Scala

Designed by Martin Odersky

Appeared in 2003

Version 2.9.0.1

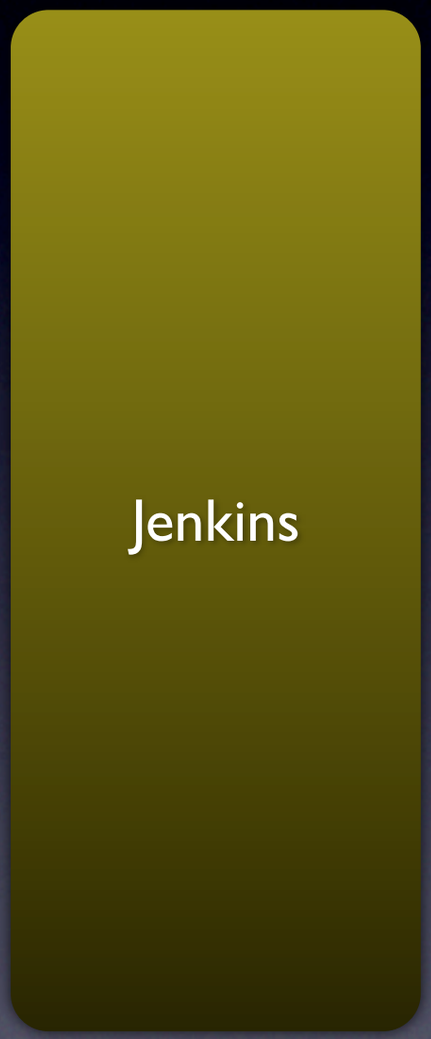
Supports object-oriented
and functional style

Scalable language - to grow
with the needs of users

Runs on the JVM

shouldExit
trapExit
waitingFor
ActorTracker(double, int, long, long, ObjectMapper)
_state() : Value
state$eq(Value) : void
$bang(Object) : void
$bang$bang(Object) : Future<Object>
$bang$bang(Object, PartialFunction<Object, A>) <A> : Future<A>
$bang$qmark(Object) : Object
$bang$qmark(long, Object) : Option<Object>
$qmark() : Object
act() : void
comovicatalogue$mastercatalogue$tracking$ActorTracker$$chanceOfTracking() : double
comovicatalogue$mastercatalogue$tracking$ActorTracker$$chanceOfTracking_$eq(double) : void
comovicatalogue$mastercatalogue$tracking$ActorTracker$$ensure(Map, long, String) : Map
comovicatalogue$mastercatalogue$tracking$ActorTracker$$expirationAfterEndEvent() : long
comovicatalogue$mastercatalogue$tracking$ActorTracker$$expirationAfterEndEvent_$eq(long) : void
comovicatalogue$mastercatalogue$tracking$ActorTracker$$logger() : Logger
comovicatalogue$mastercatalogue$tracking$ActorTracker$$mapper() : ObjectMapper
comovicatalogue$mastercatalogue$tracking$ActorTracker$$maximumTrackingCount() : int
comovicatalogue$mastercatalogue$tracking$ActorTracker$$maximumTrackingCount_$eq(int) : void
comovicatalogue$mastercatalogue$tracking$ActorTracker$$minimumTrackingTime() : long
comovicatalogue$mastercatalogue$tracking$ActorTracker$$minimumTrackingTime_$eq(long) : void
comovicatalogue$mastercatalogue$tracking$ActorTracker$$purge(Map) : Map

CI



CI

IDE

Jenkins

Eclipse

Idea

CI	IDE	Build
Jenkins	Eclipse	Maven
	Idea	Gradle

CI	IDE	Build	Web Container
Jenkins	Eclipse	Maven	JBoss
	Idea	Gradle	Jetty
			Tomcat

CI	IDE	Build	Web Container
Jenkins	Eclipse	Maven	JBoss
	Idea	Gradle	Jetty
		SBT	Tomcat

It's just JARs and WARs

Warning

Is it any good?

Concise

Classic point

Point has two integer values x and y

```
public class Point {
   private int x;
   private int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
    }
   public int getX() {
      return x;
    }
   public int getY() {
      return y;
    }
}
```

```
public class Point {
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int X { get; }
  public int Y { get; }
}
```

```
class Point(val x: Int, val y: Int)
```

Martin Odersky at Scala eXchange 2011

“ I spent a huge amount of code on it. ”

Martin Odersky at Scala eXchange 2011

“ I spent a huge amount of code on it.
I think four or five lines. ”

Object-oriented

```
scala> val p = new Point(2, 7)
p: Point = Point@6b38c54e

scala> println("[" + p.x + "," + p.y + "]")
[2,7]
```

REPL

read-eval-print-loop

Type inference

```
Map<String, Integer> stuff =
   new HashMap<String, Integer>();
```

```
Map<String, Integer> stuff =
  new HashMap<String, Integer>();
```

```
Map<String, Integer> stuff =
  new HashMap<>();
```

```
Map<String, Integer> stuff =
  new HashMap<String, Integer>();
```

```
Map<String, Integer> stuff =
  new HashMap<>();
```

```
var stuff = new HashMap[String, Int]()
```

```
val stuff = doSomething() // Returns a map
stuff foreach { println _ }
```

```
val stuff = doSomething() // Returns a map
stuff foreach { println _ }
```

```
val stuff = doSomething() // Returns a list
stuff foreach { println _ }
```

Collections

Most of the code we write deals with collections *

* These claims are made on some pretty loose evidence

Getting first names from a list of people

```
List<String> getFirstNames(Iterable<Person> people) {
  List<String> names = new ArrayList<String>();
  foreach (Person person in people) {
    names.add(person.first());
  }
  return names;
}
```

Getting first names from a list of people

```
def firstNames(people: Seq[Person]) = {
  people.map(_.first)
}
```

Create a map of people by their first name

```
def keyedByFirstName(people: Seq[Person]) = {
  people.map { p => (p.first, p) } toMap
}
```

Immutability

```
class Point(var x: Int, var y: Int)
```

(But try not to)

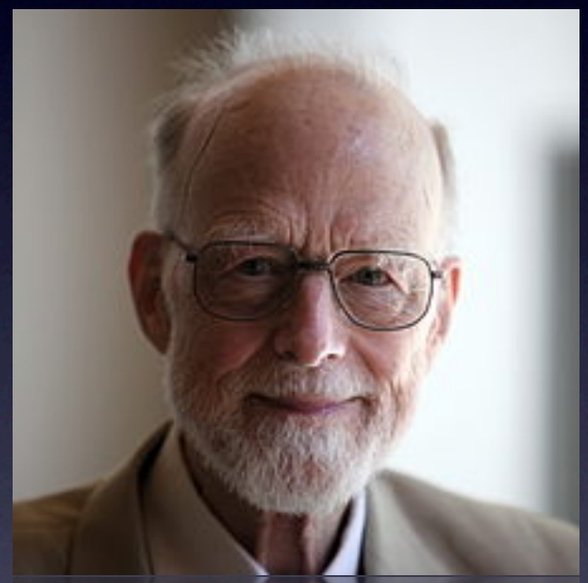
Option

Better than `java.lang.NullPointerException`

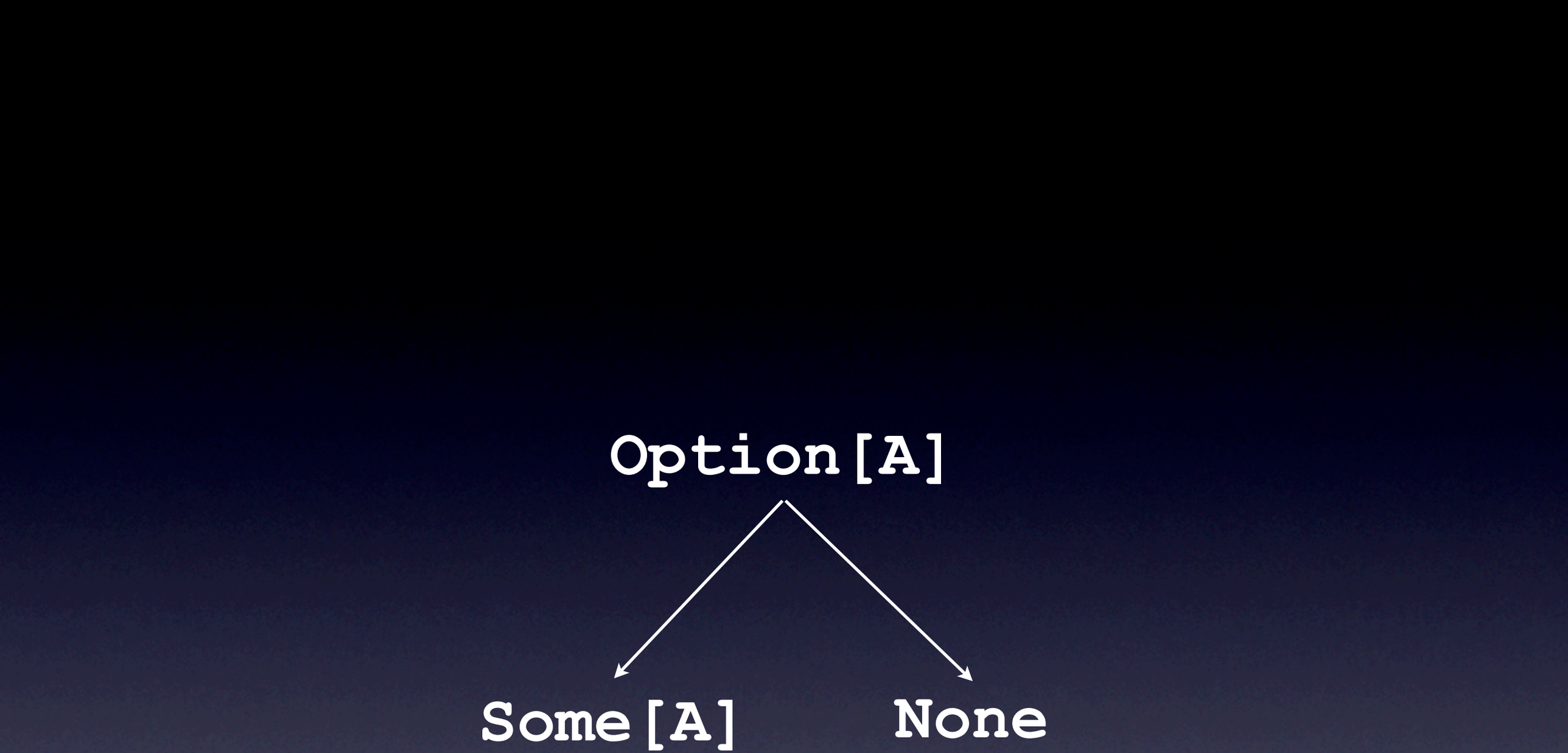
```
at com.github.restdriver.serverdriver.fil
at sun.reflect.NativeMethodAccessorImpl.i
at sun.reflect.NativeMethodAccessorImpl.i
at sun.reflect.DelegatingMethodAccessorIm
at java.lang.reflect.Method.invoke(Method
at org.junit.runners.model.FrameworkMetho
at org.junit.internal.runners.model.Refle
at org.junit.runners.model.FrameworkMetho
at org.junit.internal.runners.statements.
at org.junit.runners.BlockJUnit4ClassRunn
at org.junit.runners.BlockJUnit4ClassRunn
at org.junit.runners.BlockJUnit4ClassRunn
```

```
Map<String, String> map = new HashMap<String, String>();
map.put("foo", "bar");

List<Integer> valueLengths = new ArrayList<Integer>();

foreach (String key : Arrays.asList("foo", "rar")) {
  if (map.containsKey(key)) {
    valueLengths.add(map.get(key).length());
  } else {
    valueLengths.add(-1);
  }
}
```

Curse you Tony Hoare!



You lovely old man...


Option[A]
Some[A]
None

```
scala> val map = Map("foo" -> "bar")
map: scala.collection.immutable.Map
 [java.lang.String,java.lang.String] = Map((foo,bar))
```

```
scala> val map = Map("foo" -> "bar")
map: scala.collection.immutable.Map
 [java.lang.String,java.lang.String] = Map((foo,bar))

scala> map.get("foo")
res2: Option[java.lang.String] = Some(bar)
```

```
scala> val map = Map("foo" -> "bar")
map: scala.collection.immutable.Map
 [java.lang.String,java.lang.String] = Map((foo,bar))

scala> map.get("foo")
res2: Option[java.lang.String] = Some(bar)

scala> map.get("rar")
res3: Option[java.lang.String] = None
```

```
scala> val queryString = Map("view" -> "someView",
     | "junk" -> "ha")
queryString: ... = Map((view,someView), (junk,ha))
```

```
scala> val queryString = Map("view" -> "someView",
     | "junk" -> "ha")
queryString: ... = Map((view,someView), (junk,ha))

scala> val keys = List("items", "start", "view")
keys: ... = List(items, start, view)
```

```
scala> val queryString = Map("view" -> "someView",
     | "junk" -> "ha")
queryString: ... = Map((view,someView), (junk,ha))

scala> val keys = List("items", "start", "view")
keys: ... = List(items, start, view)

scala> val lowerCased = keys map { key =>
     |   queryString get(key) map {
     |     .toLowerCase
     |   }
     | }
lowerCased: ... = List(None, None, Some(someview))
```

```
scala> val queryString = Map("view" -> "someView",
     | "junk" -> "ha")
queryString: ... = Map((view,someView), (junk,ha))

scala> val keys = List("items", "start", "view")
keys: ... = List(items, start, view)

scala> val lowerCased = keys map { key =>
     |   queryString get(key) map {
     |     .toLowerCase
     |   }
     | }
lowerCased: ... = List(None, None, Some(someview))

scala> val ready = keys zip lowerCased filter { kv =>
     |   kv._2 isDefined
     | } map { t =>
     |   (t._1, t._2.get)
     | } toMap
ready: ... = Map((view,someview))
```

Pattern matching

```
scala> def check(result: Option[String]) = {
     |   result match {
     |     case Some(x) => println(x)
     |     case None => println("nope")
     |   }
     | }
check: (result: Option[String])Unit
```

```
scala> def check(result: Option[String]) = {
     |   result match {
     |     case Some(x) => println(x)
     |     case None => println("nope")
     |   }
     | }
check: (result: Option[String])Unit
```

```
scala> check(map.get("foo"))
bar
```

```
scala> def check(result: Option[String]) = {
     |   result match {
     |     case Some(x) => println(x)
     |     case None => println("nope")
     |   }
     | }
check: (result: Option[String])Unit

scala> check(map.get("foo"))
bar

scala> check(map.get("rar"))
nope
```

```
scala> def first(list: List[String]) = {
     |   list match {
     |     case Nil => None
     |     case x :: _ => Some(x)
     |   }
     | }
first: (list: List[String])Option[String]
```

```
scala> def first(list: List[String]) = {
     |   list match {
     |     case Nil => None
     |     case x :: _ => Some(x)
     |   }
     | }
first: (list: List[String])Option[String]
```

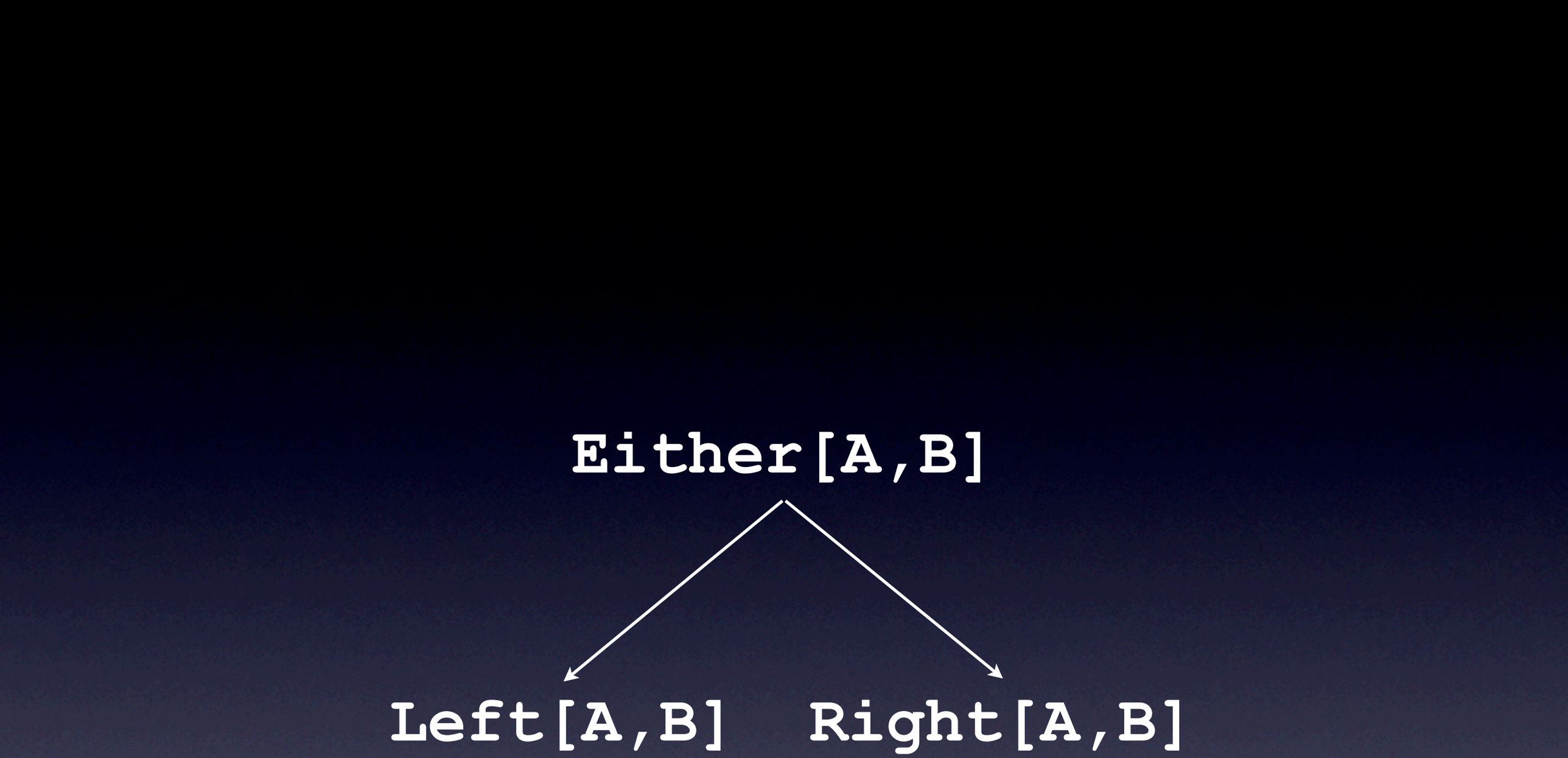
```
scala> first(List())
res13: Option[String] = None
```

```
scala> def first(list: List[String]) = {
     |   list match {
     |     case Nil => None
     |     case x :: _ => Some(x)
     |   }
     | }
first: (list: List[String])Option[String]

scala> first(List())
res13: Option[String] = None

scala> first(List("foo"))
res14: Option[String] = Some(foo)
```

```
scala> def first(list: List[String]) = {
     |   list match {
     |     case Nil => None
     |     case x :: _ => Some(x)
     |   }
     | }
first: (list: List[String])Option[String]

scala> first(List())
res13: Option[String] = None

scala> first(List("foo"))
res14: Option[String] = Some(foo)

scala> first(List("foo", "bar"))
res15: Option[String] = Some(foo)
```

Either

No more exceptions?


Either[A,B]
Left[A,B]
Right[A,B]

```
scala> case class Rejection(message: String)
defined class Rejection
```

```
scala> case class Rejection(message: String)
defined class Rejection

scala> def validate(value: String) = {
     |   value match {
     |     case null => Left(Rejection("Can't be null"))
     |     case "" => Left(Rejection("Can't be empty"))
     |     case _ => Right(value)
     |   }
     | }
validate: (value: String)Product with Either[Rejection,String]
```

```
scala> case class Rejection(message: String)
defined class Rejection

scala> def validate(value: String) = {
     |   value match {
     |     case null => Left(Rejection("Can't be null"))
     |     case "" => Left(Rejection("Can't be empty"))
     |     case _ => Right(value)
     |   }
     | }
validate: (value: String)Product with Either[Rejection,String]

scala> validate("something")
res3: Product with Either[Rejection,String] = Right(something)
```

```
scala> case class Rejection(message: String)
defined class Rejection

scala> def validate(value: String) = {
     |   value match {
     |     case null => Left(Rejection("Can't be null"))
     |     case "" => Left(Rejection("Can't be empty"))
     |     case _ => Right(value)
     |   }
     | }
validate: (value: String)Product with Either[Rejection,String]

scala> validate("something")
res3: Product with Either[Rejection,String] = Right(something)

scala> validate("")
res4: Product with Either[Rejection,String] =
  Left(Rejection(Can't be empty))
```


THE
MIDDLE
OF THE
TALK

```
using (some IDisposable) {
    . . .
}
```

```
try (some AutoCloseable) {
    ...
}
```

```
type Closeable = { def close(): Unit }
```

```
type Closeable = { def close(): Unit }

def using[C <: Closeable] (closeable: C) (f: C => Unit) {
  try {
    f(closeable)
  } finally {
    closeable.close()
  }
}
```

```
type Closeable = { def close(): Unit }

def using[C <: Closeable] (closeable: C) (f: C => Unit) {
  try {
    f(closeable)
  } finally {
    closeable.close()
  }
}

class Thing(val property: String) {
  def close() = println("Closed")
}
```

```
type Closeable = { def close(): Unit }

def using[C <: Closeable] (closeable: C) (f: C => Unit) {
  try {
    f(closeable)
  } finally {
    closeable.close()
  }
}

class Thing(val property: String) {
  def close() = println("Closed")
}

val outside = "Chilly out"

using (new Thing("foo")) { t =>
  println("Doing some business")
  println(outside)
  println(t.property)
}
```

```
type Closeable = { def close(): Unit }

def using[C <: Closeable] (closeable: C) (f: C => Unit) {
  try {
    f(closeable)
  } finally {
    closeable.close()
  }
}

class Thing(val property: String) {
  def close() = println("Closed")
}

val outside = "Chilly out"

using (new Thing("foo")) { t =>
  println("Doing some business")
  println(outside)
  println(t.property)
}
```

```
Doing some business
Chilly out
foo
Closed
```

```
type Closeable = { def close(): Unit }

def using[C <: Closeable] (closeable: C) (f: C => Unit) {
  try {
    f(closeable)
  } finally {
    closeable.close()
  }
}

class Thing(val property: String) {
  def close() = println("Closed")
}


using (new Thing("bar")) { t =>
  println("Heading for trouble")
  throw new Exception("Argh!")
}
```

```
type Closeable = { def close(): Unit }

def using[C <: Closeable] (closeable: C) (f: C => Unit) {
  try {
    f(closeable)
  } finally {
    closeable.close()
  }
}

class Thing(val property: String) {
  def close() = println("Closed")
}
```

```
using (new Thing("bar")) { t =>
  println("Heading for trouble")
  throw new Exception("Argh!")
}
```

```
Heading for trouble
Closed
Exception in thread "mai
    at com.ovi.logging.So
    at com.ovi.logging.So
    at com.ovi.logging.So
    at com.ovi.logging.So
    at scala.Function0$cl
```

Make it better?

```scala
def using[A, C <: Closeable] (closeable: C) (f: C => A) = {
  try {
    Right(f(closeable))
  } catch {
    case e => Left(e)
  } finally {
    closeable.close()
  }
}
```

```
def using[A, C <: Closeable] (closeable: C) (f: C => A) = {
  try {
    Right(f(closeable))
  } catch {
    case e => Left(e)
  } finally {
    closeable.close()
  }
}
```

```
val result = using (new Thing("foo")) { t =>
  t.property
}
println(result)
```

```
def using[A, C <: Closeable] (closeable: C) (f: C => A) = {
  try {
    Right(f(closeable))
  } catch {
    case e => Left(e)
  } finally {
    closeable.close()
  }
}
```

```
val result = using (new Thing("foo")) { t =>
  t.property
}
println(result)
```

```
Closed
Right(foo)
```

```
def using[A, C <: Closeable] (closeable: C) (f: C => A) = {
  try {
    Right(f(closeable))
  } catch {
    case e => Left(e)
  } finally {
    closeable.close()
  }
}
```

```
val result = using (new Thing("foo")) { t =>
  throw new Exception("Argh!")
}
println(result)
```

```
def using[A, C <: Closeable] (closeable: C) (f: C => A) = {
  try {
    Right(f(closeable))
  } catch {
    case e => Left(e)
  } finally {
    closeable.close()
  }
}
```

```
val result = using (new Thing("foo")) { t =>
  throw new Exception("Argh!")
}
println(result)
```

```
Closed
Left(java.lang.Exception: Argh!)
```

Traits

Interfaces on crack

```
trait Store {
  def getItem(id: String): Option[Item]
  def getItems(ids: String*): List[Option[Item]]
}
```

```
public interface Store {
  Item getItem(String id);
  List<Item> getItems(String... ids);
}
```

```
class MemoryStore extends Store {

  private val items = Map[String,Item]()

  def getItem(id: String) = {
    items.get(id)
  }

  def getItems(ids: String*) = {
    val lb = new ListBuffer[Option[Item]]()
    ids.foreach { lb += items.get(_) }
    lb.toList
  }

}
```

```
class MemoryStore extends Store {

  private val items = Map[String,Item]()

  def getItem(id: String) = {
    items.get(id)
  }

  def getItems(ids: String*) = {
    val lb = new ListBuffer[Option[Item]]()
    ids.foreach { lb += getItem(_) }
    lb.toList
  }

}
```

```
trait Store {
  def getItem(id: String): Option[Item]

  final def getItems(ids: String*) = {
    val lb = new ListBuffer[Option[Item]]()
    ids.foreach { lb += getItem(_) }
    lb.toList
  }
}
```

```
class MemoryStore extends Store {

  private val items = Map[String,Item]()

  def getItem(id: String) = {
    items.get(id)
  }

}
```

Multiple inheritance

```
class SearchRequest(val text: String)
```

```
class SearchRequest(val text: String)

class EntityRequest(val id: String)
```

```
trait Paging {
  val start: Int
  val items: Int
  val range = start until (start + items)
}
```

```
trait Paging {
  val start: Int
  val items: Int
  val range = start until (start + items)
}

class SearchRequest(val text: String,
  val start: Int, val items: Int) extends Paging
```

```
trait View {
  val view: String
}
```

```
trait View {
  val view: String
}

class EntityRequest(val text: String,
  val view: String) extends View
```

Let’s put views on searches!

Great idea!

Java says no...

```
class SearchRequest(val text: String,
val start: Int, val items: Int,
val view: String) extends Paging with View
```

Pimping

Contrary to popular belief, it is easy

```
class SearchResult(val id: String, val name: String)
```

```
class SearchResult(val id: String, val name: String) {
  def toXml = {
    <searchResult>
      <id>{ id }</id>
      <name>{ name }</name>
    </searchResult>
  }
}
```

```
class RichXmlSearchResult(val sr: SearchResult) {
  def toXml = {
    <searchResult>
      <id>{ sr.id }</id>
      <name>{ sr.name }</name>
    </searchResult>
  }
}
```

```
class RichXmlSearchResult(val sr: SearchResult) {
  def toXml = {
    <searchResult>
      <id>{ sr.id }</id>
      <name>{ sr.name }</name>
    </searchResult>
  }
}
```

```
object XmlPimps {
  implicit def result2Xml(sr: SearchResult) = {
    new RichXmlSearchResult(sr)
  }
}
```

```
class RichXmlSearchResult(val sr: SearchResult) {
  def toXml = {
    <searchResult>
      <id>{ sr.id }</id>
      <name>{ sr.name }</name>
    </searchResult>
  }
}
```

```
object XmlPimps {
  implicit def result2Xml(sr: SearchResult) = {
    new RichXmlSearchResult(sr)
  }
}
```

```
import XmlPimps._
new SearchResult("id", "name").toXml
```

```
class RichXmlSearchResult(val sr: SearchResult) {
  def toXml = {
    <searchResult>
      <id>{ sr.id }</id>
      <name>{ sr.name }</name>
    </searchResult>
  }
}
```

```
object XmlPimps {
  implicit def result2Xml(sr: SearchResult) = {
    new RichXmlSearchResult(sr)
  }
}
```

```
import XmlPimps._
result2Xml(new SearchResult("id", "name")).toXml
```

Default parameters

SBT

Akka

Scalaz

for comprehensions

Currying

Scalatest

So much more...

Specs2

.NET

Compound types

Extractors

Actors

Higher-order functions

Why should we use it?

Less code

Immutability

Powerful language features

Collections API

More fun?

Any Scala being done here?

Questions?