Pragmatic

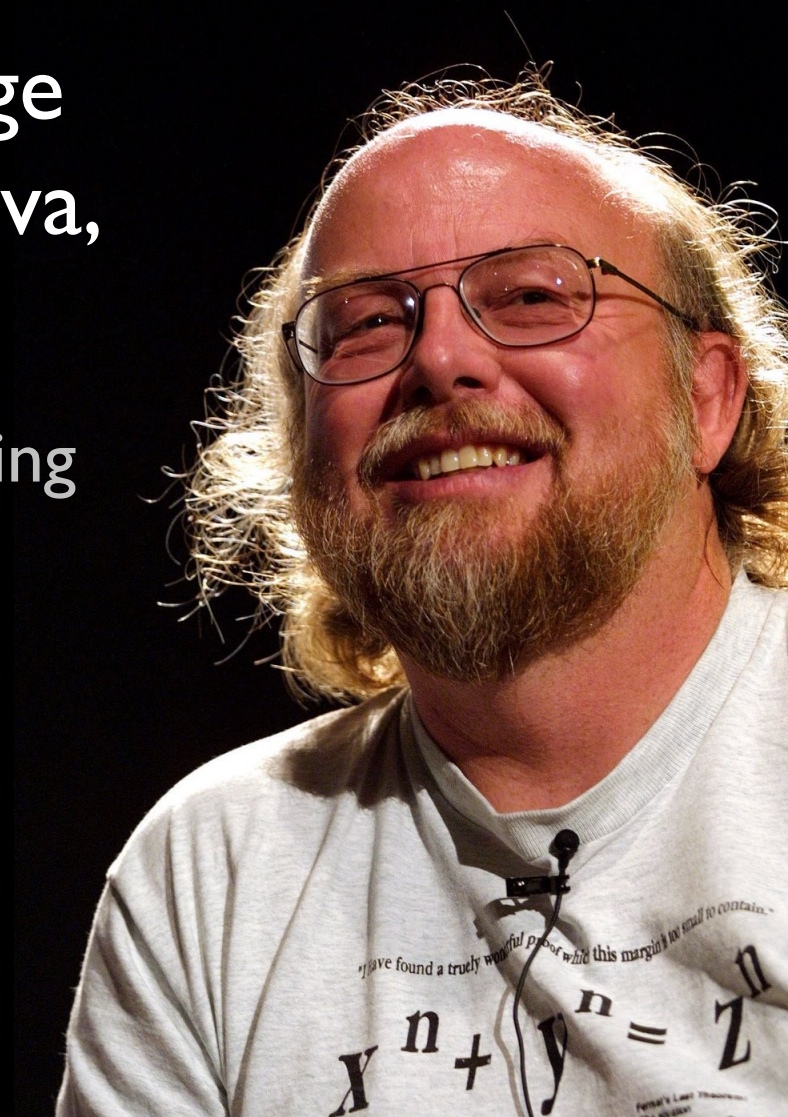
Real-World Scala

Jonas Bonér

Scalable Solutions

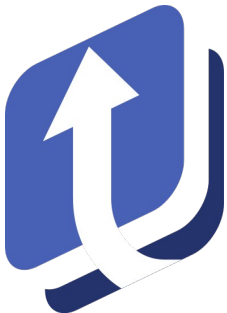
“If I were to pick a language to use today other than Java, it would be Scala”

James Gosling



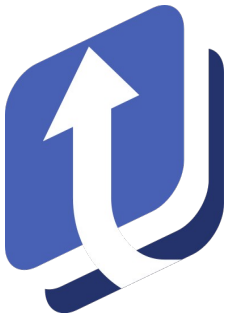
Chat app in Lift

- Build a multi-user, comet-based chat app
- About 30 lines of code
- Three slides worth
- Slides By David Pollak, creator of Lift



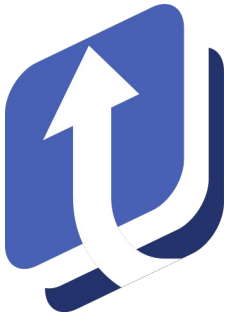
Define Messages

```
case class Add(who: Actor)
case class Remove(who: Actor)
case class Messages(msgs: List[String])
```



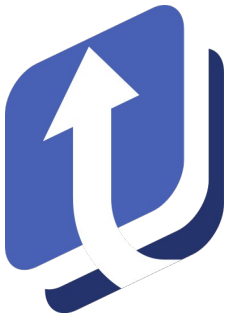
Chat Server

```
object ChatServer extends Actor {
  private var listeners: List[Actor] = Nil
  private var msgs: List[String] = Nil
  def act = loop {
    react {
      case s: String =>
        msgs = s :: msgs
        listeners.foreach(l => l ! Messages(msgs))
      case Add(who) =>
        listeners = who :: listeners
        who ! Messages(msgs)
      case Remove(who) => listeners -= who
    }
  }
  this.start
}
```



Chat Comet Component

```
class Chat extends CometActor {
  private var msgs: List[String] = Nil
  def render =
    <div>
      <ul>{ msgs.reverse.map(m => <li>{ m }</li>) }</ul>
      { ajaxText("", s => { ChatServer ! s; Noop }) }
    </div>
  override def localSetup = ChatServer ! Add(this)
  override def localShutdown = ChatServer ! Remove(this)
  override def lowPriority = {
    case Messages(m) => msgs = m; reRender(false)
  }
}
```



Demo

Scalable language

OO + FP

Martin Odersky

Pragmatic

Since 2003

Sponsored by EPFL

Seamless Java interoperability

Friendly and supportive community

Production ready

Runs on the JVM

Statically typed

is...

Expressive & light-weight

```
val phonebook = Map(
  “Jonas” -> “123456”,
  “Sara”  -> “654321”)
phonebook += (“Jacob” -> “987654”)
println(phonebook(“Jonas”))
```

High-level

Java version

```
boolean hasUpperCase = false;
for (int i = 0; i < name.length(); i++) {
  if (Character.isUpperCase(name.charAt(i)))
  {
    hasUpperCase = true;
    break;
  }
}
```

High-level

Scala version

```
val hasUpperCase = name.exists(_.isUpperCase)
```

Concise

```java
// Java
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(age: int) {
  this.age = age;
 }
}
```

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

Pure OO

```
1 + 2
1.+(2)
```

```
123.toString()
```

Extensible

```
val service = actor {
  loop {
    receive {
      case Add(x,y) => reply(x+y)
      case Sub(x,y) => reply(x-y)
    }
  }
}
service ! Add(4, 2)    ➔ 6
```

Pragmatic

```
def users =
  <users>
    <user role=”customer”>
      <name>{ user.name }</name>
      <password>{ user.password }</password>
      <email>{ user.email }</email>
    </user>
    ...
  </users>
```

Pragmatic

```
users match {
  case <users>{users @ _*}</users> =>
    for (user <- users)
      println("User " + (user \ "name").text)
}
```

Great for DSLs

Apache Camel

```
“direct:a“ ==> {
  loadbalance roundrobin {
    to (“mock:a“)
    to (“mock:b“)
    to (“mock:c“)
  }
}
```

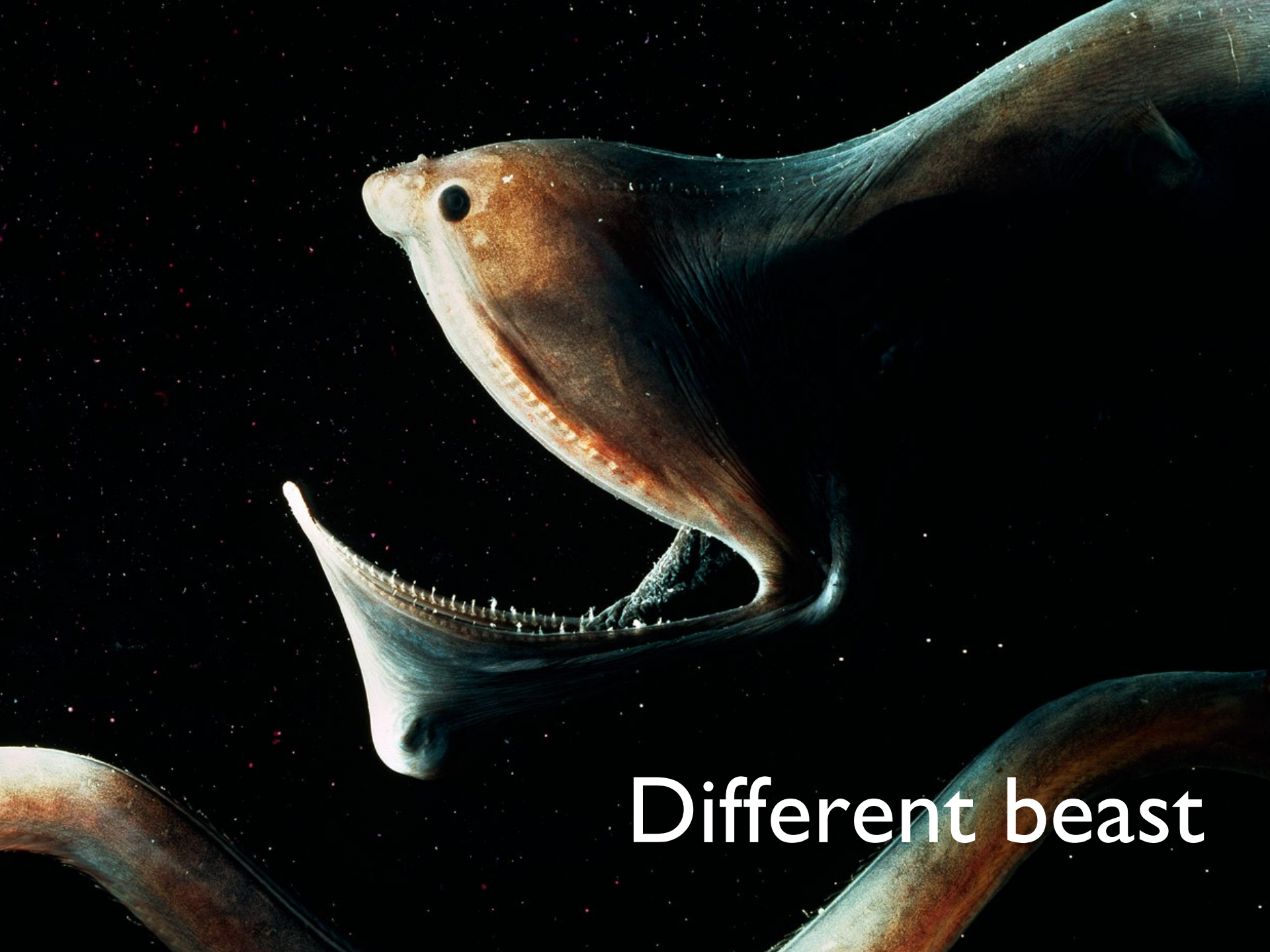
Scala is

deep

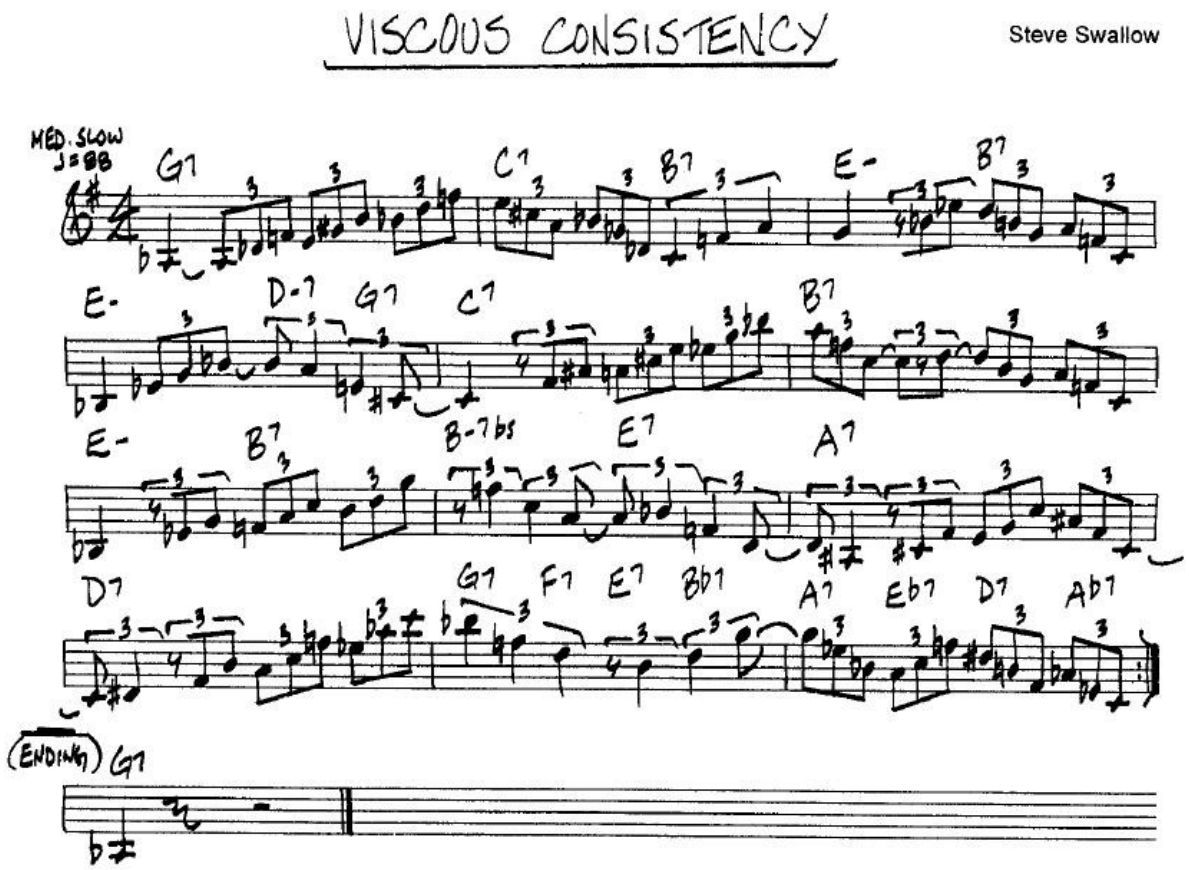
But you don't need to go very deep to still have fun and be productive

Java


Tastier Java


Different beast

Composition



Composition

in large

2

building blocks

Traits

&

Self-type annotations

Trait

```
trait Dad {
  private var children: List[Child] = Nil

  def addChild(child: Child) =
    children = child :: children

  def getChildren = children.clone
}
```

Base

```
class Man(val name: String) extends Human
```


Plumbing

Static mixin composition

```
class Man(val name: String) extends Human with Dad
```

Static mixin composition usage

```
class Man(val name: String) extends Human with Dad
```

```
val jonas = new Man("Jonas")
jonas.addChild(new Child("Jacob"))
```

Dynamic mixin composition

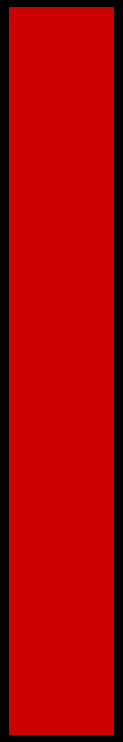
```
val jonas = new Man("Jonas") with Dad
```

Dynamic mixin composition usage

```
val jonas = new Man("Jonas") with Dad
```

```
jonas.addChild(new Child("Jacob"))
```

different type of traits



Rich interface

```
trait RichIterable[A] {
  def iterator: Iterator[A] // contract method

  def foreach(f: A => Unit) = {
    val iter = iterator
    while (iter.hasNext) f(iter.next)
  }

  def foldLeft[B](seed: B)(f: (B, A) => B) = {
     var result = seed
     foreach(e => result = f(result, e))
     result
   }
}
```

Rich interface

```
val richSet =
    new java.util.HashSet[Int]
    with RichIterable[Int]

richSet.add(1)
richSet.add(2)

richSet.foldLeft(0)((x, y) => x + y)
  ➔ 3
```

2

Stackable modifications

```
trait IgnoreCaseSet
  extends java.util.Set[String] {

  abstract override def add(e: String) = {
    super.add(e.toLowerCase)
  }
  abstract override def contains(e: String) = {
    super.contains(e.toLowerCase)
  }
  abstract override def remove(e: String) = {
    super.remove(e.toLowerCase)
  }
}
```

Stackable modifications

```
val set =
    new java.util.HashSet[String]
    with IgnoreCaseSet

set.add("HI THERE")       // uppercase

set.contains("hi there") // lowercase
    ➔ true
```

Add another trait interceptor

```
trait LoggableSet
  extends java.util.Set[String] {

  abstract override def add(e: String) = {
    println(“Add :“ + e)
    super.add(e)
  }

  abstract override def remove(e: String) = {
    println(“Remove :“ + e)
    super.remove(e)
  }
}
```

Run the stack of interceptors

```
val set =
  new java.util.HashSet[String]
  with IgnoreCaseSet
  with LoggableSet

set.add("HI THERE")
  ➔ "Add: HI THERE"
```

Prints in uppercase

Change the order

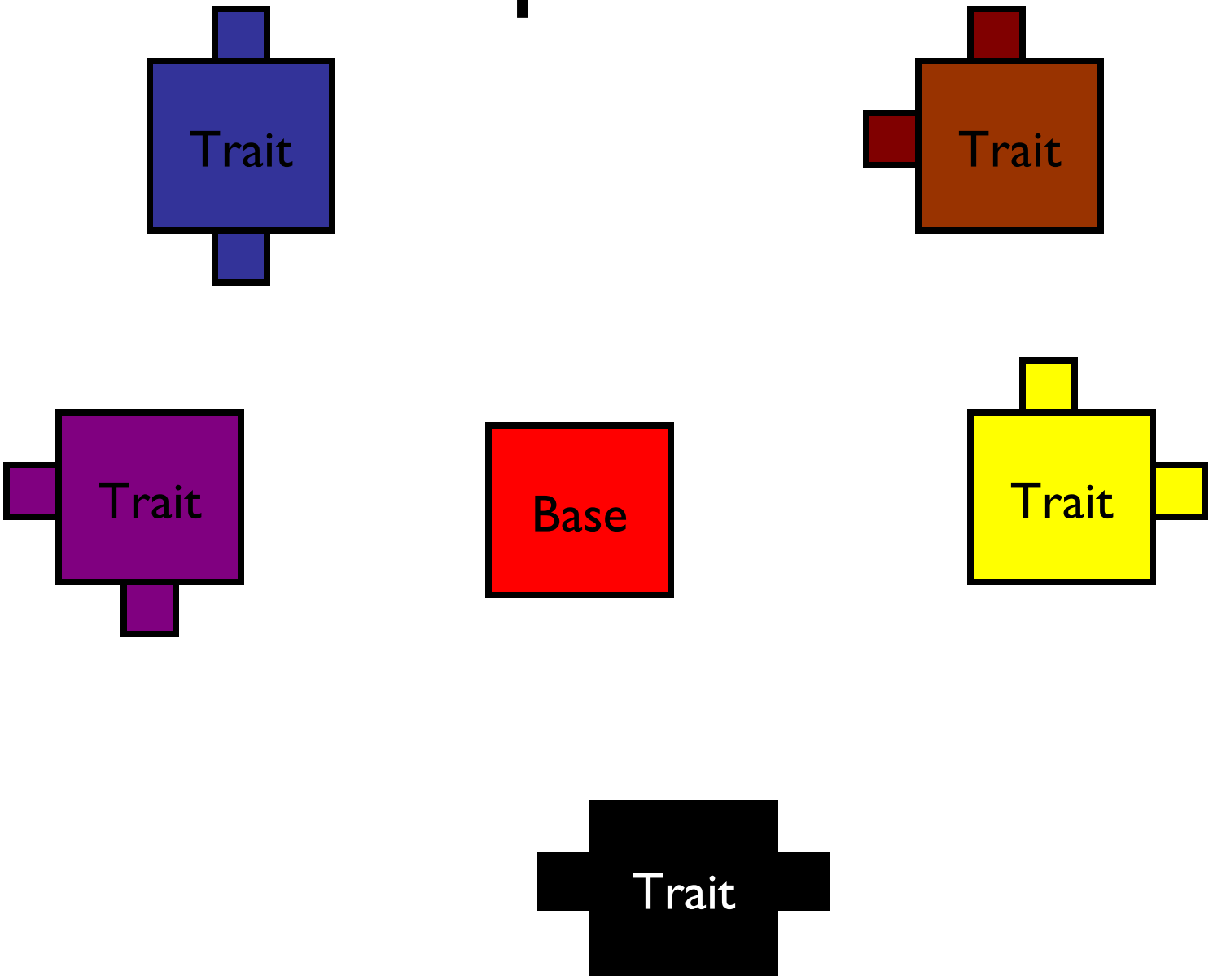
```
val set =
  new java.util.HashSet[String]
  with LoggableSet
  with IgnoreCaseSet

set.add("HI THERE")
  ➔ "Add: hi there"
```

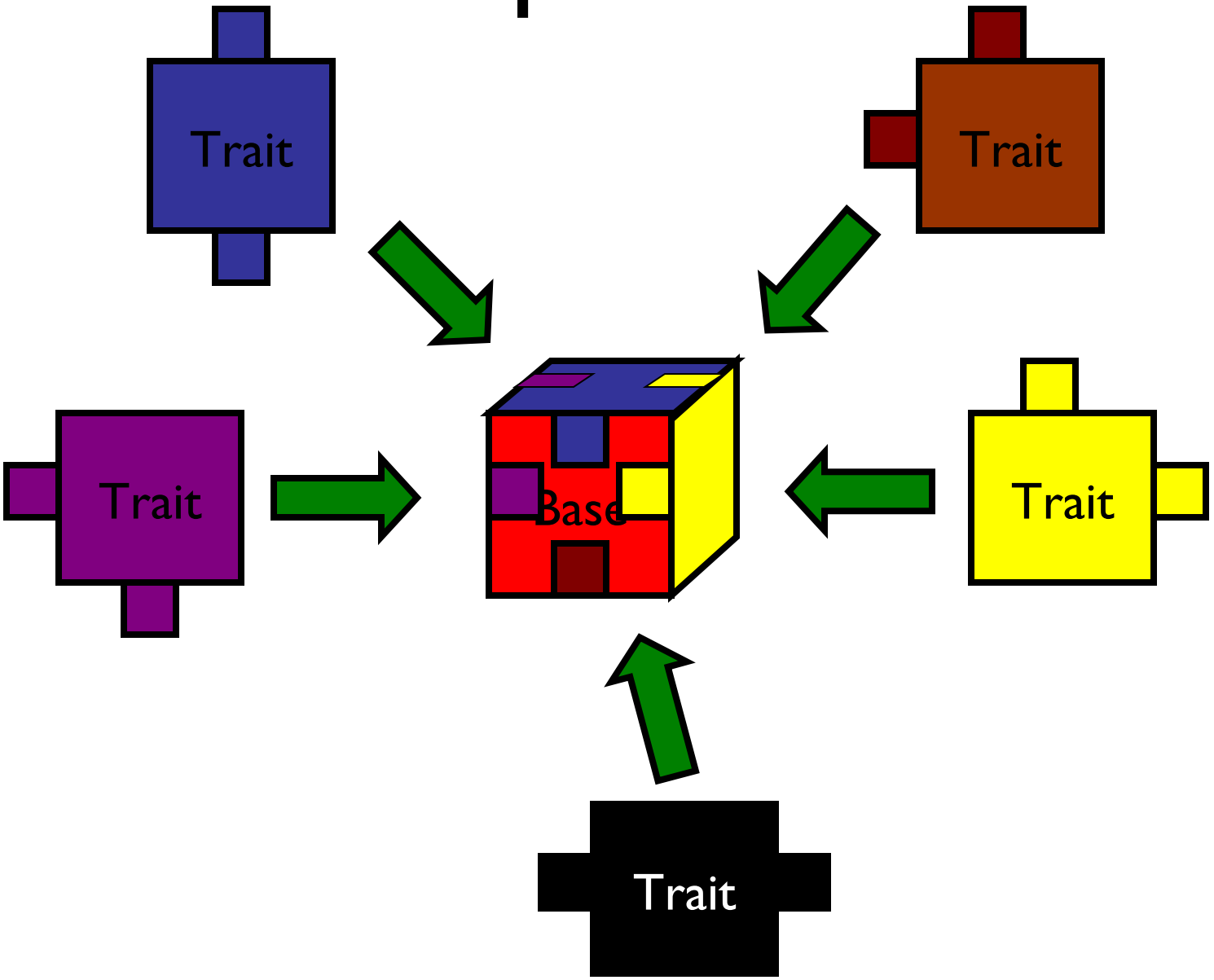
Prints in lowercase

3

Multiple views



Multiple views



Traits

Multiple
personalities

Traits

Base

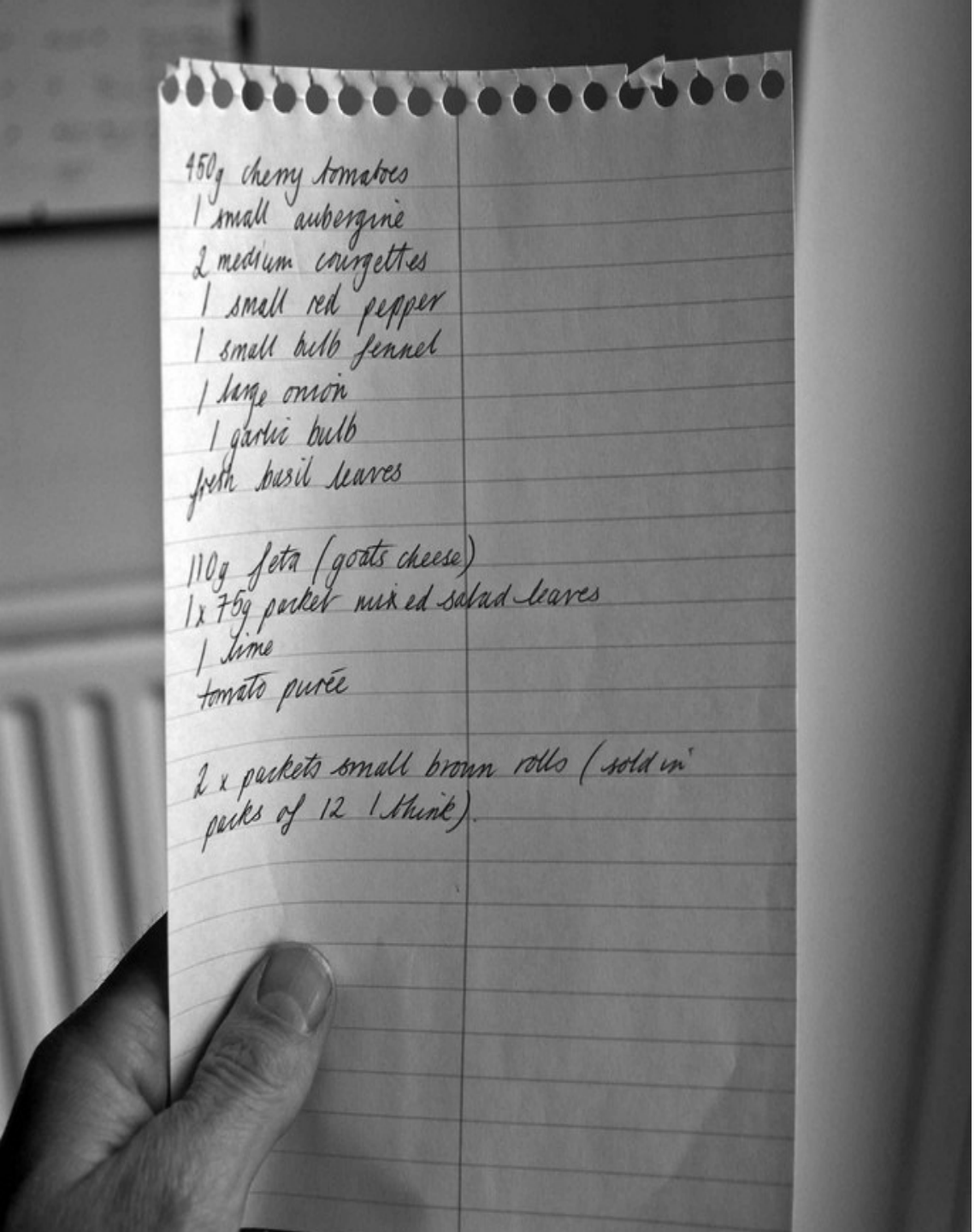
```
class Order(val cust: Customer)
```

Facets

```
trait Entity { ... }
trait InventoryItemSet { ... }
trait Invoicable { ... }
trait PurchaseLimiter { ... }
trait MailNotifier { ... }
trait ACL { ... }
trait Versioned { ... }
trait Transactional { ... }
```

Composition

```
val order = new Order(customer)
    with Entity
    with InventoryItemSet
    with Invoicable
    with PurchaseLimiter
    with MailNotifier
    with ACL
    with Versioned
    with Transactional
```



What do you need?

DI

```
this: <deps> =>
```

```
this: <deps> =>
```

Dependency declaration

```
trait UserService {
  this: UserRepository =>
  ...
  // provided with composition
  userRepository.merge(user)
}
```

…using self-type annotation

scala dependency injection

Google Search | I'm Feeling Lucky

Advanced Search
Preferences
Language Tools

Advertising Programs - Business Solutions - About Google - **Go to Google Sverige**


Duck typing

Duck typing

Structural Typing:

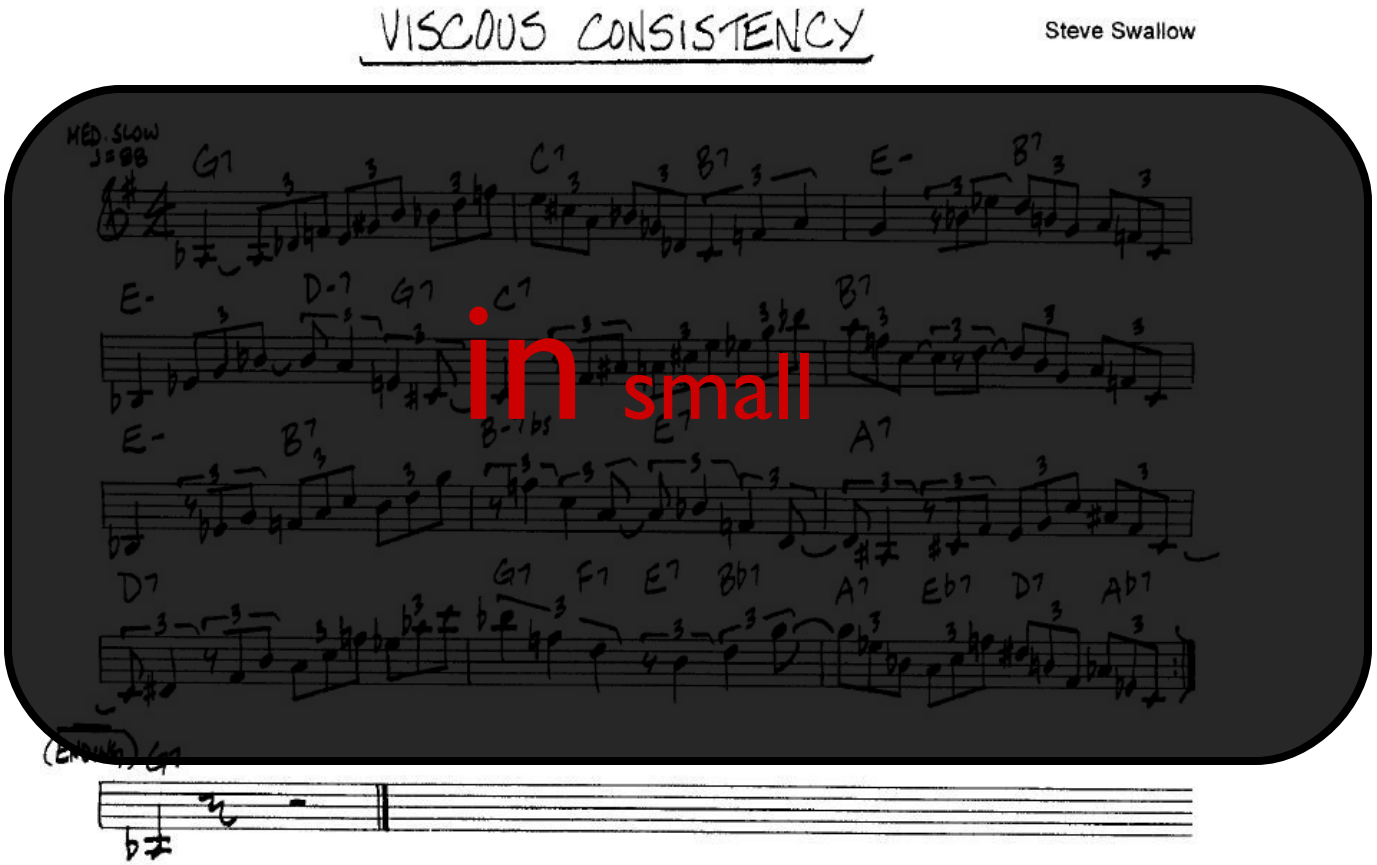
Duck-typing done right

```
def authorize(target: { def getACL: ACL }) = {

  val acl = target.getACL
  ... //authorize
}
```

Statically enforced

Composition



“It's Time to Get Good at Functional Programming”

Dr Dobb’s Journal

Issue December 2008

What's all the buzz about?

Deterministic

Reusable

High-level

FP

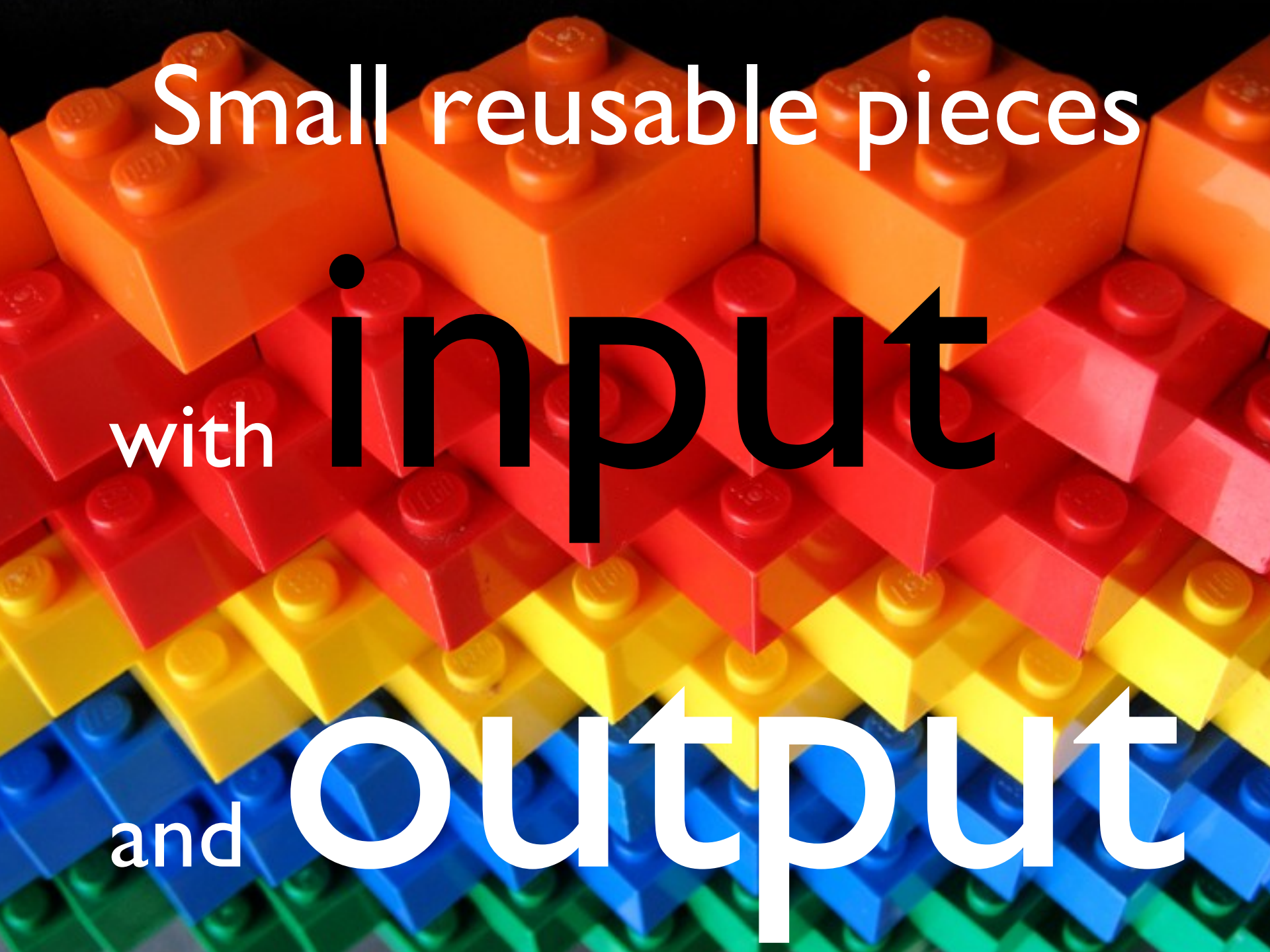
Referentially transparent

Immutable

Declarative

Orthogonal


FP is like
Lego


Small reusable pieces
with input
and output

Unix pipes

```
cat File | grep 'println' | wc
```

Functions

```
(x: Int) => x + 1
```

Functions as values

```
val inc = (x: Int) => x + 1
```

```
inc(1)  ➔ 2
```

Functions as parameters

high-order

```
List(1, 2, 3).map((x: Int) => x + 1)
```

➔ `List(2, 3, 4)`

Functions as parameters

with sugar

```
List(1, 2, 3).map((x: Int) => x + 1)
```

```
List(1, 2, 3).map(x => x + 1)
```

```
List(1, 2, 3).map(_ + 1)
```

Functions as closures

```
val addMore = (x: Int) => x + more
```

What is more?

Some value outside the function's lexical scope

```
var more = 7
val addMore = (x: Int) => x + more
```

```
addMore(3) ➔ 10
more = 8
addMore(3) ➔ 11
```

Functional

Data Structures

Immutable

The almighty

List

Lists

Grow at the front

Insert at front ➔ O(1)

Insert at end ➔ O(n)

List creation

```
List(1, 2, 3)
```

```
1 :: 2 :: 3 :: Nil
```

Basics

```
val list = List(1, 2, 3)
list.head    ➔ 1
list.tail    ➔ List(2, 3)
list.isEmpty ➔ false
```

High-level operations

```
val list = List(1, 2, 3)
list.map(_ + 1)       ➔ List(2, 3, 4)
list.filter(_ < 2)    ➔ List(3)
list.exists(_ == 3)   ➔ true
list.drop(2)          ➔ List(3)
list.reverse          ➔ List(3, 2, 1)
list.sort(_ > _)      ➔ List(3, 2, 1)
List.flatten(list)    ➔ List(1, 2, 3)
list.slice(2, 3)      ➔ List(3)
...
```

115

functions defined on List

Tuples

```
def getNameAndAge: Tuple2[String, Int] = {
  val name = ...
  val age = ...
  (name, age)
}
```

```
val (name, age) = getNameAndAge
println("Name: " + name)
println("Age: " + age)
```

Other

functional data structures:

Maps

Sets

Trees

Stacks

For comprehensions

```
for (n <- names)
  println(n)
```

Like SQL queries

```
for {
   att <- attendees
   if att.name == “Fred”
   lang <- att.spokenLanguages
   if lang == “Danish”
} println(att)
```

Find all attendees named Fred that speaks Danish

for / yield

```
val companiesForAttendeesFromLondon =
  for {
    att <- attendees
    if att.address.city == “London”
  } yield att.company
```

Everything returns a value

```
for (thing <- thingsFromHere)
yield getRealThing(thing)
```

Everything returns a value

```
val things =
  for (thing <- thingsFromHere)
  yield getRealThing(thing)
```

Everything returns a value

```
if (fromHere) {
  for (thing <- thingsFromHere)
  yield getRealThing(thing)
} else {
  for (thing <- thingsFromThere)
  yield thing
}
```

Everything returns a value

```
val things =
  if (fromHere) {
    for (thing <- thingsFromHere)
    yield getRealThing(thing)
  } else {
    for (thing <- thingsFromThere)
    yield thing
  }
```

Everything returns a value

```
try {
  if (fromHere) {
    for (thing <- thingsFromHere)
    yield getRealThing(thing)
  } else {
    for (thing <- thingsFromThere)
    yield thing
  }
} catch {
  case e => error(e); Nil
}
```

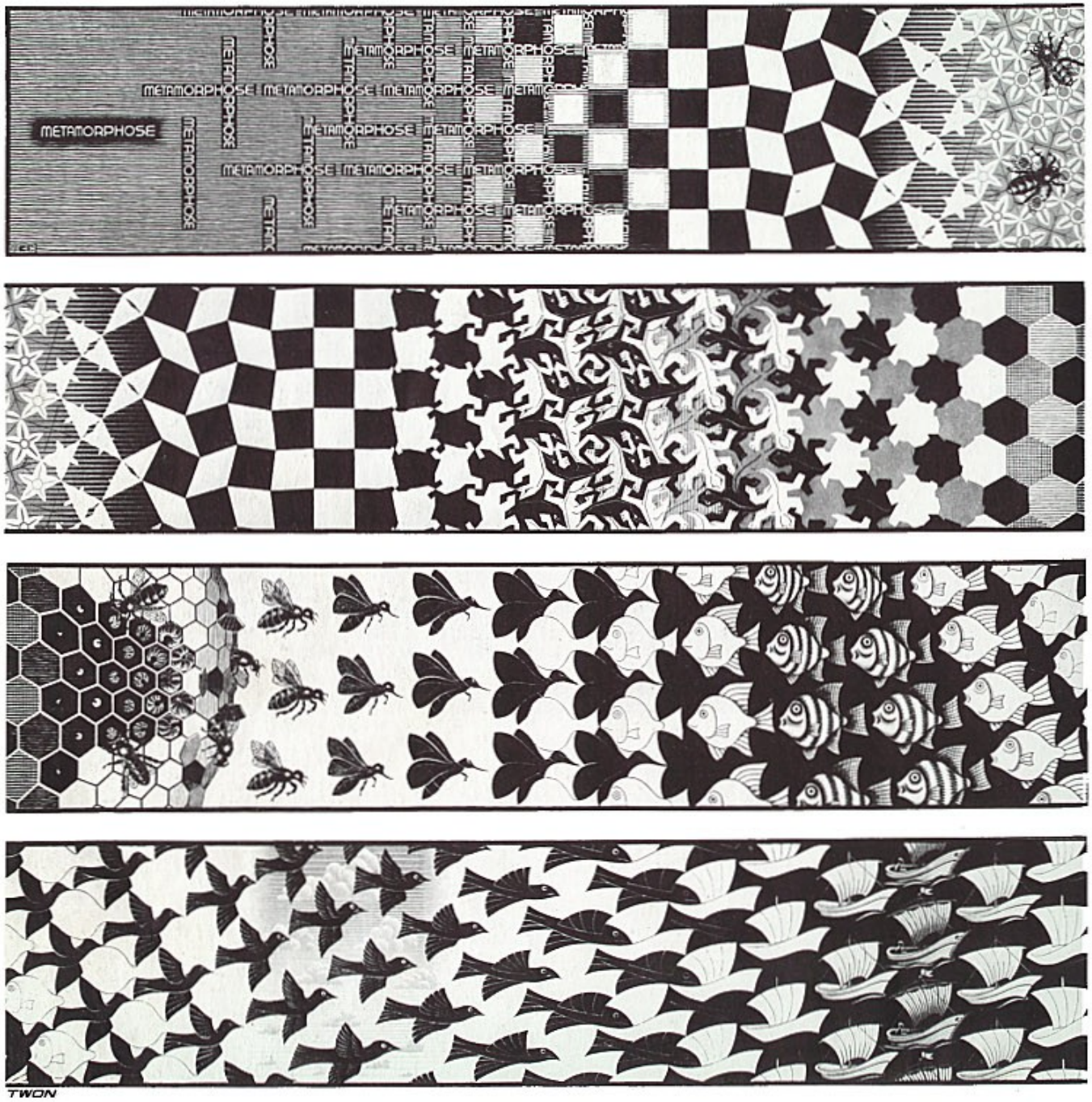
Everything returns a value

```
val things =
  try {
    if (fromHere) {
      for (thing <- thingsFromHere)
      yield getRealThing(thing)
    } else {
      for (thing <- thingsFromThere)
      yield thing
    }
  } catch {
    case e => error(e); Nil
  }
```

Everything returns a value

```
def getThingsFromSomewhere(
  fromHere: Boolean): List[Thing] = {
  try {
    if (fromHere) {
      for (thing <- thingsFromHere)
      yield getRealThing(thing)
    } else {
      for (thing <- thingsFromThere)
      yield thing
    }
  } catch {
    case e => error(e); Nil
  }
}
```

Pattern
matching



Pattern matching

```
def matchAny(a: Any): Any = a match {
  case 1                          => “one”
  case “two”                      => 2
  case i: Int                     => “scala.Int”
  case <tag>{ t }</tag>           => t
  case head :: tail               => head
  case _                          => “default”
}
```

Tools

Tools: scala/bin

scala

scalac

fsc

scaladoc

sbaz

Tools: IDEs

Eclipse

NetBeans

IntelliJ IDEA

Emacs

JEdit

Building

Maven

Ant

SBT (Simple Build Tool)

Tools: Testing

Specs

ScalaCheck

ScalaTest

SUnit

Frameworks: Web

Lift

Sweet

Slinky

Pinky

Learn more

jonas@jonasboner.com
http://jonasboner.com
http://scala-lang.org

Professional help

Consulting Training Mentoring

http://scalablesolutions.se

Pragmatic

Real-World Scala

Jonas Bonér

Scalable Solutions