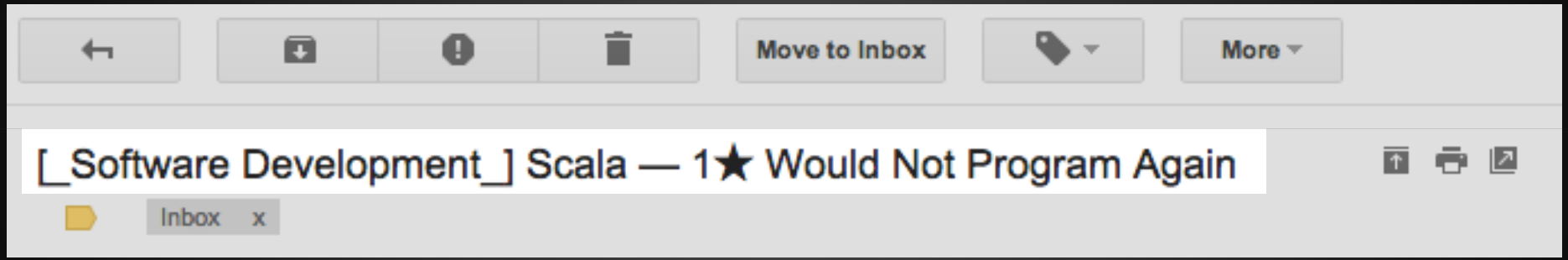
Scala (the good parts)

lessons learned while getting stuff done


CompFused.com


Sitting in a 3.8-metre sea kayak and watching a four-metre great white approach you is a fairly tense experience


Move to Inbox
More
[_Software Development_] Scala — 1★ Would Not Program Again
Inbox x

▼ [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again
Re: [_Software Development_] Scala — 1★ Would Not Program Again

Who am I?

- Consultant developer at ThoughtWorks

- I’ve been using Scala for the last two years

- Started at TST (Travel Syndication Solutions)

- Now working with Loyal3

Scala for **application development**

so **why** use Scala?

let’s see, in **code**

“Classes should be immutable unless there's a very good reason to make them mutable....

If a class cannot be made immutable, limit its mutability as much as possible.”

“Classes should be immutable unless there's a very good reason to make them mutable....

If a class cannot be made immutable, limit its mutability as much as possible.”

- Joshua Bloch (*Effective Java*)

Case Classes

and pattern matching

```
1  class Person {
2    var name: String = null
3  }
4
5  val bob = new Person
6
7  bob.name = "Bob"
8
9  bob.name should be ("Bob")
10
11 bob.name = bob.name + " Senior"
12
13 bob.name should be ("Bob Senior")
```

```
1  class Person {
2    var name: String = null
3  }
4
5  val bob = new Person
6
7  intercept[NullPointerException] {
8    bob.name.startsWith("B")
9  }
```

```
1  class Person {
2    var name: String = null
3  }
4
5  val bob = new Person
6  bob.name = "Bob"
7
8  val alsoBob = new Person
9  alsoBob.name = "Bob"
10
11 bob should not be alsoBob
```

```scala
class Person {
  var name: String = null

  override def equals(obj: Any): Boolean =
    obj match {
      case p: Person => p.name == name
      case _ => false
    }

  override def hashCode() = name.hashCode
}

def Person(name: String) = {
  if (name == null) {
    throw new IllegalArgumentException
  } else {
    val p = new Person
    p.name = name
    p
  }
}
```

```scala
val bob = Person("Bob")
val alsoBob = Person("Bob")

bob should be (alsoBob)
```

```scala
class Person(val name: String) {

  override def equals(obj: Any): Boolean = obj match {
    case p: Person => p.name == name
    case _ => false
  }

  override def hashCode() = name.hashCode
}

val bob = new Person("Bob")
val newBob = new Person(bob.name + " Senior")

newBob.name should be ("Bob Senior")
```

```scala
class Person(val name: String,
             val age: Int,
             val job: String) {
  override def equals(obj: Any): Boolean = {...}

  override def hashCode() = {...}

  def copy(name: String = name,
           age: Int = age,
           job: String = job) =
    new Person(name, age, job)
}

val bob = new Person("Bob", 42, "Cryptographer")

val newBob = bob.copy(name = bob.name + " Senior")

newBob.name should be ("Bob Senior")
```

```scala
case class Person(name: String, age: Int, job: String)

val bob = Person("Bob", 42, "Cryptographer")

val newBob = bob.copy(name = bob.name + " Senior")

newBob.name should be ("Bob Senior")
```

```scala
case class Person(name: String, age: Int, job: String)

val bob = Person("Bob", 42, "Cryptographer")

bob match {
  case Person(name, age, job) => age should be (42)
}
```

```scala
case class Name(first: String, last: String)
case class Person(name: Name, age: Int, job: String)

val bob = Person(
  name = Name("Bob", "Smith"),
  age = 42,
  job = "Cryptographer"
)

bob match {
  case Person(Name(first, "Smith"), _, _) =>
    first should be ("Bob")
  case Person(Name(first, "Jones"), _, _) =>
    fail("Last name should not be Jones')
}
```

case classes make **data types** lightweight

use them

but beware, they don’t mix with inheritance

Collections

that you actually want to use

```scala
val stringList = new java.util.ArrayList[String]()

stringList.add("1")
stringList.add("2")
stringList.add("3")

var total: Int = 0

for (string <- stringList.asScala) {
  val int = string.toInt

  if (int > 1) {
    total += int
  }
}

total should be (5)
```

```scala
val stringList = Seq("1", "2", "3")

var total: Int = 0
for (string <- stringList) {
  val int = string.toInt

  if (int > 1) {
    total += int
  }
}

total should be (5)
```

```scala
val stringList: Seq[String] = Seq("1", "2", "3")

val intList: Seq[Int] = stringList map { string =>
  string.toInt
}

var total: Int = 0
for (int <- intList) {
  if (int > 1) {
    total += int
  }
}

total should be (5)
```

```scala
val stringList = Seq("1", "2", "3")

val intList = stringList map { string =>
  string.toInt
}

val bigIntList = intList filter { int =>
  int > 1
}

var total: Int = 0
for (int <- bigIntList) {
  total += int
}

total should be (5)
```

```scala
val stringList = Seq("1", "2", "3")

val intList = stringList map { string =>
  string.toInt
}

val bigIntList = intList filter { int =>
  int > 1
}

var total: Int = 0
for (int <- bigIntList) {
  total += int
}

total should be (5)
```

```scala
val stringList = Seq("1", "2", "3")

val intList = stringList map { string =>
  string.toInt
}

val bigIntList = intList filter { int =>
  int > 1
}

val total = bigIntList reduce { (a, b) =>
  a + b
}

total should be (5)
```

```scala
val stringList = Seq("1", "2", "3")

val total = stringList
  .map(_.toInt)
  .filter(_ > 1)
  .reduce(_ + _)

total should be (5)
```

```scala
val stringList = new java.util.ArrayList[String]()

stringList.add("1")
stringList.add("2")
stringList.add("3")

var total: Int = 0

for (string <- stringList.asScala) {
  val int = string.toInt

  if (int > 1) {
    total += int
  }
}

total should be (5)
```

```scala
val stringList = Seq("1", "2", "3")

val total = stringList
  .map(_.toInt)
  .filter(_ > 1)
  .reduce(_ + _)

total should be (5)
```

think in **transformations**

“I call it my billion-dollar mistake”

- Tony Hoare

“I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.”

- Tony Hoare

Options

they aren’t optional

```
1  def lookup(i: Int) = if (i == 0) "result" else null
2
3  val maybeResult: String = lookup(0)
4
5  val output: String =
6    if (maybeResult != null) maybeResult else "nada!"
7
8  output should be ("result")
```

```scala
def lookup(i: Int) =
  if (i == 0) Some("result") else None

val maybeResult: Option[String] = lookup(0)

val output: String = maybeResult getOrElse "nada!"

output should be ("result")
```

```scala
def lookup1(i: Int) = if (i == 0) "result" else null
def lookup2(s: String) = if (s == "result") 42 else null

val maybeString: String = lookup1(0)

val output: String = if (maybeString == null) {
  null
} else {
  val maybeInt = lookup2(maybeString)
  if (maybeInt == null) {
    null
  } else {
    maybeInt.toString
  }
}

output should be ("42")
```

```scala
def lookup1(i: Int) = if (i == 0) Some("result") else None
def lookup2(s: String) = if (s == "result") Some(42) else None

val maybeResult: Option[Option[String]] =
  lookup1(0) map { string =>
    lookup2(string) map { int =>
      int.toString
    }
  }

val flatMaybeResult: Option[String] = maybeResult match {
  case Some(Some(x)) => Some(x)
  case _ => None
}

val output: String = flatMaybeResult getOrElse "nada!"

output should be ("42")
```

```scala
def lookup1(i: Int) = if (i == 0) Some("result") else None
def lookup2(s: String) = if (s == "result") Some(42) else None

val maybeResult: Option[String] =
  lookup1(0) flatMap { string =>
    lookup2(string) map { int =>
      int.toString
    }
  }

val output: String = maybeResult getOrElse "nada!"

output should be ("42")
```

```scala
val maybeString = lookup1(0)

if (maybeString == null){
  null
} else {
  val maybeInt = lookup2(maybeString)
  if (maybeInt == null) {
    null
  } else {
    maybeInt.toString
  }
}
```

```scala
lookup1(0) flatMap { string =>
  lookup2(string) map { int =>
    int.toString
  }
}
```

Option(**null**) should be (None)

think in **transformations**

the **type system** is on your side

‘maybe’ prefix is useful

```
Some("thing").isDefined should be (true)
Some("thing").get should be ("thing")
```

For Comprehensions

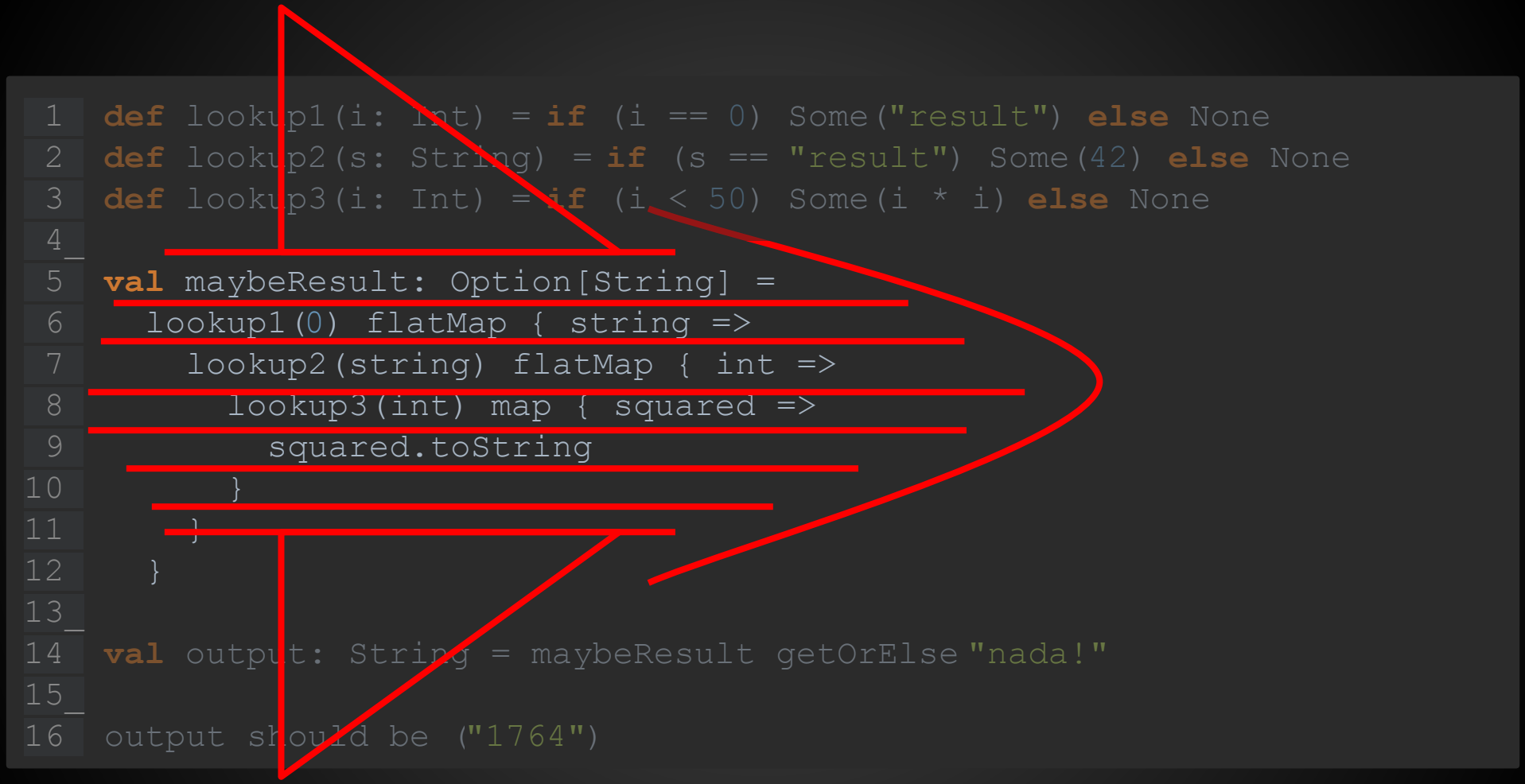
destroy the spaceships

```scala
def lookup1(i: Int) = if (i == 0) Some("result") else None
def lookup2(s: String) = if (s == "result") Some(42) else None
def lookup3(i: Int) = if (i < 50) Some(i * i) else None

val maybeResult: Option[String] =
  lookup1(0) flatMap { string =>
    lookup2(string) flatMap { int =>
      lookup3(int) map { squared =>
        squared.toString
      }
    }
  }

val output: String = maybeResult getOrElse "nada!"

output should be ("1764")
```


def lookup1(i: Int) = if (i == 0) Some("result") else None
def lookup2(s: String) = if (s == "result") Some(42) else None
def lookup3(i: Int) = if (i < 50) Some(i * i) else None
val maybeResult: Option[String] =
lookup1(0) flatMap { string =>
lookup2(string) flatMap { int =>
lookup3(int) map { squared =>
squared.toString
}
}
}
val output: String = maybeResult getOrElse "nada!"
output should be ("1764")

```scala
def lookup1(i: Int) = if (i == 0) Some("result") else None
def lookup2(s: String) = if (s == "result") Some(42) else None
def lookup3(i: Int) = if (i < 50) Some(i * i) else None

val maybeResult: Option[String] = for {
  string <- lookup1(0)
  int <- lookup2(string)
  squared <- lookup3(int)
} yield squared.toString

val output: String = maybeResult getOrElse "nada!"

output should be ("1764")
```

for one liners, just use map

```
(for (i <- Seq(1, 2)) yield i * i) should be (Seq(1, 4))
```

```
(for (r <- Some("result")) r) should be ()
```

Error Handling

eithers, ADTs, and pattern matching

```scala
class LookupException extends Exception

def lookup(i: Int): String = throw new LookupException

val result = try {
  lookup(1)
} catch {
  case e: LookupException => "nada!"
}

result should be ("nada!")
```

```scala
case class LookupError(msg: String)

def lookup(i: Int): Either[LookupError, String] =
  Left(LookupError("nada!"))

val result: String = lookup(1) match {
  case Right(s) => s
  case Left(LookupError(msg)) => msg
}

result should be ("nada!")
```

```scala
sealed trait LookupError
case class NotFound() extends LookupError
case class DatabaseUnavailable() extends LookupError

def lookup: Either[LookupError, String] =
  if (false) Right("result") else Left(NotFound())

val result = lookup match {
  case Right(string) => string
  case Left(NotFound()) => "not found"
}

result should be ("not found")

// [warn]
// ErrorHandlingSpec.scala:64: match may not be exhaustive.
// It would fail on the following input: Left(DatabaseUnavailable())
//   val result = lookup match {
```

algebraic data types

exceptions should be exceptional

Modularity

stay classy

```scala
trait OneComponent {
  trait One {
    def int: Int
  }

  def one: One
}
```

```scala
trait ProductionOneComponent extends OneComponent {
  class ProductionOne extends One {
    def int: Int = 1
  }

  def one: One = new ProductionOne
}
```

```scala
class Application { this: OneComponent =>

  def run() {
    one.int should be (1)
  }
}

(new Application with ProductionOneComponent).run()
```

```scala
implicit class OneInt(one: One) {
  def int: Int = 1
}

class One

class Application(one: One) {

  def run() {
    one.int should be (1)
  }
}

new Application(new One).run()
```

```
1  trait Number {
2    protected def int: Int
3  }
4
5  trait One extends Number {
6    protected def int: Int = 1
7  }
8
9  class Application extends One {
10
11   def run() {
12     int should be (1)
13   }
14 }
15
16 (new Application).run()
```

```scala
trait One {
  def int: Int
}

class ProductionOne extends One {
  def int: Int = 1
}

class Application(one: One) {

  def run() {
    one.int should be (1)
  }
}

new Application(new ProductionOne).run()
```

```scala
trait Number {
  def int: Int
  def string: String = s"Number $int"
}

class One extends Number {
  def int: Int = 1
}

(new One).string should be ("Number 1")
```

```scala
trait Number {
  def int: Int
}

trait AsString {
  protected def int: Int
  def string: String = s"Number $int"
}

class One extends Number with AsString {
  def int: Int = 1
}

(new One).string should be ("Number 1")
```

```scala
trait Number {
  def int: Int
}

trait AsString {
  protected def int: Int
  def string: String = s"Number $int"
}

class One extends Number {
  def int: Int = 1
}

val instance = new One with AsString

instance.string should be ("Number 1")
```

clarity is king

cleverness should be **encapsulated**

the good parts?

embrace **types**

immutability is key

OO **and** FP

think in **expressions**

Scala is **not Java**

?

Thanks!

Code: https://github.com/jacksingleton/scala-the-good-parts

Feedback: jacksingleton@thoughtworks.com

We're hiring: http://thoughtworks.com/join

ThoughtWorks®