Scala War Stories

You go to war with the code you have...

Not the code you might want, or wish to have at a later time.

Your Presenter

- Paul Phillips (paulp@improving.org)
- Full-time on scalac for ~5 years
- Co-founder of Typesafe
- I don't speak for Typesafe!

Jvm Language Summit

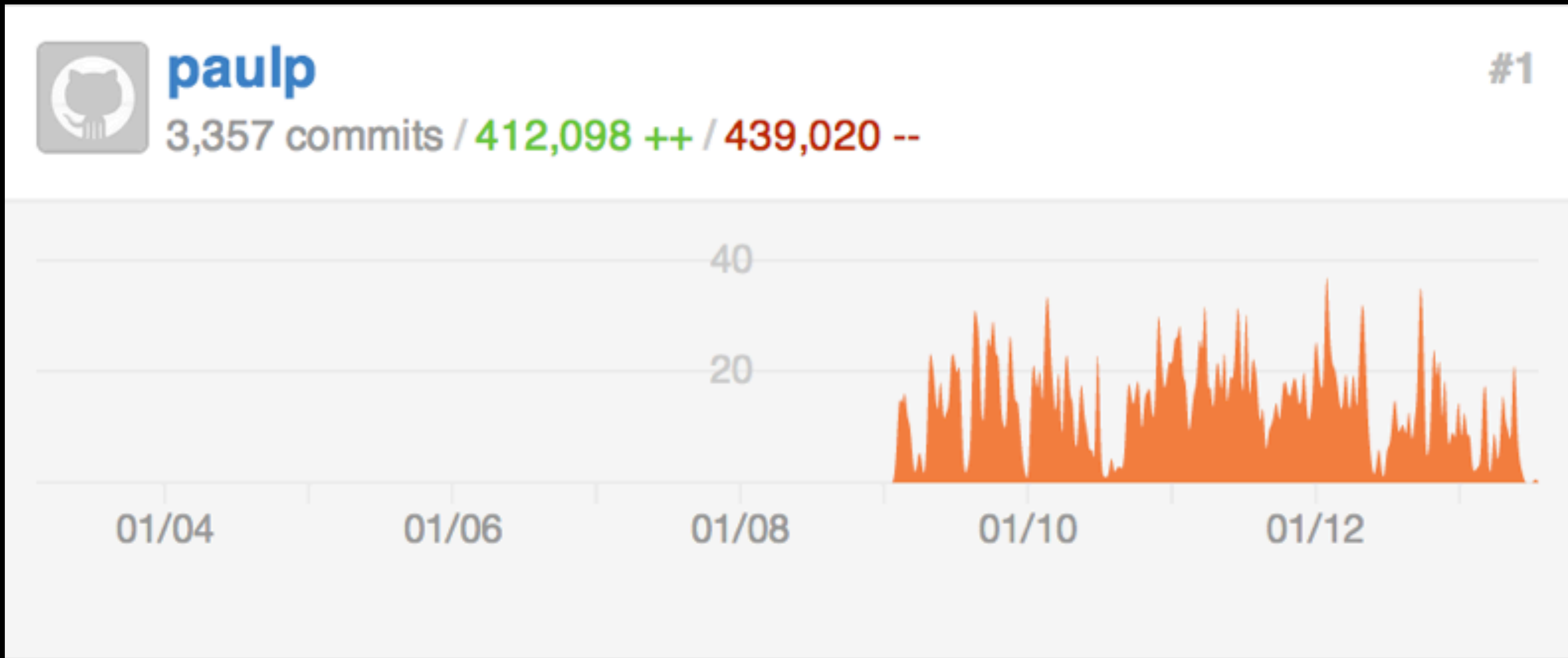
- I attend maybe 3 conferences a year
- I'm 6/6 to jvml (and I'm not local)
- Many thanks to Brian and all those from Oracle who make it happen
- Equal thanks to all the attendees, from whom I have learned much

Set expectations on "stun"

- I worked like a dog on these slides
- Sadly, it doesn't show
- So many dead ends
- If you find your time has been ill-spent, I owe you dinner

Credentials

(even a blind squirrel finds a few nuts in 3357 nut-searches)



Source: https://github.com/scala/scala/graphs

Discredentials

- Our bytecode sophistication is low
- ...historically anyway. Present company excepted.
- Our implementation makes "easy" things hard
- Higher layers are insatiable time sinks
- Bytecode improvements in the last year largely thanks to @JamesIry and @gkossakowski

Tales Untold

suffice to say I consider these real categories

- Attempting to omit:
  - Friendly fire incidents
  - Slipshod construction by former officers
  - Proxy warfare
  - Cavalry charging machine gun nests

Scala War Stories

Three quarters of the things on which all code is based are lying in a fog of uncertainty to a greater or lesser extent.

Constraints

- Assume non-negotiable standing orders
  - separate compilation
  - "java interop" (depends on who you ask)
  - no classloader magic or bytecode rewriting
  - more recently: seek binary compatibility

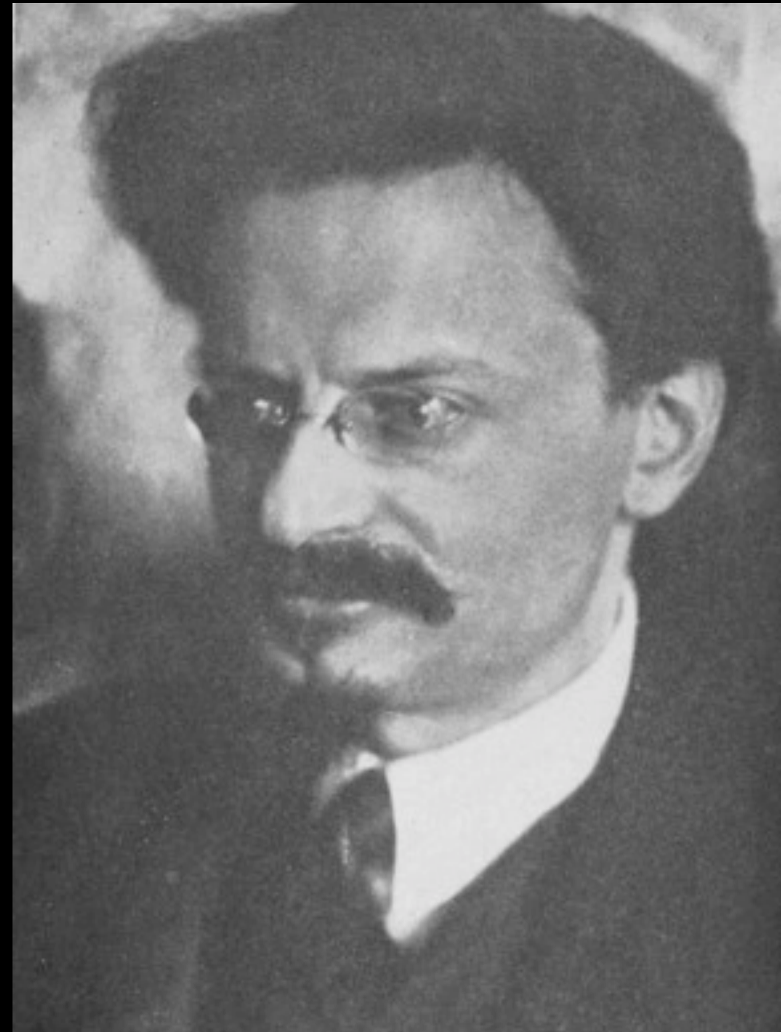
Many a slip twixt...

- Scala language semantics
- In principle, platform agnostic
- Conceived without much deference to the jvm

- Runnable bytecode
- Once-virtuous agnosticism now a liability
- Leaky abstractions threaten to inundate lowlands

Scala War Stories

You may not be interested in code, but code is interested in you.



Scala: the Unification Church

three namespaces: packages, methods, and fields
unify all terms, the "uniform access principle"

irregular handling of primitive types
unify all types under single object model

irregular handling of Arrays
unify Arrays with other collections

irregular handling of void, null, and non-termination
the Church welcomes Unit, Null, and Nothing!

Unification can always burn

- By definition, you are eliminating a distinction
- That means if your cover is not airtight, breakage will ensue wherever the distinction appears
- Or maybe you can eliminate it so thoroughly nobody will ever have to think of it again
- Ha ha, there's always a first time

What is NaN == NaN?

```
public class J {
  static float fp = Float.NaN;
  static java.lang.Float fb = new java.lang.Float(fp);

  public static void main(String[] args) {
    System.out.println(fp == fp);
    System.out.println(fb.equals(fb));
  }
}

% java J
false
true
```

What is NaN == NaN?

```
public class J {
  static float fp = Float.NaN;
  static java.lang.Float fb = new java.lang.Float(fp);

  public static void main(String[] args) {
    System.out.println(fp == fp);
    System.out.println(fb.equals(fb));
  }
}

% java J
false
true
```

Where resides foo.bar?

```
package foo;
public class J { J() { } }

package foo
class S private[foo]()

// Those definitions are "identical" to the point it is
// possible to express, but still not identical enough

package foo.bar
class B1 extends foo.S // allowed
class B2 extends foo.J // must be disallowed, or...

java.lang.IllegalAccessError: tried to access method
foo.J.<init>()V from class foo.bar.B2
    at foo.bar.B2.<init>(b.scala:3)
```

Scala War Stories

Never, never, never believe any feature will be smooth and easy, or that anyone who embarks on the strange voyage can measure the tides and hurricanes he will encounter.

It's lonely at the top

"the limits of my language means the limits of my world"

```
// Do these classes admit the same set of type arguments?

// In java, every type parameter is implicitly <: Object
public class J<T> { }

// In scala every type is <: Any
class S[T]

// Should we assume they would have written "Any" if their
// language had a word for it? And does it matter?
```

```
// The irregularities we're unsuccessfully papering over
// are also one reason it matters

class A {
  // ()[Ljava/lang/Object;
  def x1: Array[_ <: AnyRef] = null

  // ()Ljava/lang/Object;
  def x2: Array[_ <: Any] = null
}

// In other words, Array[Object] erases to Array[Object]
// but Array[Any] erases to Object
```

It's also lonely at the bottom

"What's better than the best thing and worse than the worst thing?"

```
scala> def f(xs: Array[_ <: String]) = xs.length
f: (xs: Array[_ <: String])Int
// public int f(java.lang.String[]);

scala> f(Array[Nothing]())
java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to
[Ljava.lang.String;

// Oops, we have painted ourselves into the lion's den
// Our bottom types have semantics which cannot be represented
// on the jvm - an Array[Nothing] is at best an Array[Object],
// certainly not an Array[String]

MORE BRIDGES? NOOOOOOOOOOOOOO
```

Digression on bridges: seq() in List

a trifling 34

```
abstract sc.Iterable<A> sc.GenIterable.seq()
abstract sc.Iterable<A> sc.Iterable.seq()
abstract sc.LinearSeq<A> sc.LinearSeq.seq()
abstract sc.LinearSeq<A> sc.LinearSeqLike.seq()
abstract sc.Seq<A> sc.GenSeq.seq()
abstract sc.Seq<A> sc.GenSeqLike.seq()
abstract sc.Seq<A> sc.Seq.seq()
abstract sc.Traversable<A> sc.GenTraversable.seq()
abstract sc.Traversable<A> sc.Traversable.seq()
abstract sc.TraversableOnce<A>
sc.GenTraversableOnce.seq()
abstract sc.TraversableOnce<A> sc.Parallelizable.seq()
abstract sc.TraversableOnce<A> sc.TraversableOnce.seq()
abstract sci.Iterable<A> sci.Iterable.seq()
abstract sci.LinearSeq<A> sci.LinearSeq.seq()
abstract sci.Seq<A> sci.Seq.seq()
abstract sci.Traversable<A> sci.Traversable.seq()
```

```
sc.Iterable sc.AbstractSeq.seq()
sc.Iterable sci.List.seq()
sc.Iterable<A> sc.AbstractIterable.seq()
sc.LinearSeq sci.List.seq()
sc.Seq sci.List.seq()
sc.Seq<A> sc.AbstractSeq.seq()
sc.Traversable sc.AbstractIterable.seq()
sc.Traversable sc.AbstractSeq.seq()
sc.Traversable sci.List.seq()
sc.Traversable<A> sc.AbstractTraversable.seq()
sc.TraversableOnce sc.AbstractIterable.seq()
sc.TraversableOnce sc.AbstractSeq.seq()
sc.TraversableOnce sc.AbstractTraversable.seq()
sc.TraversableOnce sci.List.seq()
sci.Iterable sci.List.seq()
sci.LinearSeq<A> sci.List.seq()
sci.Seq sci.List.seq()
sci.Traversable sci.List.seq()
```

HOW MANY ARTIFACTS
IS TOO MANY?

We don't need Nothing

my, aren't we the self-reliant ones

```
// same as before
scala> def f(xs: Array[_ <: String]) = xs.length
f: (xs: Array[_ <: String])Int
// public int f(java.lang.String[]);

scala> f(Array[Int with String]())
java.lang.ClassCastException: [I cannot be cast to
[Ljava.lang.String;
```

Calamity! Arrays reify uninhabited types!

the takeaway

- If we faithfully represent our type system at the bytecode level, there will be blood
- If we weaken our types until the jvm won't give us trouble, java interop is a casualty
- descriptors and signatures are the only language we know which java can hear

Scala War Stories

Nowadays it is the fashion to emphasize the horrors of the last project. I didn't find it so horrible. There are just as horrible things happening in this codebase, if only we had eyes to see them.

If you can't unify, diversify

insufficiently granular access
so create a bunch of new levels

single inheritance
as many supers as you like

no bottom types
More applause for Messrs. Null and Nothing

no xml literals
okay, cheap shot

diversify, continued

eager evaluation
by-name parameters, lazy vals,
objects, Stream, views...

invariant type parameters
definition site variance

no metaprogramming
macros! (but "experimental")

At War with Primitives

```
// We forward == to equals,
// but we can't do so naively

scala> null.equals("")
java.lang.NullPointerException

scala> (new jl.Long(1L)).equals(new jl.Integer(1))
res1: Boolean = false
```

Okay, so we'll be "smart" about it.

```
scala> def f1[T](x: T, y: T) = x == y
f1: [T](T,T)Boolean

scala> f1[java.io.Serializable](1, 1L)
res0: Boolean =

scala> def f1(x: java.io.Serializable, y: java.io.Serializable) = x == y
f1: (java.io.Serializable,java.io.Serializable)Boolean

scala> f1(1, 1L)
res1: Boolean =
```

```
scala> def f1[T](x: T, y: T) = x == y
f1: [T](T,T)Boolean

scala> f1[java.io.Serializable](1, 1L)
res0: Boolean = true

scala> def f1(x: java.io.Serializable, y: java.io.Serializable) = x == y
f1: (java.io.Serializable,java.io.Serializable)Boolean

scala> f1(1, 1L)
res1: Boolean =
```

```
scala> def f1[T](x: T, y: T) = x == y
f1: [T](T,T)Boolean

scala> f1[java.io.Serializable](1, 1L)
res0: Boolean = true

scala> def f1(x: java.io.Serializable, y: java.io.Serializable) = x == y
f1: (java.io.Serializable,java.io.Serializable)Boolean

scala> f1(1, 1L)
res1: Boolean = false
```

```
scala> def f1[T](x: T, y: T) = x == y
f1: [T](T,T)Boolean

scala> f1[java.io.Serializable](1, 1L)
res0: Boolean = true

scala> def f1(x: java.io.Serializable, y: java.io.Serializable) = x == y
f1: (java.io.Serializable,java.io.Serializable)Boolean

scala> f1(1, 1L)
res1: Boolean = false
```

You mean...

- java boxed types scoff at our ideas
- So we must intercept every call to == which could conceivably be comparing boxed numerics
- And that's one which was missed

Scala War Stories

Great is the guilt of unnecessary code.

Where equals goes, hashCode follows

```
scala> 1.hashCode
res0: Int = 1

scala> 1d.hashCode
res1: Int = 1072693248

scala> 1 == 1d
res2: Boolean =

scala> 1 == 1072693248
res3: Boolean =
```

Where equals goes, hashCode follows

```
scala> 1.hashCode
res0: Int = 1

scala> 1d.hashCode
res1: Int = 1072693248

scala> 1 == 1d
res2: Boolean = true

scala> 1 == 1072693248
res3: Boolean =
```

Where equals goes, hashCode follows

```
scala> 1.hashCode
res0: Int = 1

scala> 1d.hashCode
res1: Int = 1072693248

scala> 1 == 1d
res2: Boolean = true

scala> 1 == 1072693248
res3: Boolean = false
```

Where equals goes, hashCode follows

```
scala> 1.hashCode
res0: Int = 1

scala> 1d.hashCode
res1: Int = 1072693248

scala> 1 == 1d
res2: Boolean = true

scala> 1 == 1072693248
res3: Boolean = false
```

You can't mean...

- With == and equals, we already had indirection
- Not so with hashCode. Invent a new method.
- A warm welcome for new member Hash Hash.

```
scala> 1d.hashCode
res0: Int = 1072693248

scala> 1d.##
res1: Int = 1
```

So do we really performance instance checks whenever the operands might be java boxes?

```
trait Bippy
class A {
  def f(x: Any, y: Any) = x == y
  def g(x: Bippy, y: Bippy) = x == y
}

public boolean f(Object, Object);
 0: aload_1
 1: aload_2
 2: // BoxesRunTime.equals
 5: ifeq  12
 8: iconst_1
 9: goto  13
 12:  iconst_0
 13:  ireturn
```

```
public boolean g(Bippy, Bippy);
 0: aload_1
 1: aload_2
 2: astore_3
 3: dup
 4: ifnonnull 15
 7: pop
 8: aload_3
 9: ifnull  22
 12:  goto  26
 15:  aload_3
 16:  // Object.equals:(LObject;)Z
 19:  ifeq  26
 22:  iconst_1
 23:  goto  27
 26:  iconst_0
 27:  ireturn
```

Scala War Stories

The supreme art of programming is to solve the problem without code.

I'm a Simple Trait

```
trait B {
  private[this] val m = "B"
}
class X extends B
```

```
// The interface
public interface B {
  public abstract int B$$m();
  public abstract void B$_setter_$B$$m_$eq(int);
}

// The implementation class
public abstract class B$class {
  public static void $init$(B);
     0: aload_0
     1: iconst_1
     2: invokeinterface #13, 2 // B.B$_setter_$B$$m_$eq:(I)V
     7: return
}
```

```
public class X implements B {
  private final int B$$m;
  public int B$$m() ...
  public void B$_setter_$B$$m_$eq(int) ...

  public X();
     0: aload_0
     1: invokespecial #24
     4: aload_0
     5: invokestatic  #30  // Method B$class.$init$:(LB;)V
     8: return
}
```

The scenic route

1) class X inherits private field m from trait B
2) static "trait constructor" $init$ is in B$class
3) interface B offers a public setter for use by $init$
4) X implements the setter to set private field B$$m
5) X's constructor passes X itself to B$class.$init$
6) B$class.$init$ sets B$$m via the public setter

m is private

(the NSA confirms it)

```
// Change the name of "private" m to mmm.
// Recompile B, leave X alone.
scala> new X
AbstractMethodError: X.B$_setter_$B$$mmm_$eq(I)V

// Or change private m's type from Int to Long.
scala> new X
AbstractMethodError: X.B$_setter_$B$$m_$eq(J)V
```

Scala War Stories

Veni, vidi, vi.

Okay, this one is a bust.