// List(3, 5, 7)
```

$(f \text{ andThen } g)(x)$  is equivalent to  $g(f(x))$



# Higher order function

In Scala, functions are instances/objects of trait Function

The definitions of compose and andThen.

```
object Function { // this is like a package
  trait Function1[+A][-B] {
    def apply(x:A):B {...}
    // apply() gives us
    // the syntactic sugar form like f(x)
    def compose[C](that:C=>A):C=>B =
      (c:C) => this(that(c))
    def andThen[C](that:B=>C):A=>C =
      (a:A) => that(this(a))
  }
}
```

Both compose and andThen return functions as results.

# Functions vs methods

In Scala there is a clear distinction between functions and methods.

- Methods are introduced by `def` keyword, e.g.

```
scala> def f(x:Int) = x  
f: (x: Int)Int
```

- Functions are implementations of the `FunctionX` traits. e.g. lambda expressions are functions.

```
scala> val g = (x:Int) => x  
g: Int => Int = <function1>
```

- Similarity: both can be used as argument and results

```
scala> List(1,2).map (f) == List(1,2).map (g)  
res2: Boolean = true
```

# Functions vs methods

In Scala there is a clear distinction between functions and methods.

- Difference: methods do not possess combinators

```
scala> f compose g
<console>:10: error: missing arguments for method f;
follow this method with '_' if you want to treat it
as a partially applied function
      f compose g
        ^
```

```
scala> g compose f
res1: Int => Int = <function1>
```

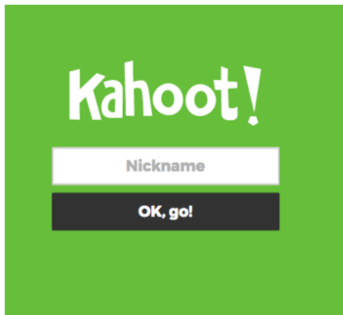
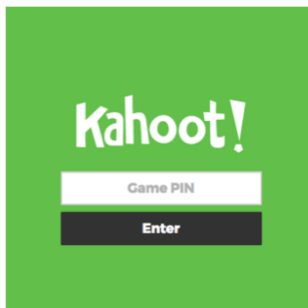
- To convert a method to a function.

```
scala> f _
res1: Int => Int = <function1>
```

## Before we take a break...

Let's play some quiz games. Open the web browser in your mobile, tablet or laptop.

- visit `http://kahoot.it`
- key in the game pin, then
- key in your name, please use real name to claim the prize.



# Before we take a break...

**Q1** Where is the Eiffel Tower located?



Next ▶

**0**

Answers



Paris



Rome



Moscow



London

# Partial functions

There are situations in which we need to deal with partial functions, naturally, we can use the `Option` type in Scala (similar to the `Maybe` type in Haskell).

```
def div(x:Int)(y:Int):Option[Int] = {  
    if (y == 0) None else Some(x/y)  
}
```

```
val xs = List(0,1,2) map (x => div(2)(x))
```

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

```
val ys = xs filter (x => x.nonEmpty)
```

```
yields List(Some(2), Some(1))
```

```
val zs = ys map ( x => x match { case Some(y) => y } )
```

```
yields List(2,1)
```

Alternatively, we may use the Scala builtin PartialFunction

```
def div(x:Int): PartialFunction[Int,Int] = {  
  case (y:Int) if y != 0 => x/y  
}  
val xs = List(0,1,2) collect ( div(2) )
```

yields List(2, 1).

Note that we have to use the collect method instead of map

```
val ys = List(0,1,2) map ( div(2) )
```

Will result in a scala.MatchError

# Partial functions

`orElse` allows us to compose partial functions which are complementing on their domains.

```
def div(x:Int): PartialFunction[Int,Int] = {  
  case (y:Int) if y != 0 => x/y  
}  
  
val mkZero: PartialFunction[Int,Int] = {  
  case (y:Int) if y == 0 => 0  
}  
  
val ys = List(0,1,2) map ( div(2) orElse mkZero )
```

yields in a `List(0, 2, 1)`



# Implicit functions

Implicit function and implicit parameter allow us to define default behaviors.

```
implicit def logger(mesg:String):Unit = {  
    println("[Log] " + mesg)  
}  
  
def fib(x:Int)(implicit log:String=>Unit):Int = {  
    x match {  
        case 0 => 0  
        case 1 => 1  
        case _ => {  
            log("computing fib(" + x +)")")  
            fib(x-1) + fib(x-2)  
        }  
    }  
}
```

```
scala> fib(5)
[Log] computing fib(5)
[Log] computing fib(4)
[Log] computing fib(3)
[Log] computing fib(2)
[Log] computing fib(2)
[Log] computing fib(3)
[Log] computing fib(2)
res7: Int = 5
scala> fib(5)( x => Unit)
res8: Int = 5
```

Scala resolves implicit parameters as follows,

- ➊ Search in current scope
  - ➊ Implicits defined in current scope
  - ➋ Implicits defined in Explicit imports
  - ➌ Implicits defined in wildcard imports
- ➋ Search for associated types in
  - ➊ Companion objects of a type
  - ➋ Implicit scope of an argument's type
  - ➌ Outer objects for nested types
  - ➍ Other dimensions

# Type classes

Type classes was introduced in Haskell to allow mixing parametric polymorphism and ad-hoc polymorphism.

## Haskell Type Class Example

```
class Eq a where
    (==) :: a -> a -> Bool

instance Eq Int where
    (==) x y = eqInt x y

instance Eq a => Eq [a] where
    (==) [] [] = True
    (==) (x:xs) (y:ys) = (x == y) && (xs == ys)
    (==) _ _ = False

check :: (Eq [a]) => [a] -> [a] -> Bool
check xs ys = xs == ys
```

# Type classes

In Scala, type classes are “encoded” using traits and implicit definitions, i.e. `implicit val`, `implicit def` and `implicit object`

```
trait Eq[A] { def eq(x:A,y:A):Boolean }
implicit val eqInt = new Eq[Int] {
  def eq(x:Int,y:Int):Boolean = x == y
}
implicit def eqList[A](implicit ev:Eq[A]) =
  new Eq[List[A]] {
    def eq(xs>List[A],ys>List[A]):Boolean =
      (xs,ys) match {
        case (Nil,Nil) => true
        case (x::xs2, y::ys2) => ev.eq(x,y) && eq(xs2,ys2)
        case (_, _) => false
      }
  }
}
```

By resolving implicit parameter the compiler builds the “evidence” *ev* AKA the dictionary.

```
def check[A](x:List[A], y:List[A])  
  (implicit ev:Eq[List[A]]) = ev.eq(x,y)
```

```
scala> check(List(1,2,3), List(1,2,3))  
res6: Boolean = true
```

The above builds the evidence `ev = eqList(eqInt)`

# Type classes

To mimick type classes' effect in OOP languages like Java, we would have to use interface and function overriding with lots of cares. (We omit the getter and setter methods for brevity.)

```
public interface Eq<T> { public boolean eq(T that); }
public class Int implements Eq<Int> {
    private int val;
    public Int(int x) { val = x; }
    public boolean eq(Int that) {
        return (val == that.getVal());
    }
}

public class List<T extends Eq<T>> implements Eq<List<T>> {
    private Node<T> pHead;
    public List() { pHead = null;}
    public void insert(T v) {
        Node<T> n = new Node<T>(v);
        n.setNext(pHead);
        pHead = n;
    }
    public boolean eq(List<T> that) {
        return pHead.eq(that.getPHead());
    }
}
```

# Type classes

To mimick type classes' effect in OOP languages like Java, we would have to use interface and function overriding with lots of cares. (We omit the getter and setter methods for brevity.)

```
public class Node<T extends Eq<T>> implements Eq<Node<T>> {  
    private T val;  
    private Node<T> next;  
    public Node(T v) {  
        val = v;  
        next = null;  
    }  
    public boolean eq(Node<T> that) {  
        if (!val.eq(that.getVal())) { return false; }  
        else {  
            if ((next == null) && (that.getNext() == null)) { return true; }  
            else if ((next == null) && (that.getNext() != null)) { return false; }  
            else if ((next != null) && (that.getNext() == null)) { return false; }  
            else { return next.eq(that.getNext()); }  
        }  
    }  
}
```



To mimick type classes' effect in OOP languages like Java, we would have to use interface and function overriding with lots of cares.

Finally we can implement the check function in Java as follows,

```
public static <T extends Eq<T>> boolean check(List<T> l1, List<T> l2) {  
    return l1.eq(l2);  
}
```

For real world applications implemented in Scala, we do not define all the type classes from the scratch. We use a library project called scalaz. <https://github.com/scalaz/scalaz> which provides many useful type classes.

## A vague analogy

- Algebraic data types define abstractions of run-time data (or values), i.e. what are the possible values.
- Monads define abstractions of *computations*, i.e. what the possible computations.

## So what are monads good for?

- segregating pure computation from computation with side-effect, (i.e. i/o)
- capture program control flow
- generic data traversal
- define user-programmable states and transitions
- compose computations

# Monad example: Safe Integer arithmetics

Let's model safe arithmetic computation

```
sealed trait SafeVal[+T]  
case class Val[T](x:T) extends SafeVal[T]  
case class Err(msg:String) extends SafeVal[Nothing]  
  
type SafeInt = SafeVal[Int]
```

# Monad example: Safe Integer arithmetics

```
def add(x:SafeInt, y:SafeInt):SafeInt = x match {  
  case Err(msg) => Err(msg)  
  case Val(i) => y match {  
    case Err(msg) => Err(msg)  
    case Val(j) => Val(i + j)  
  }  
}  
  
def div(x:SafeInt, y:SafeInt):SafeInt = x match {  
  case Err(msg) => Err(msg)  
  case Val(i) => y match {  
    case Err(msg) => Err(msg)  
    case Val(j) => {  
      if (j == 0) {Err("div by 0")} else {Val (i/j)}  
    }  
  }  
}
```

# Monad example: Safe Integer arithmetics

We spot many similar code structure, let's rewrite them in terms of some higher order combinators `pnt` and `bnd`.

```
def pnt[A](x: => A): SafeVal[A] = Val(x)
def bnd[A,B](sa: SafeVal[A])
  (f: A => SafeVal[B]): SafeVal[B] = sa match {
  case Err(msg) => Err(msg)
  case Val(a)   => f(a)
}
```

- Note that `x:=> A` defines a *lazy* formal parameter `x` of type `A`.
- `pnt` “lifts” / “injects” a value into the `SafeVal` object.
- `bnd` “composes” the previous `SafeVal` with the subsequent computation.

# Monad example: Safe Integer arithmetics

```
def add(x:SafeInt, y:SafeInt):SafeInt = {  
  bnd(x)(x1 => {  
    bnd(y)(y1 => pnt(x1+y1))  
  })  
}  
  
def div(x:SafeInt, y:SafeInt):SafeInt = {  
  bnd(x)( x1 => {  
    bnd(y)( y1 => {  
      if (y1 == 0) { Err("div by 0") }  
      else { pnt(x1 / y1) }  
    })  
  })  
}
```

# Monad example: Safe Integer arithmetics

If we use type classes `Monad` and `MonadPlus` provided by `scalaz`, we can make it even more concise.

```
implicit object safeValMonadPlus
  extends MonadPlus[SafeVal] {
  override def point[A](a: => A): SafeVal[A] = Val(a)
  override def bind[A, B](sa: SafeVal[A])
    (f: A => SafeVal[B]): SafeVal[B] = sa match {
    case Err(msg) => Err(msg)
    case Val(a)   => f(a)
  }
  ...
}
```



# Monad example: Safe Integer arithmetics

```
def add(x:SafeInt,y:SafeInt):SafeInt = {  
  for ( x1 <- x  
    ; y1 <- y  
    ) yield (x1 + y1)  
}  
  
def div(x:SafeInt,y:SafeInt):SafeInt = {  
  for ( x1 <- x  
    ; y1 <- y  
    ; if y1 != 0  
    ) yield (x1 / y1)  
}
```

Roughly speaking, for (x <- e1 ; e2) is desugared to  
bind(e1)(x => e2)

# Monad example: Matrix multiplication

Given

```
val l1 = List(1,2,3)
val l2 = List('a','b')
```

We want to define the matrix multiplication `mult`

```
scala> mult(l1,l2)
res0: List[(Int, Char)] = List((1,a), (1,b), (2,a), (2,b),
  (3,a),(3,b))
```

```
def mult[A,B](as:List[A], bs:List[B]):List[(A,B)] = {
  as flatMap ( a =>
    bs map ( b => (a,b) )
  )
}
```

Note `List(List(1),List(2)).flat` yields `List(1,2)`. `flatMap` is defined as the composition of `flat` and `map`.

# Monad example: Matrix multiplication

List is a monad,

As defined in the scalaz library

```
implicit object listMonadPlus extends MonadPlus[List] {  
  override def point[A](a: => A):List[A] = List(a)  
  override def bind[A, B](sa: List[A])  
    (f: A => List[B]): List[B] = sa flatMap f  
  ...  
}
```

What a programmer needs to write

```
def mult[A,B](as:List[A], bs:List[B]) : List[(A,B)] = for  
  ( a <- as  
  ; b <- bs  
  ) yield (a,b)
```

There many types of commonly used monads

- List monad - list traversal and aggregate computation.
- State monad - computation with explicit state passing
- Reader monad - input scanning computation
- Monadic parser combinator - LL-style top-down parsing computation

Monads are composable via monad transformer.

So you still want to see how to mimick monads in Java?

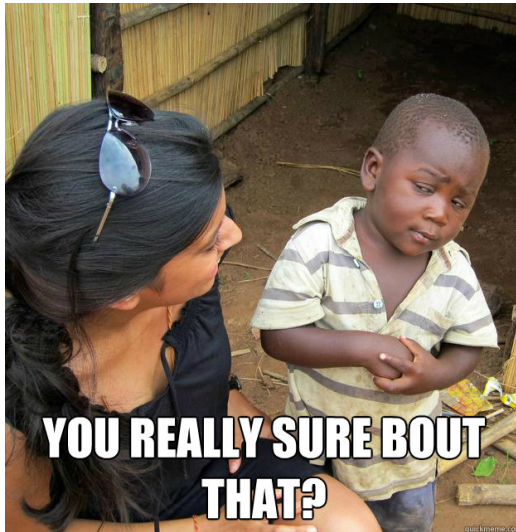


Image taken from <http://quickmeme.com>

Ok, if you are really interested, look here.

`http://github.com/luzhuomi/parsecj/tree/master/  
Parsec/src/parsec`

# Data Parallelism

Scala has built-in support of data parallelism. For instance,

```
object Fib extends App {  
  def fib(x:Int):Int = x match {  
    case 0 => 0  
    case 1 => 1  
    case n => fib(n-1) + fib(n-2)  
  }  
  println((30 to 40).toList.map(fib))  
}
```

```
$ time scala Fib  
List(832040, 1346269, 2178309, ... )  
real      0m1.621s  
user      0m1.624s  
sys       0m0.059s
```



# Data Parallelism

.par converts a sequential data collection into a parallel collection part. .seq has the opposite effect.

```
object ParFib extends App {  
  def fib(x:Int):Int = x match {  
    case 0 => 0  
    case 1 => 1  
    case n => fib(n-1) + fib(n-2)  
  }  
  println((30 to 40).toList.par.map(fib))  
}
```

```
$ time scala ParFib  
List(832040, 1346269, 2178309, ... )  
real      0m0.891s  
user      0m1.901s  
sys       0m0.061s
```

- Multi-threading
- Actors
- Async computation via Future

# Concurrent Programming

Multi-threading in Scala is similar to Java

```
object ThreadFib extends App {  
  def fib(x:Int):Int = x match {  
    case 0 => 0  
    case 1 => 1  
    case n => fib(n-1) + fib(n-2)  
  }  
  
  for (i <- 30 to 40) {  
    new Thread(new Runnable {  
      def run() {  
        println(fib(i))  
      }  
    }).start  
  }  
}
```

Actor model is a concurrent programming model.

- Actors are primitive objects with specific roles and functionalities
- Actors (objects) are coordinating with each other via message passing

Let's consider a ping pong simulation using Scala actors

- There are two actors Ping and Pong.
- Ping keeps an internal counter
  - when it receives a start message, it increments the counter and send a ping message to Pong
  - when it receives a pong message, it increments the counter.
    - if the counter  $> 99$ , it sends a stop message to pong then terminates.
    - otherwise, sends a ping message to Pong.
- Pong
  - when it receives a stop message, it terminates.
  - when it receives a ping message, it sends a pong message to Ping.

```
case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage
```

```
class Ping(pong: ActorRef) extends Actor {  
  var count = 0  
  def incrementAndPrint = {count += 1; println("ping")}  
  def receive = {  
    case StartMessage =>  
      incrementAndPrint  
      pong ! PingMessage  
    case PongMessage =>  
      incrementAndPrint  
      if (count > 99) {  
        sender ! StopMessage  
        println("ping stopped")  
        context.stop(self)  
      }  
      else { sender ! PingMessage }  
  }  
}
```

```
class Pong extends Actor {  
  def receive = {  
    case PingMessage =>  
      println("  pong")  
      sender ! PongMessage  
    case StopMessage =>  
      println("pong stopped")  
      context.stop(self)  
  }  
}
```

```
object PingPongTest extends App {  
  val system = ActorSystem("PingPongSystem")  
  val pong = system.actorOf(Props[Pong],  
    name = "pong")  
  val ping = system.actorOf(Props(new Ping(pong)),  
    name = "ping")  
  ping ! StartMessage  
}
```



The ping pong example is pretty contrived, let's consider a less contrived example in which we try to compute `fib(n)` in parallel. The idea is as follows,

- if `n` is less than 10, we compute `fib(n)` locally
- if `n` is greater than or equal to 10, we create two fib actors. The first actor computes `fib(n-1)` and the second actor computes `fib(n-2)`. The two actors can run in parallel if there is enough threads. When both actors revert with some results, we sum up the results.

Let's define the messages for communication as follows,

```
case class FibRequest(n:Int)
case class FibResult(n:Int)
case object Stop
```

# Actors

```
class FibActor extends Actor {  
  def receive = {  
    case Stop => context.stop(self)  
    case FibRequest(n) if n < 10 => sender ! FibResult(fib(n))  
    case FibRequest(n) if n >= 10 => {  
      val child1 = context.actorOf(Props[FibActor])  
      val child2 = context.actorOf(Props[FibActor])  
      val future1:Future[Any] = child1 ? FibRequest(n-1) // ‘ask’ op  
      val future2:Future[Any] = child2 ? FibRequest(n-2)  
      val dest = sender  
      val future1and2 = for ( // Future[T] is a monad  
        FibResult(v1) <- future1;  
        FibResult(v2) <- future2  
      ) yield v1+v2  
      future1and2 onComplete {  
        case Success(v) =>  
          child1 ! Stop; child2 ! Stop  
          dest ! FibResult(v) // can't use sender  
        case Failure(e) => println("failed!" + e.toString)  
      }  
    }  
  }  
}
```

```
def main(args:Array[String]):Unit = {  
  val system = ActorSystem("FibSystem")  
  val fibActor = system.actorOf(Props[FibActor])  
  val n = if (args.length > 0) args(0).toInt else 10  
  val f = fibActor ? FibRequest(n)  
  f onComplete {  
    case Success(v) => { println(v) ; fibActor ! Stop}  
    case Failure(e) => println("failed!" + e.toString)  
  }  
}
```

We've covered

- OOP in Scala
- Functional Programming in Scala which includes,
  - Algebraic data type and pattern matching
  - Higher order functions
  - Partial functions
  - Implicit functions
  - Type classes
  - Monads
- Parallel programming in Scala
- Concurrent programming in Scala with
  - threadings
  - actors

Latex source and example codes can be found here

<https://github.com/luzhuomi/learning-scala>