



Scala

~ a tragedy in two parts ~

the tragedy being we aren't using it all the time



The Scala Language

Praise for Scala



I can honestly say if someone had shown me the Programming in Scala ... I'd probably have never created Groovy.



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

More Praise for Scala



I've written production apps in Haskell, taught advanced FP and type theory, published a paper on category theory and still think that Scala is over-complicated, a bastard child of OO and FP with some XML thrown in for reasons unknown...



...Don't get me wrong, it's better than anything else you can get on the JVM...

```
object Program {  
  def main(args: Array[String]) : Unit = {  
    println("Hello World")  
  }  
}
```

```
object Program extends App {  
  println("Hello World")  
}
```

objects vs classes

optional return type

```
object Program {  
  def main(args: Array[String]) : Unit = {  
    println("Hello World")  
  }  
}
```

java without semicolons?

```
object Program extends App {  
  println("Hello World")  
}
```

means "i will return something". but what?

a trait

```
class Person(var name: String, var age: Int)
```

class definition



```
class Person(var name: String, var age: Int)
```



constructor params
and properties
(FYI vars suck)


```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }

    public int getAge() { return age; }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```
public class Person {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() { return name; }

  public int getAge() { return age; }

  public void setName(String name) {
    this.name = name;
  }

  public void setAge(int age) {
    this.age = age;
  }
}
```

boilerplate

boilerplate

boilerplate

boilerplate

boilerplate

boilerplate

```
val list1 = List(1,2,3,4)
val list2 = 1 :: 2 :: 3 :: 4 :: Nil
val list3 = 0 :: list1

list1.head
list1.tail
list1.isEmpty

list1.map(_ + 10)
list1.map(i => i + 10)

// +100s more
```

companion object

val is sorta final

crazy operators

```
val list1 = List(1,2,3,4)
val list2 = 1 :: 2 :: 3 :: 4 :: Nil
val list3 = 0 :: list1
```

prepend

but the tip of the iceberg

```
list1.head
list1.tail
list1.isEmpty
```

brackets schmackets!

```
list1.map(_ + 10)
list1.map(i => i + 10)
```

sweet sweet sugar

```
// +100s more
```

true story


```
def currentUserCredentials = {  
  ("username", "password")  
}
```

```
val (username, password) = currentUserCredentials
```


```
println(username)  
println(password)
```

returns a tuple, also return statements? pah!

tuple as a destructive assignment



```
def currentUserCredentials = {  
  ("username", "password")  
}
```



```
val (username, password) = currentUserCredentials
```

```
println(username)  
println(password)
```

```
case class Person(name: String, skills: List[String])

val attendees = List(
  Person("james", List("scala", ".net")),
  Person("will", List("workday", ".net")),
  Person("luke", List("ios", "drinking redbull")),
  Person("jonny", List("java", "workday"))
)

val coolPeople = for {
  attendee <- attendees if attendee.name.startsWith("j")
  skills <- attendee.skills if skills.contains("scala")
} yield attendee.name

coolPeople.map(println(_))
```

case class




```
case class Person(name: String, skills: List[String])
```

```
val attendees = List(  
  Person("james", List("scala", ".net")),  
  Person("will", List("workday", ".net")),  
  Person("luke", List("ios", "drinking redbull")),  
  Person("jonny", List("java", "workday"))  
)
```

```
val coolPeople = for {  
  attendee <- attendees if attendee.name.startsWith("j")  
  skills <- attendee.skills if skills.contains("scala")  
} yield attendee.name
```

```
coolPeople.map(println(_))
```



select * from attendees
where name like 'j%'
and 'scala' in skills


```
case class Person(name: String, skills: List[String])

implicit def t2p(t: Tuple2[String, List[String]]) = Person(t._1, t._2)

val attendees = List[Person](
  ("james", List("scala", ".net")),
  ("will", List("workday", ".net")),
  ("luke", List("ios", "drinking redbull")),
  ("jonny", List("java", "workday"))
)
```

implicit conversion

```
case class Person(name: String, skills: List[String])
```

```
implicit def t2p(t: Tuple2[String, List[String]]) = Person(t._1, t._2)
```

```
val attendees = List[Person](  
  ("james", List("scala", ".net")),  
  ("will", List("workday", ".net")),  
  ("luke", List("ios", "drinking redbull")),  
  ("jonny", List("java", "workday"))  
)
```

entirely optional generic type declaration

tuples as Persons

```
case class Person(name: String, skills: List[String])
```

```
class PersonHelper(person: Person){  
  def withSkills(skills: String*) : Person = {  
    person.copy(skills = person.skills ::: skills.toList)  
  }  
}
```

```
implicit def ph2p(person: Person) = new PersonHelper(person)
```

```
val attendees : List[Person] = List(  
  Person("james", List.empty).withSkills("scala", ".net"),  
  Person("will", List.empty).withSkills("workday", ".net"),  
  Person("luke", List.empty).withSkills("ios", "drinking redbull"),  
  Person("jonny", List.empty).withSkills("java", "workday")  
)
```

helper methods

```
case class Person(name: String, skills: List[String])
```

```
class PersonHelper(person: Person){  
  def withSkills(skills: String*) : Person = {  
    person.copy(skills = person.skills ::: skills.toList)  
  }  
}
```

implicit conversion

```
implicit def ph2p(person: Person) = new PersonHelper(person)
```

```
val attendees : List[Person] = List(  
  Person("james", List.empty).withSkills("scala", ".net"),  
  Person("will", List.empty).withSkills("workday", ".net"),  
  Person("luke", List.empty).withSkills("ios", "drinking redbull"),  
  Person("jonny", List.empty).withSkills("java", "workday")  
)
```

extension methods?
opening classes?
no ta!

```
trait Logging {
  def log(msg: String) = println(msg)
}

trait CleanerLogging extends Logging {
  override def log(msg: String) = println("[INFO] " + msg)
}

class SuperThing extends Thing with Logging {
  def doThing {
    log("doing my thing")
  }
}

object Program extends App {
  val thing1 = new SuperThing
  val thing2 = new SuperThing with CleanerLogging

  thing1.doThing
  thing2.doThing
}
```

a trait

```
trait Logging {  
  def log(msg: String) = println(msg)  
}
```

```
trait CleanerLogging extends Logging {  
  override def log(msg: String) = println("[INFO] " + msg)  
}
```

```
class SuperThing extends Thing with Logging {  
  def doThing {  
    log("doing my thing")  
  }  
}
```

multiple inheritance
static composition


```
object Program extends App {  
  val thing1 = new SuperThing  
  val thing2 = new SuperThing with CleanerLogging  
  
  thing1.doThing  
  thing2.doThing  
}
```

dynamic
composition

```
object Program extends scala.App {
  def guess(stream: Seq[Any]) = {
    stream match {
      case Seq(1, _) => "Number"
      case Seq('a', _) => "String"
      case _ => "Not a clue"
    }
  }

  println(guess(Seq(1,2,3,4)))
  println(guess(Seq(1)))
  println(guess(Seq('a', 'b', 'c')))
}
```

```
object Program extends scala.App {  
  def guess(stream: Seq[Any]) = {  
    stream match {  
      case Seq(1, _) => "Number"  
      case Seq('a', _) => "String"  
      case _ => "Not a clue"  
    }  
  }  
  
  println(guess(Seq(1,2,3,4)))  
  println(guess(Seq(1)))  
  println(guess(Seq('a', 'b', 'c')))  
}
```




pattern match
all the things


```
class FamilyMember
case class Mother(name: String) extends FamilyMember
case class Father(name: String, occupation: String) extends FamilyMember
case class Child(name: String, siblings: Option[List[Child]]) extends FamilyMember
```

```
object Program extends scala.App {
  def greet(person: FamilyMember) = {
    person match {
      case Mother(n) =>
        "Hello Mrs. %s".format(n)
      case Father(n, o) =>
        "Hello Mr. %s, hows the %sing industry?".format(n,o)
      case Child(n, s) =>
        "Howdy %s, how are your %d siblings doing?".format(n,
          s.getOrElse(List.empty).size)
      case _ =>
        "We dont take kindly to strangers here"
    }
  }
}
```

**a terrible pattern matching
example using cases classes**



```
println(greet(Mother("Emma")))
println(greet(Father("James", "Software Engineer")))
println(greet(Child("Ollie", Option(List(Child("Nathaniel", None)))))
}
```

```
object Program extends App {  
  
  val xml =  
    <cool_rankings>  
      <item position="1">billy zane</item>  
      <item position="2">eskimos</item>  
      <item position="3">stephen fry</item>  
    </cool_rankings>  
  
  xml \\ "item" foreach { item =>  
    println(item \\ "@position" + " - " + item.text)  
  }  
}
```

xml as 1st
class citizen

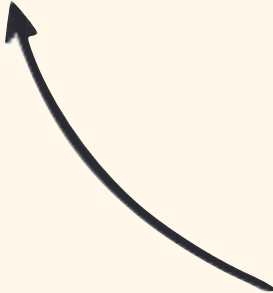
```
object Program extends App {  
  
  val xml =  
    <cool_rankings>  
      <item position="1">billy zane</item>  
      <item position="2">eskimos</item>  
      <item position="3">stephen fry</item>  
    </cool_rankings>  
  
  xml \\ "item" foreach { item =>  
    println(item \\ "@position" + " - " + item.text)  
  }  
}
```

xpath operator

DSL HELL!!!
here be dragons

```
object SquareRoot extends Baysick {  
  def main(args:Array[String]) = {  
    10 PRINT "Enter a number"  
    20 INPUT 'n  
    30 PRINT "Square root of " % "'n is " % SQR('n)  
    40 END  
  
    RUN  
  }  
}
```

John 11:35





Scala is complicated. but your code
doesn't need to be. See lift vs Play



IoC Containers & Dependency Injection
are techniques for solving problems
you don't have in Scala



Scala works with all your existing Javas,
JRubies, Groovies etc.



Scala is as fast and often faster
than its Java counterparts



II

Scala Web Frameworks



Scalatra

demo

play! 

demo

