Scala:
The *Industrial* Parts

Marius Eriksen (@marius, marius@twitter.com)
Scala Symposium, June 13, 2015

Agenda

Setting, scale

How we use Scala (distilled)

The pitfalls of Scala

Taming Scala

Setting, scale

Twitter is a *large organization.*

- $O(10^3)$ developers
- $O(10^8)$ users
- $O(10^7)$ lines of code
- $O(10^4)$ opinions
- 5+ cafeterias

Engineering

Engineers come from *different experience levels.*

- Many new grads; junior engineers
- Many senior engineers without FP background
- Many disciplines: mobile, web, machine learning, OS, HPC, systems, runtimes, etc.

Monolithic repository

- Everything is built, deployed from source.

Large degree of consistency

- Consistent set of systems and libraries

Computing environment

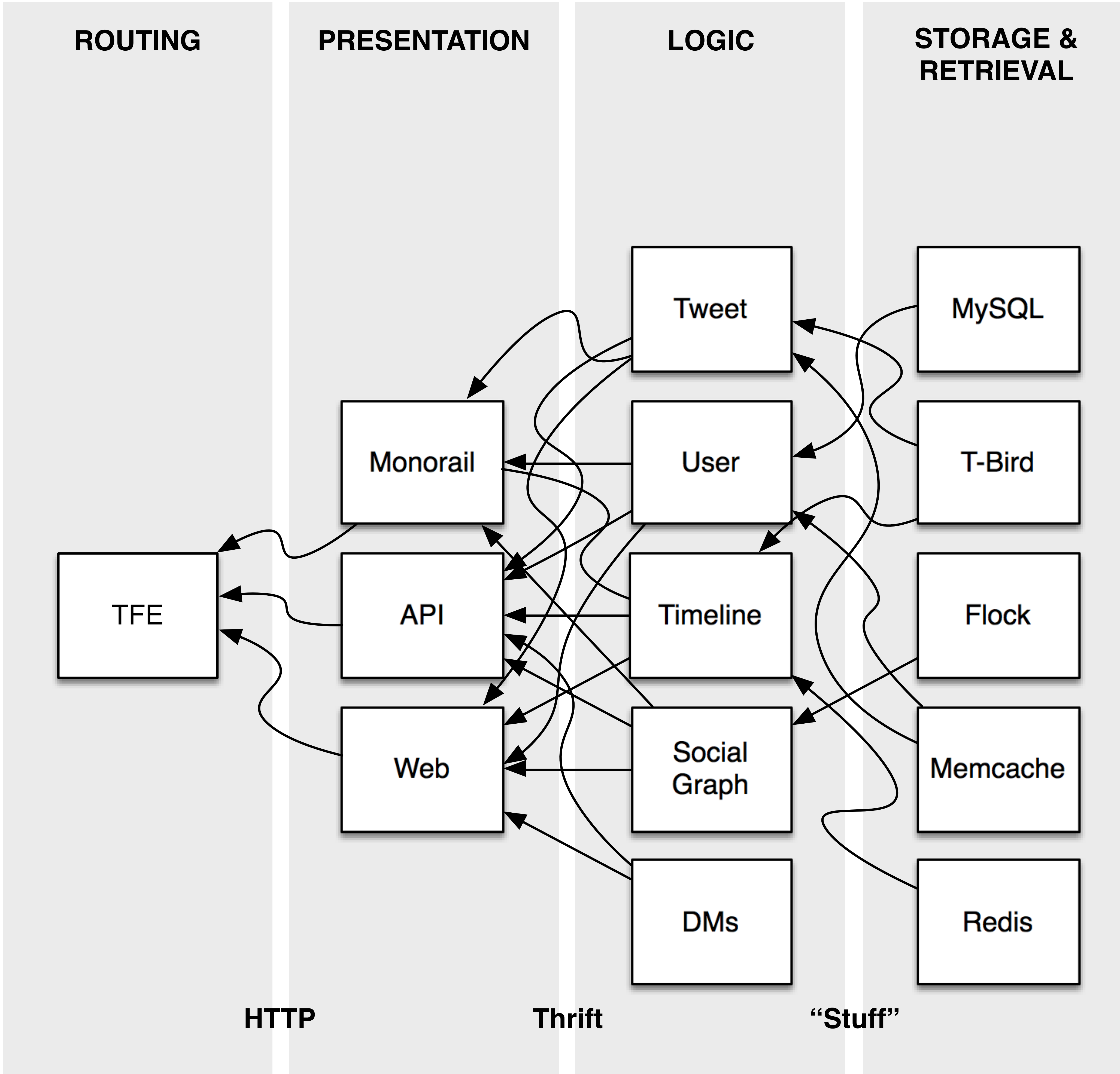
Our target is the *datacenter*.

Thus our focus on efficiency, performance, correctness, resilience, and safety are viewed through this lens.

This in turn informs how we use our tools and languages.

Design space

Functionality	*It has to work*
Scalability	*It must be growable*
Operability	*It must be diagnosable, fixable*
Efficiency	*It must be cheap*
Flexibility	*It must be changeable*


ROUTING
PRESENTATION
LOGIC
STORAGE & RETRIEVAL
Tweet
MySQL
Monorail
User
T-Bird
TFE
API
Timeline
Flock
Web
Social Graph
Memcache
DMs
Redis
HTTP
Thrift
“Stuff”

Caveat emptor

This talk is about the use of Scala in our *setting*. It is highly distilled.

It may not apply to your own use; or maybe any other use at all.

(But I’d like to think that it generalizes.)

Your server as a function

Services

- Highly concurrent
- Complicated operating environment: asynchronous networks, partial failures, noisy neighbors
- Needs to support many protocols (e.g., Mux, HTTP, Thrift, memcached, MySQL, redis..)

“Concurrent programs wait faster” —Hoare

Futures

```
// A container for a future T-typed
// value. May fail or never complete
// at all.
val f: Future[T]
```

Futures

More than ersatz threads: Future-based concurrency has great impedance matching in distributed systems

- Asynchronous
- **`Future`** typing is *good*
- Composable results/errors
- Persistent, easy to reason with
- Liberate *semantics* from *mechanics*

The basis for (nearly) all concurrency at Twitter.

Futures

Engineers are familiar and comfortable with collections

Futures give them access to concurrent programming through the same, natural APIs

Without the need to deal with the minutiae and book-keeping of dealing with threads as resources

This has been huge

Services

Services are asynchronous functions, used to represent real services

```
trait Service[Req, Rep]
    extends (Req => Future[Rep])

val http:    Service[HttpReq, HttpRep]
val redis:   Service[RedisCmd, RedisRep]
val thrift: Service[TFrame, TFrame]
```

Services are symmetric

```
// Client:
val http = Http.newService(..)

// Server:
Http.serve(..,
  new Service[HttpReq, HttpRep] {
    def apply(..) = ..
  }
)

// A proxy:
Http.serve(.., Http.newService(..))
```

Filters

A `Service` talks about an application; a `Filter` talks about application-agnostic behaviors, e.g.,

- Timeouts
- Retries
- Statistics
- Logging
- Authentication

Composes `Services`

Filters

```
trait Filter[
  ReqIn, ReqOut,
  RepIn, RepOut]
extends
  ((ReqIn, Service[ReqOut, RepIn])
    => Future[RepOut])
```

In other words, given a request and a service, a filter produces a response

Timeout filter

```
class TimeoutFilter[Req, Rep](
  to: Duration)
extends Filter[Req, Rep, Req, Rep] {

  def apply(
    req: Req,
    svc: Service[Req, Rep]) =
  svc(req).within(to)
}
```

Filters & services

```
val timeout =
  new TimeoutFilter(1.second)
val auth = new AuthFilter

val authAndTimeout: Filter[..] =
  auth andThen timeout

val service: Service[..] = ..

val authAndTimeoutService =
  authAndTimeout andThen service
```

In the real world

```
recordHandletime      andThen
traceRequest          andThen
collectJvmStats       andThen
parseRequest          andThen
logRequest            andThen
recordClientStats     andThen
sanitize              andThen
respondToHealthCheck andThen
applyTrafficControl  andThen
virtualHostServer
```

A systems basis

Futures, services, and filters are the *orthogonal* basis upon which our service software is written

- Easy to answer what functionality belongs where.

The style of programming encourages good modularity, separation of concerns.

- Enhanced flexibility.
- Piecemeal composition.

Most of our systems are phrased as big future transformers.

- Simple to reason about.

Your Server as a Function

Marius Eriksen
Twitter Inc.
marius@twitter.com

n a large-scale setting, where systems ex-
urrency and environmental variability, is
n the most experienced programmer. Ef-
tness are paramount—goals which have
h modularity, reusability, and flexibility.
actions which combine to present a pow-
for building safe, modular, and efficient

Services Systems boundaries are repr
functions called *services*. They prov
form API: the same abstraction re
servers.

Filters Application-agnostic concerns (
thentication) are encapsulated by *filte*
services from multiple independent r

Stitch

Service-oriented programming

- Want concurrency between calls
- Want to be efficient by taking advantage of batch APIs (e.g., fetch every tweet in a timeline)
- Want clear, flexible, modular code

This often leads to spaghetti code that mixes operational concerns—e.g., batching—with application code.

A typical batch API

```
sealed trait Req
case class ReqA(...) extends Req
case class ReqB(...) extends Req

sealed trait Resp
case class RespA(...) extends Resp
case class RespB(...) extends Resp

def call(reqs: Seq[Req])
  : Future[Seq[Resp]]
```

Typical use

```
val reqs = ... // mix of Req{A,B}s
val resps = call(reqs) // mix of
Resp{A,B}s

reqs.zip(resps).map {
  case (ReqA(...), RespA(...)) => ...
  case (ReqB(...), RespB(...)) => ...
  case _ => // can’t happen
}
```

Stitch

```
def call(req: ReqA): Stitch[RespA]
def call(req: ReqB): Stitch[RespB]

Stitch.join(
  call(reqA) map { respA => ...
  call(reqB) map { respB => ...
)
```

Stitch is a monad

```
trait Stitch[T] {
  def map[U](f: T => U): Stitch[U]
  def flatMap[U](f: T => Stitch[U])
    : Stitch[U]
  def handle[T](f: Throwable => T)
    : Stitch[T]
  def rescue[T](f: Throwable =>
Stitch[T])
    : Stitch[T]
  …
}
```

Stitch is a monad

```
object Stitch {
  def value[T](t: T): Stitch[T]
  def join[A,B](
    a: Stitch[A], b: Stitch[B])
    : Stitch[(A, B)]
  def collect[T](ss: Seq[Stitch[T]])
    : Stitch[Seq[T]]
  def traverse[T, U](ts: Seq[T])
    (f: T => Stitch[U]): Stitch[Seq[U]]
  def run[T](s: Stitch[T]): Future[T]
}
```

Service adaptors

```
case object CallGroup
    extends SeqGroup[Req, Resp] {
  def run(calls: Seq[Req]) =
    service.call(calls)
}

Stitch.call(req, CallGroup)
  : Stitch[Resp]
```

Execution model

A query is represented as a syntax tree.

When called, we find exposed calls.

Can “see” into join, traverse, map but not flatMap (data dependency).

Group calls together, execute batch RPCs.

On each RPC return, simplify query, repeat.

Stitch

Separation of concerns

- Batching/query plan vs. application logic

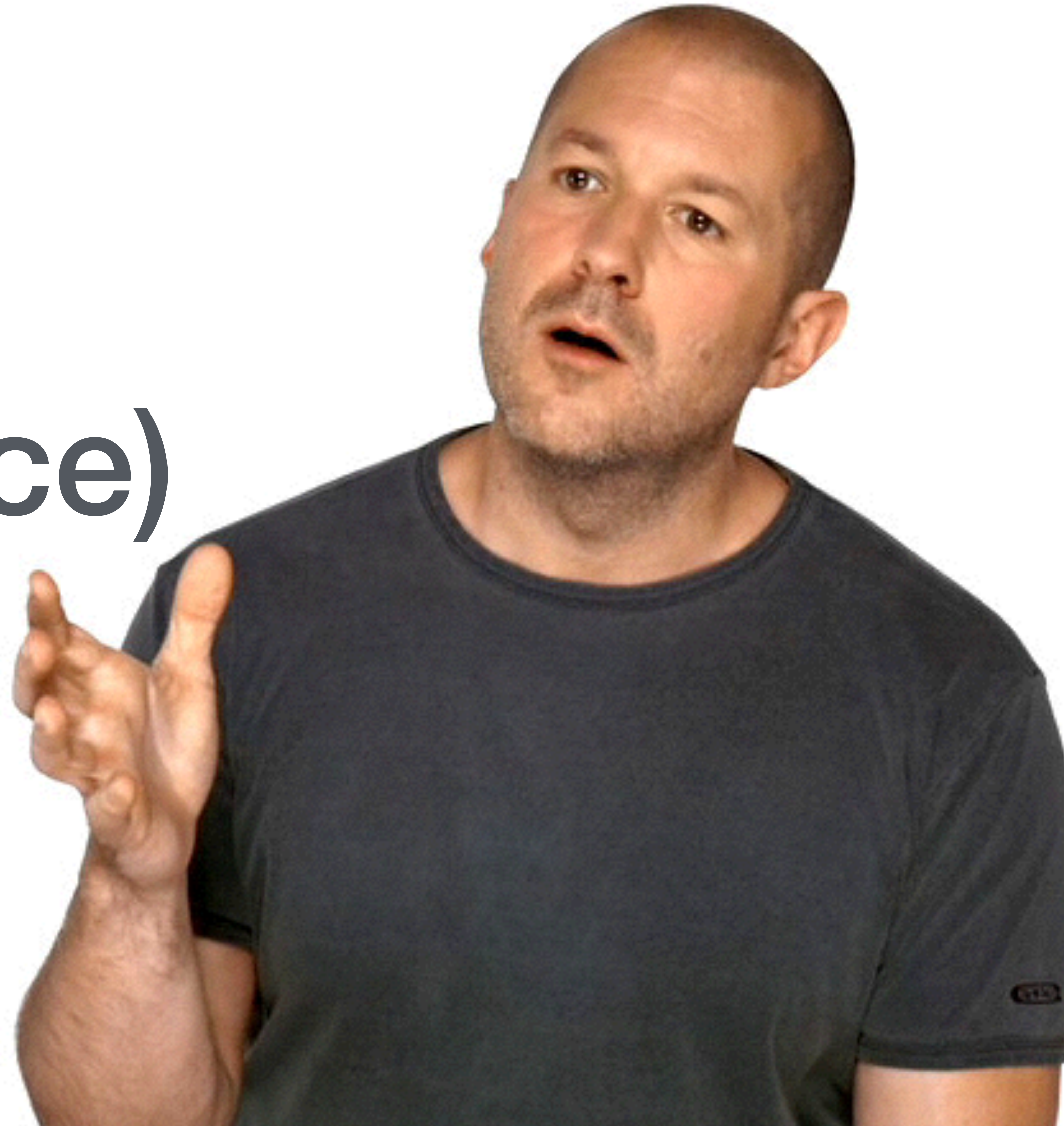
Composition

- Queries are combined, and made more efficient for it!

Other interesting uses

- DSLs for data processing — Scalding
- Online/offline/nearline unification — Summingbird
- Raw comparator generators via macros
- Software stacks as first class values
- Self-adjusting computation

Where Scala is Complex (In practice)



Scala’s complexities

In practice, Scala is quite large and complex.

- While the language itself is reasonably orthogonal, interactions are quite tricky.
- This is multiplied by interactions with Java, the JVM, Java’s object model, syntax sugar, inference, …

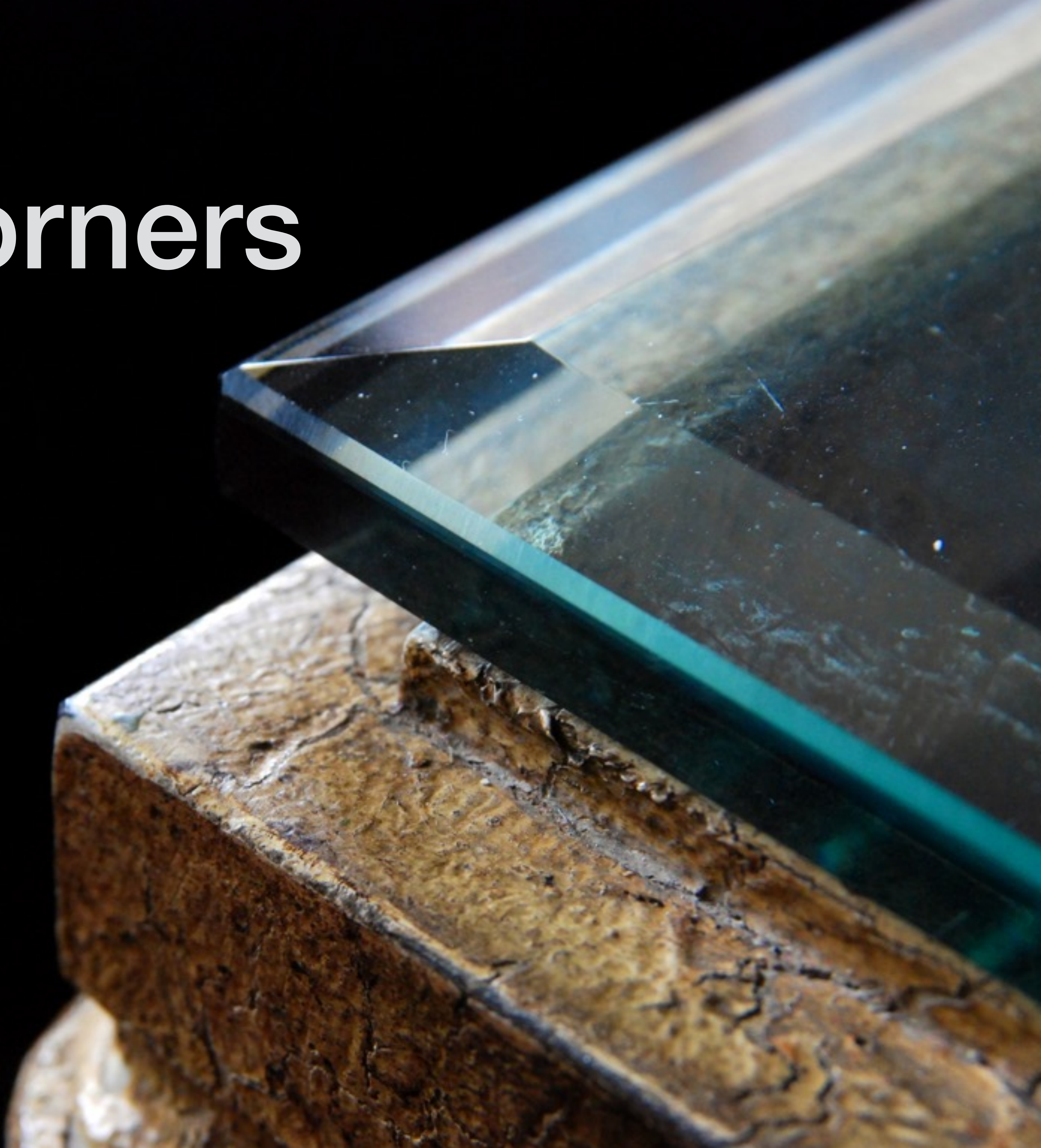
You can do many powerful things with Scala; but is the cost justified in our setting?

Ingredients

Type system; classes, traits, type classes, structural types, abstract types, …; Java interoperability; cost of abstraction; syntax sugar; interaction between features; boxing; variance; existentials; uppercase constants, matching; initialization order; implicit resolution; closure capture; lazy initialization; return-as-exception; nullary methods vs (); call-by-name; …

(Some ingredients don't mix well.)

A few sharp corners



Underscore literals

Don’t always do what you expect them to.

```
future.map {
  count.incrementAndGet()
  process(_)
}
```

Underscore literals (2)

“Weird” type checking errors.

```
keys.map { (A(_), b) }
```

Uppercase vals

```
val X = 123
val x = 333
val seq: Seq[Int] = ...

seq match {
  case Seq(X) => "X"
  case Seq(x) => "x"
  case _ => "unknown"
}
```

Initialization order, vals

Oldie-but-goodie. Still bites people a lot.

```
trait Client {
  val connection: Connection
}

trait EnrichedClient { self: Client =>
  val rich = Enrich(connection)
}

new Client with EnrichedClient {
  val connection = new TcpConnection
}
```

Lazy val deadlocks

```
object Y {
  lazy val x = X.x
}

object X {
  lazy val x = 1
  lazy val y = Y.x
}
```

(Accidental) structured types

```
val client = new {
  def connect(): Unit = …
  …
}

vs.

object client {
  def connect(): Unit = …
}
```

Collections

Scala's collections are extremely powerful. Most problems can be dispatched with a few lines of carefully chosen combinators.

"Easy to use: A small vocabulary of 20-50 methods is enough to solve most collection problems in a couple of operations." —Scala docs

Magic

But: no-one understands how they work: there is an excess of "magic."

- Performance semantics/issues; difficult to debug.
- Often difficult to reason about — what is a **`Seq`**?

Can often lead to, or even encourage, cryptic code.

- Code is read more than it is written.

Difficult to reason about locality.

flatMap

Try to get a non-expert to understand this signature.
(Which is simple by collections standards.)

```
def flatMap[B, That](f: A =>
GenTraversableOnce[B])(implicit bf:
CanBuildFrom[Repr, B, That]): That
```

breakOut

```
val elems: Seq[Elem]
val map: Map[Key, Value] =
  elems.map { elem =>
    (elem.key, elem.value)
  } .toMap
```

Avoid creating intermediaries:

```
import collection.breakOut
val map: Map[Key, Value] =
  elems.map { elem =>
    (elem.key, elem.value)
  }(breakOut)
```

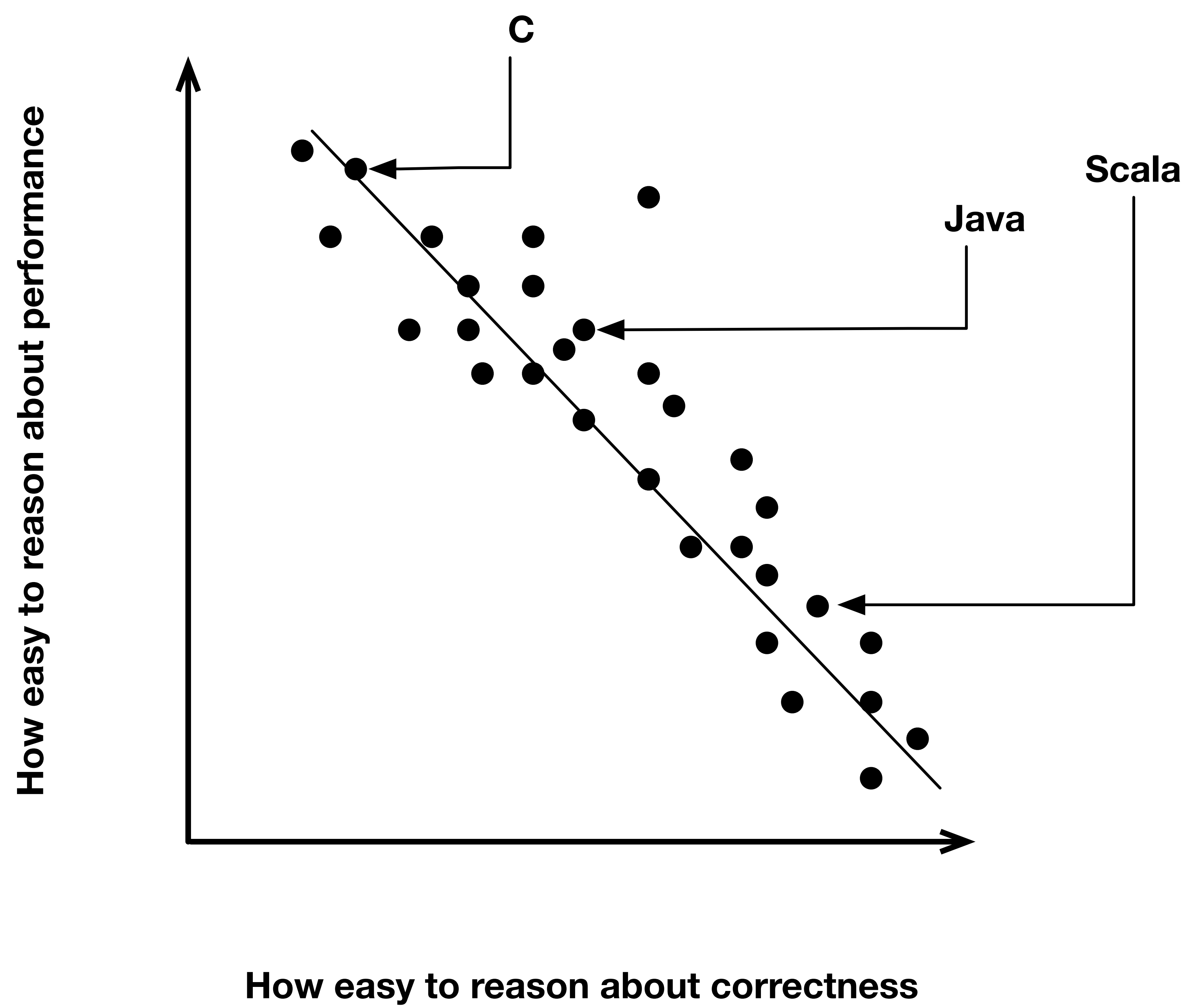
DRY at all costs

Collections complicated things relatively simple; but usually you just don't want to do complicated things, but rather simple things in a predictable way.

Scala's collections make the wrong tradeoffs here. The details of a DRY implementation leak to the UX, cost model, and predictability.

Performance vs. correctness

Performance/correctness



Performance/correctness

```
def distinct: Repr = {
  val b = newBuilder
  val seen = mutable.HashSet[A]()
  for (x <- this) {
    if (!seen(x)) {
      b += x
      seen += x
    }
  }
  b.result()
}
```

Innocuous-seeming code...

```
def process(seq: Seq[Int]): Unit = {
  for (i <- seq if i < 10)
    return

  …
}
```

All of the layers

You have to understand a large number of layers to understand Scala's performance characteristics.

- Syntax sugar
- Compiler frontend
- Java object model mapping
- Compiler back-end
- Java runtime model

And all of their interactions.

Tooling

In many ways, language tooling is more important than the language itself; e.g.,

- profilers
- IDEs
- formatting, linting
- source code formatting
- upgrading

Runtime tooling

Scala inherits much of Java's runtime tooling, but we all know these:

```
at Main$$anon$1$Foo$$anonfun$bar$2.apply(frame.scala:6)
at Main$$anon$1$Foo$$anonfun$bar$2.apply(frame.scala:5)
at scala.collection.TraversableLike$$anonfun$map
$1.apply(TraversableLike.scala:244)
at scala.collection.TraversableLike$$anonfun$map
$1.apply(TraversableLike.scala:244)
at scala.collection.immutable.List.foreach(List.scala:
318)
at scala.collection.TraversableLike
$class.map(TraversableLike.scala:244)
at
scala.collection.AbstractTraversable.map(Traversable.sc
ala:105)
```

Compilation speed

Compilation speed

One of the most frequent complaints among newcomers and old-timers alike.

Speed is one of the most important features. Scala doesn't really provide it.

Standard arguments aren't convincing to most.

Refactoring

Large-scale refactoring is difficult with Scala.

Example: a simple transformation required a custom compiler plugin to be built:

```
future.get -> Await.result(future)
```

Tools can be hugely beneficial in our setting.

O*pin"ion*a`ted (?), a. Stiff in opinion; firmly or unduly adhering to one's own opinion or to preconceived notions; obstinate in opinion. Sir W. Scott.

Why is it important?

Instant familiarity.

- Consistent, predictable, and simple code.

Much of modern software engineering involves spelunking into code quickly; familiarizing yourself.

- Consistency breeds familiarity.

Scala is *un*opinionated

By its very nature, Scala is a rather *unopinionated* language.

- Many ways to do any one thing

Scala offers a *buffet of abstraction*.

- Newcomers are bewildered; experts spend a lot of time picking tools.
- Unnecessary effort.

With great power comes great responsibility.

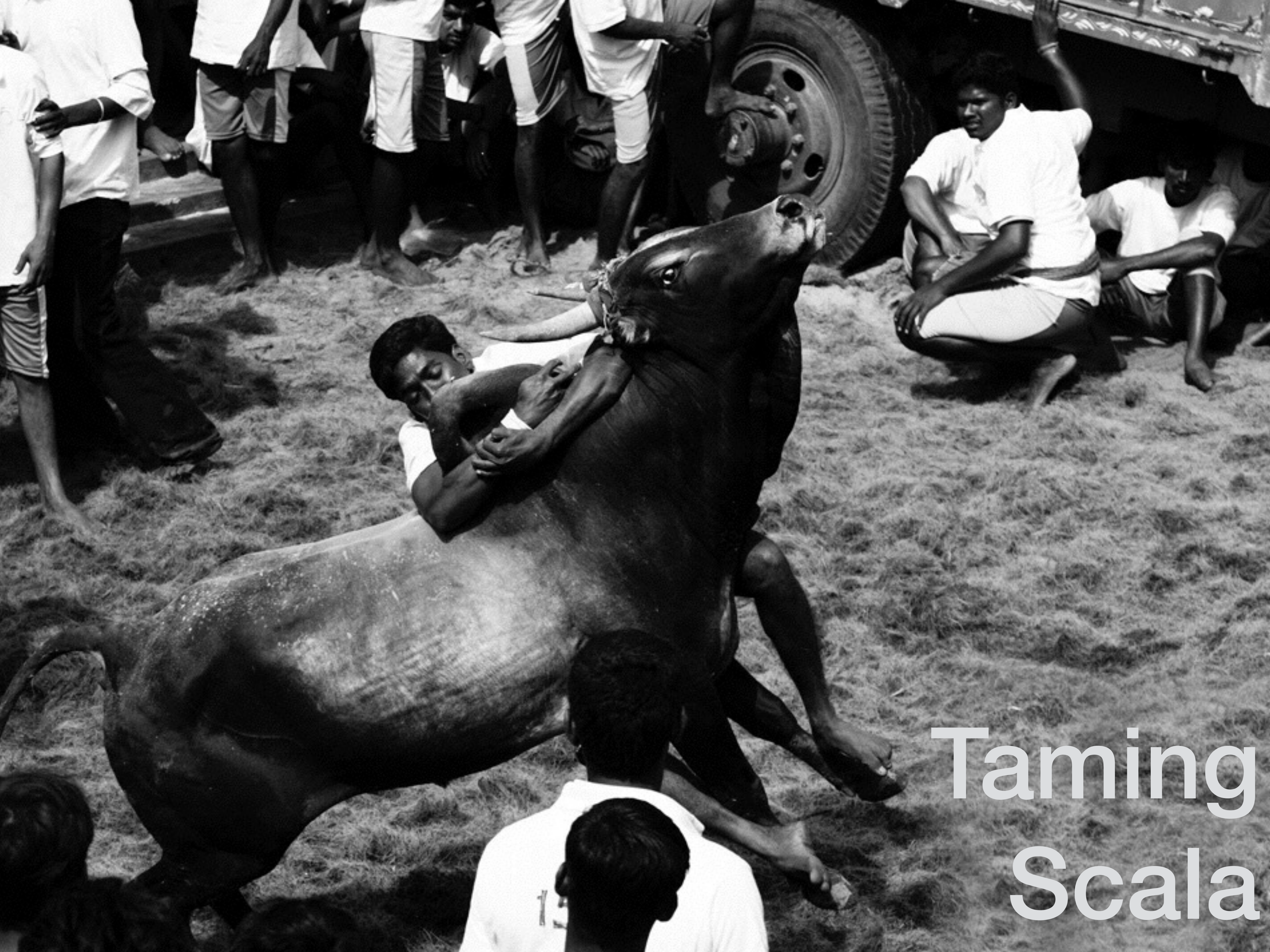
- Is it possible to have both?

Scala’s buffet of abstraction

Even simple problems in Scala requires you to answer many questions.

For example, which tool of abstraction should you reach for?

- Traits + mix-ins?
- Classes + hierarchy?
- Type classes?
- “ML style” modules + syntax?


Taming
Scala

Taming Scala

How do we make the best use out of such a powerful language in our setting?

- Large scale organization
- Mix of experience levels
- Efficiency is paramount

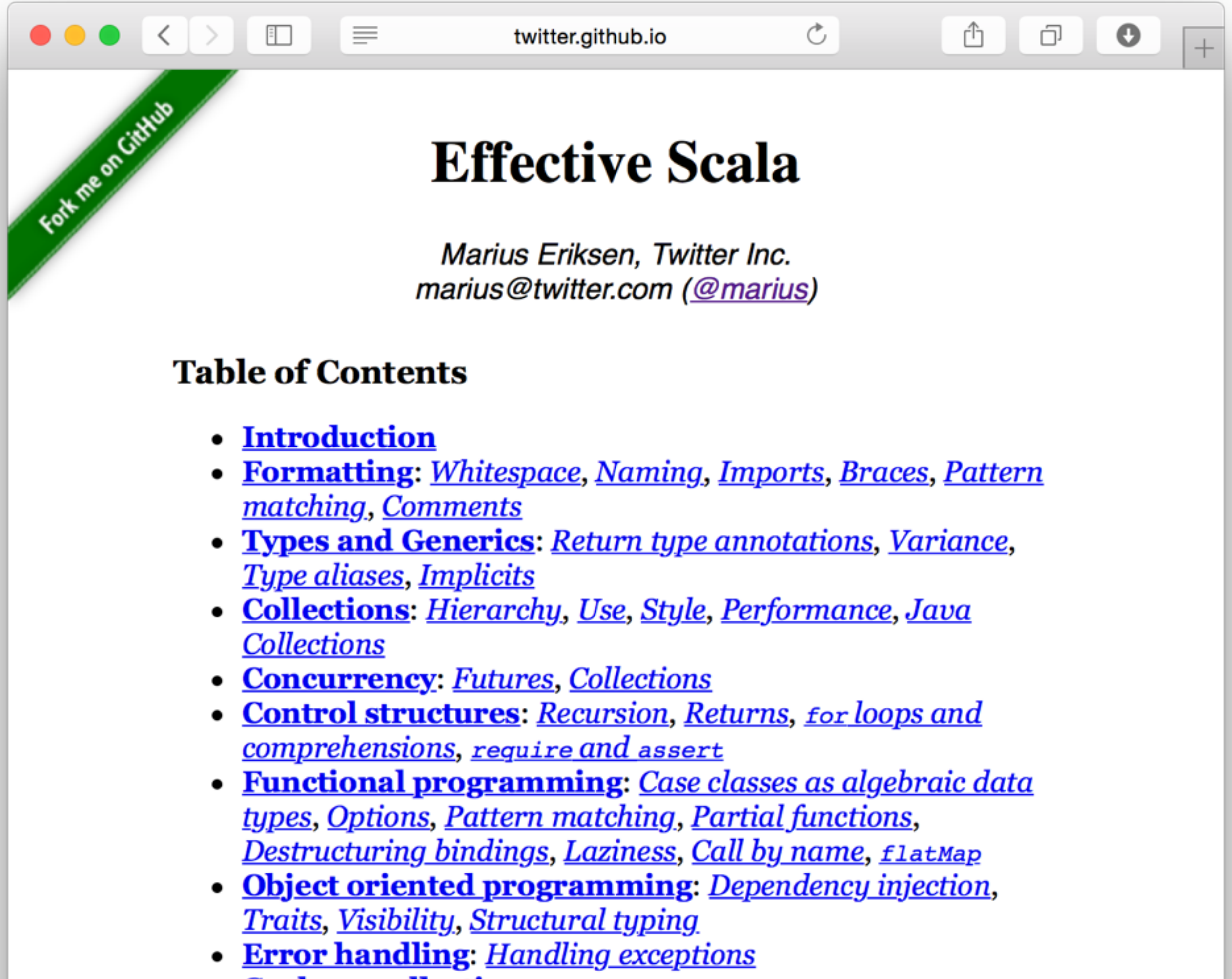
Our view is utilitarian: we want wield Scala as a useful tool; it's a means to an end.

Usage

Internal style guide, focused mainly on the semantic level.

Formatting is important but we mostly adhere to the official recommendations.

- Usage Guide
  - Nullary method application
  - Implicits
  - Call by name
  - Functions from methods and eta-expansion
  - Avoid catch-all exception handling code
  - Try vs try
  - Keep visibility as narrow as possible
  - Composition over inheritance
  - Dependency injection
  - Tuples vs named structs
  - Micro-benchmarking
  - Destructuring bindings
  - Dereference syntax and method chaining



twitter.github.io

Fork me on GitHub

Effective Scala

Marius Eriksen, Twitter Inc.
marius@twitter.com ([@marius]())

Table of Contents

Usage

Restrict feature set, e.g.,

- no structural types;
- prefer eta-expansion of methods;
- limited function literal syntax;
- composition over inheritance;
- limit “scalaz-style” programming;
- use common libraries and frameworks;
- no/limited DSLs;
- etc.

Culture

This is *extremely* important.

- Nobody can tell engineers what to do, but we can establish a culture and technical tradition.

Build teams — "infect" them.

- Don't let splinter teams/cultures/traditions evolve.

Not entirely successful at Twitter:

- Runtime systems, analytics systems, frontend systems.

Be *helpful*

We always try to be available to:

- answer “how to” questions;
- resolve usage concerns;
- discuss usage questions; and
- just discuss Scala generally.

Multiple forums:

- HipChat
- Google group/mailing list
- Code reviews
- Tech talks

Teach

New hire orientation

Scala, and related classes:

- Beginning Scala
- Finagle; concurrency
- Advanced Scala Type System
- Functional programming in Scala
- Performant Scala

Tech talks

Tooling

Large-scale refactoring

- One-off compiler plugins

IntelliJ project generation

Build artifact caching

Intelligent CI

Build hygiene

Much of the build pain is, in a sense, self-inflicted:
"Doctor it hurts when I do this."

Make building easier: Use fine-grained packages without circular dependencies.

Maybe we can even "seal" packages?


Closing
Thoughts

marius eriksen
@marius

Follow

Don’t put so much faith in the ability of any programming language to solve all your software engineering problems.

9:01 PM - 1 Jul 2014

36 39

Scala is fancy, expressive

Scala is a fancy, expressive language. You can easily do fancy, very expressive things in it.

- It’s a *very* useful tool; but it’s also a very sharp one.
- Most of the time, we’re better off using a small subset of the language.
- Don’t bring a nuclear weapon to a knife fight.

On brevity

Brevity is a double-edged sword. Used correctly, it can enhance clarity; used badly, it can serve to obscure.

With Scala, we’re constantly tempted by power, and we too often succumb to it.

We want this power, but not all the time! KISS.

On abstraction

“The curse of a very powerful and regular language is that it provides no barriers against over-abstraction.” —Martin Odersky

“The purpose of abstracting is *not* to be vague, but to create a new semantic level in which one can be absolutely precise.” —Edsger W. Dijkstra

Finally, some pithy rules

1. Introduce abstraction when it increases precision; when it serves to clarify.
2. Brevity is not the goal; clarity is.
3. Rich data structures are overrated.
4. Consistency, familiarity, and predictability are the most important traits of code. Have empathy for the user.
5. Write books; not poems.

Thanks