DRW TRADING GROUP

polyglotprogramming.com

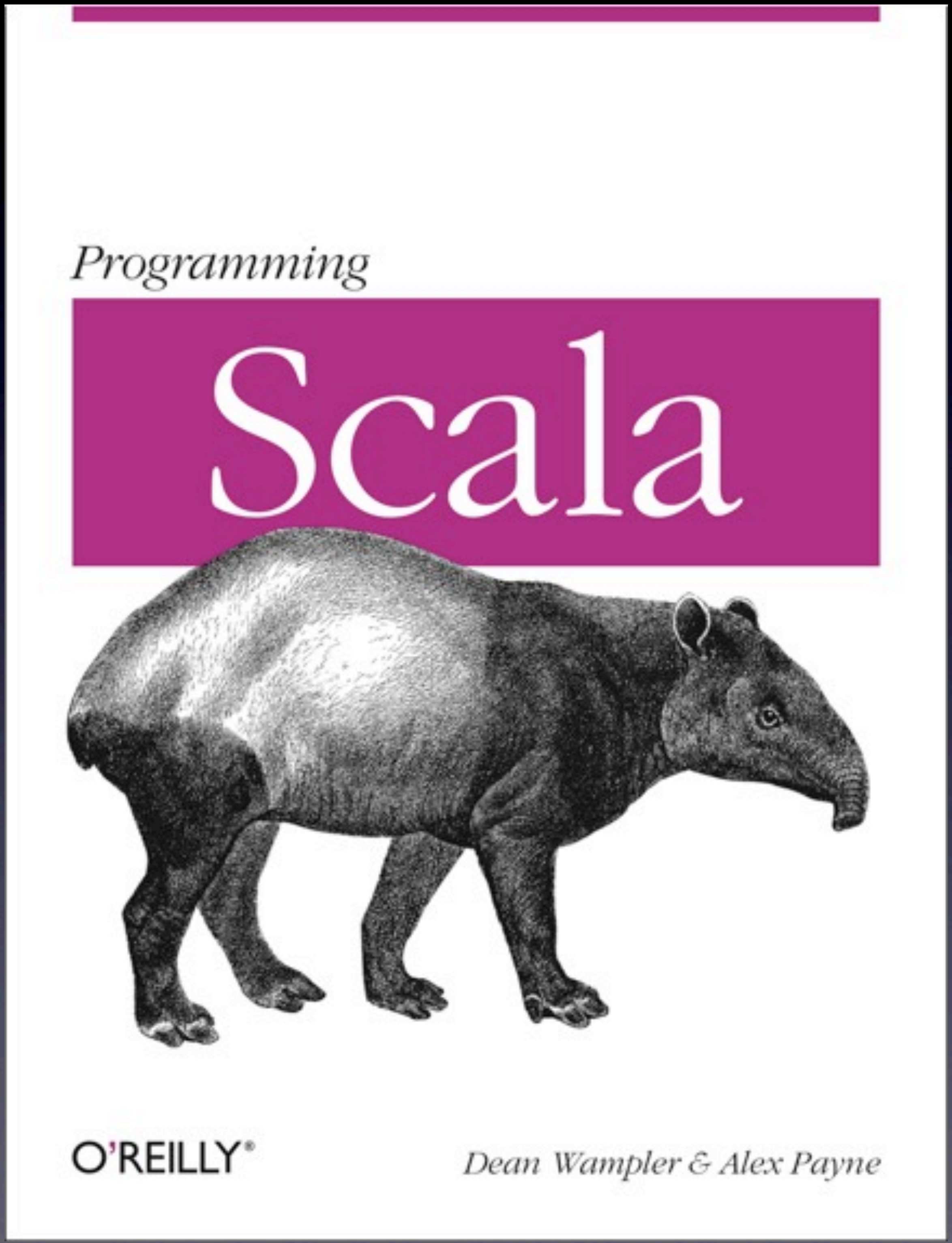
Scala for *Erlang* Programmers

Dean Wampler
dean@deanwampler.com
@deanwampler
polyglotprogramming.com/talks

<shameless-plug/>

Co-author, *Programming Scala*

programmingscala.com



IEEE Software
Building the Community of Leading Software Practitioners
www.computer.org/software

Guest Editor,
IEEE Software
Special Issue on
Multi-paradigm Programming

computer.org/software

Why Erlang?

Erlang's history is a microcosm of FP's history:

It's been *used* for *decades* by a *select few*...

… and now *everybody* is using it.

Why Erlang

now?

#1
We need *Functional Programming*.

#2
Lots of services behave like *telephony switches*.

basho

apache
CouchDB
relax

github
SOCIAL CODING

f

RabbitMQ™

•••

Why Scala?

#1
Java is
old …,

#1

… but people want to keep their JVM/.NET *investment*.

#2
The *power* of *types* *compel* you!

#3 *Composability* and *scalability* features.

#4
The *marriage* of *OOP* and *FP*.

What is Scala?

Martin Odersky

- Helped design java *generics*.
- Co-wrote *GJ* that became *javac* (v1.3+).
- Understands Computer Science *and* Industry.

Martin Odersky

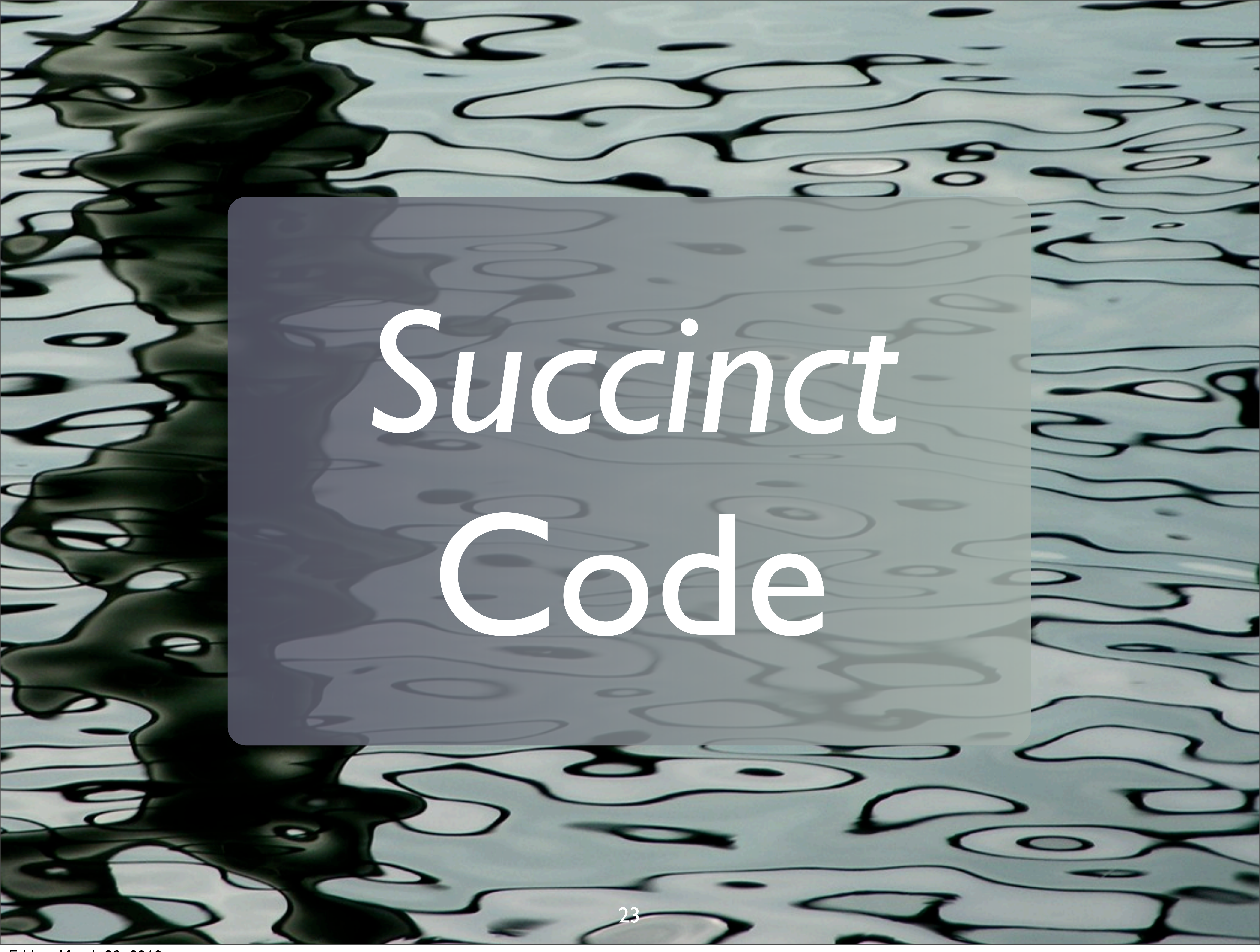
- Inspired by:
  - *Haskell*.
  - *Prolog*.
  - … and *Erlang*!

Appealing if you like:

- *Rigor*.
- Deeply thought-through *principles*.
- *Static* typing.

Not appealing if you find

- Rigor is *tedious*.
- Dynamic languages are *easier*.



Succinct Code

```
$ scala
Welcome to Scala version 2.7.7 ...

scala> "hello" + "world"
res0: java.lang.String = helloworld

scala> "hello".+("world")
res1: java.lang.String = helloworld
```

```
object.method(arguments)
```

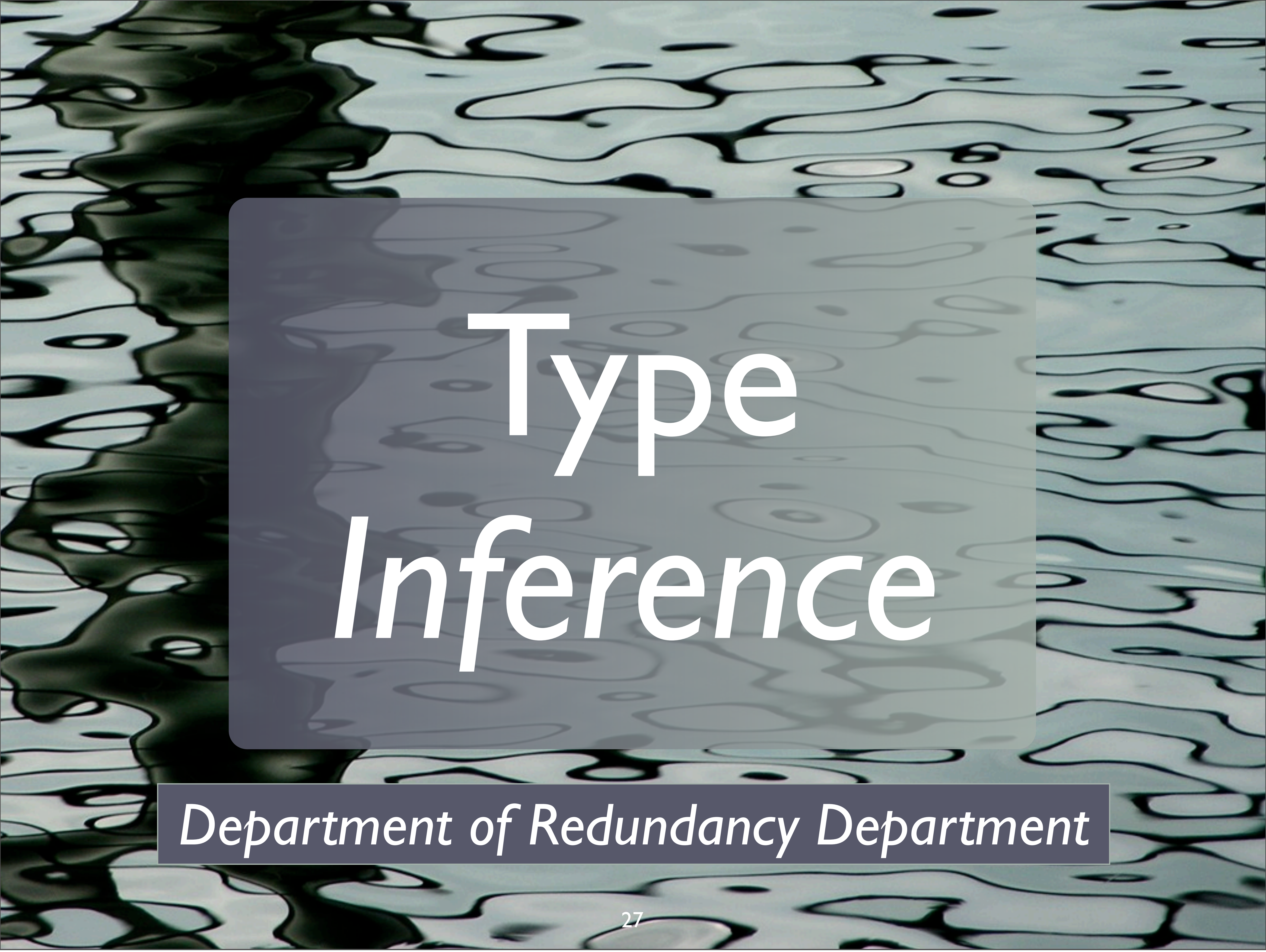
same as

```
object method arguments
```

Method Names

Almost any character allowed

pseudo operator overloading.



Type *Inference*

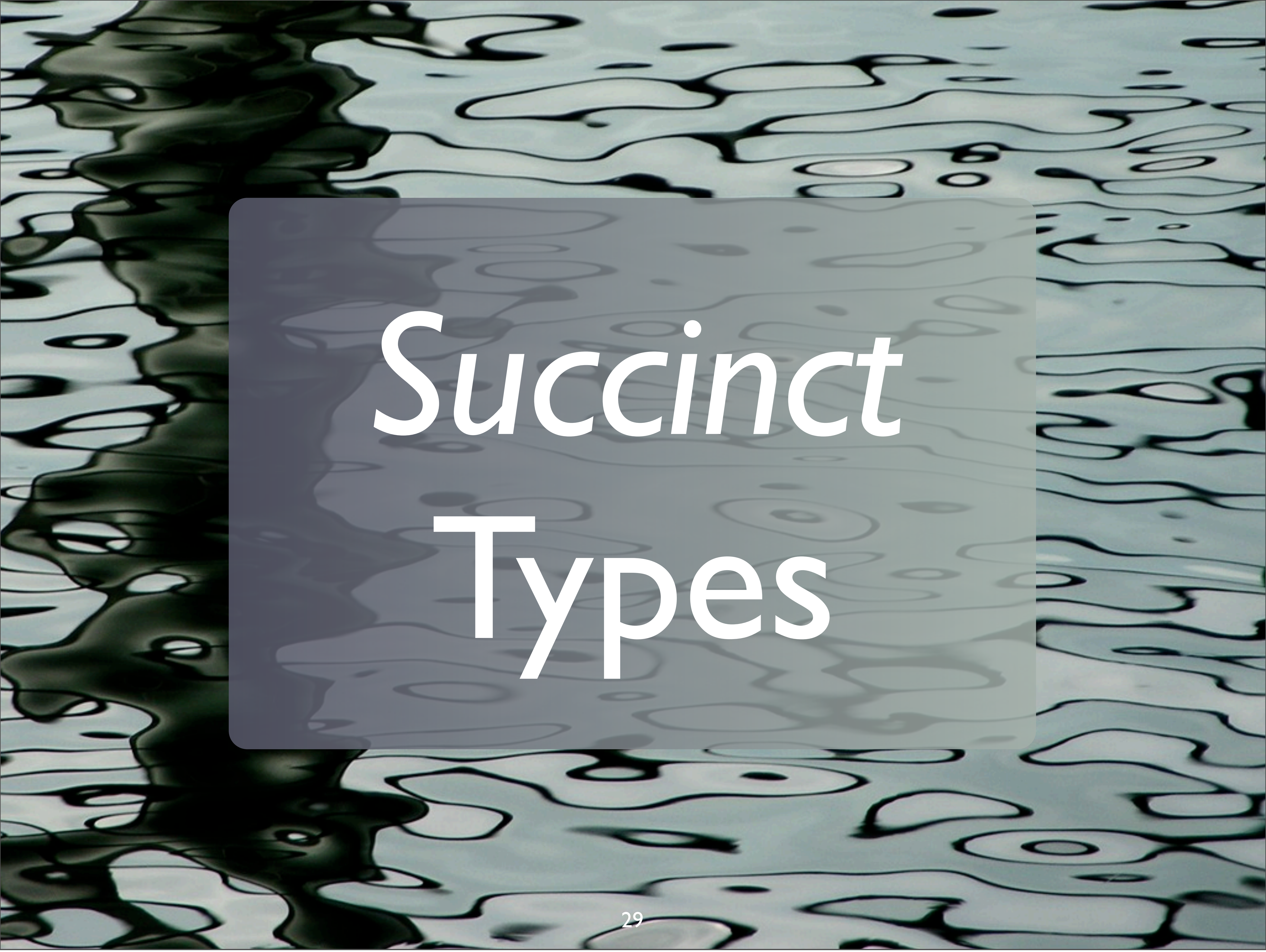
Department of Redundancy Department

Type *inferencing* in *Scala*

Read-only "variable"

```
val name = “Dean Wampler”
val map = Map(“name” -> “Dean”,
              “age” -> 29, …)
val tuple = (0, “two”, 3.14159)
var count = 0
```

Read-write variable



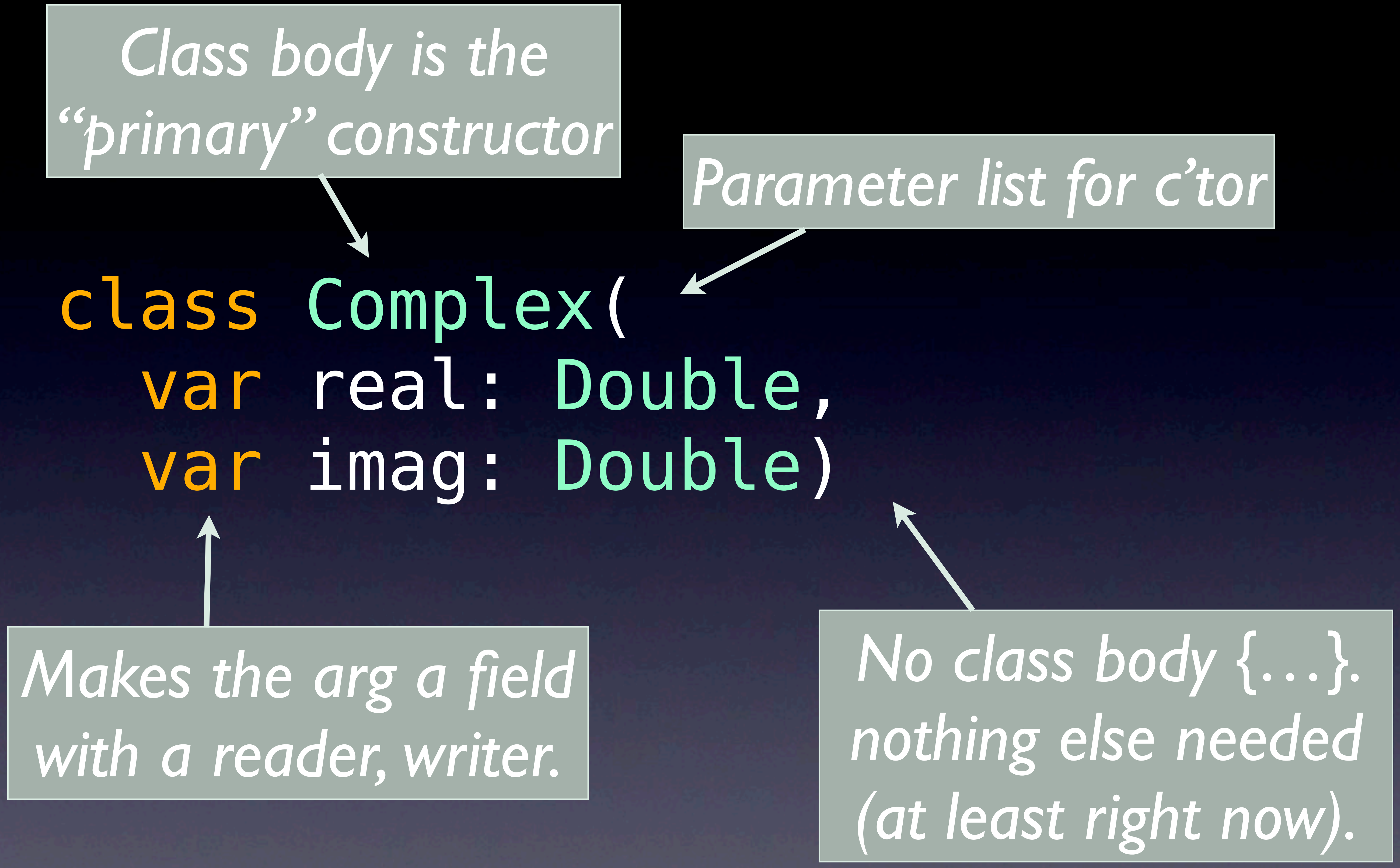
Succinct Types

Typical Java.

```
class Complex {
  private double real;
  private double imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }

  public void double getReal() {return this.real;}
  public void setReal(double real) {
    this.real = real;
  }

  public void double getImag() {return this.imag;}
  public void setImag(double imag) {
    this.imag = imag;
  }
}
```

Typical Scala!

```
class Complex(
  var real: Double,
  var imag: Double)

val c = new Complex(1.2, 3.4)
```



Class body is the "primary" constructor

Parameter list for c'tor

```
class Complex(
  var real: Double,
  var imag: Double)
```

Makes the arg a field with a reader, writer.

No class body {…}. nothing else needed (at least right now).

Should Be *Immutable*

```
class Complex(
  val real: Double,
  val imag: Double)
```

Make the objects immutable!!

Case Classes

```
case class Complex(
  ~~val~~ real: Double,
  ~~val~~ imag: Double)

val c = ~~new~~ Complex(1.2, 3.4)
```

More succinct.

Case Classes

```
case class Complex(
        real: Double,
        imag: Double)


val c = Complex(1.2, 3.4)
```

More succinct.

Default Values

```
case class Complex(
      real: Double = 0.0,
      imag: Double = 0.0)

val c = Complex(1.0) // real
val zero = Complex()
```

Scala v2.8

Erlang Records

To me, these types of classes feel a lot like Erlang Records.

Scala gives you nice type checking.

Arguments for *Static Types*

- Compile-time *error checking*.
- Run-time *optimizations*.

User-defined *Operators*

… and our own datatypes.

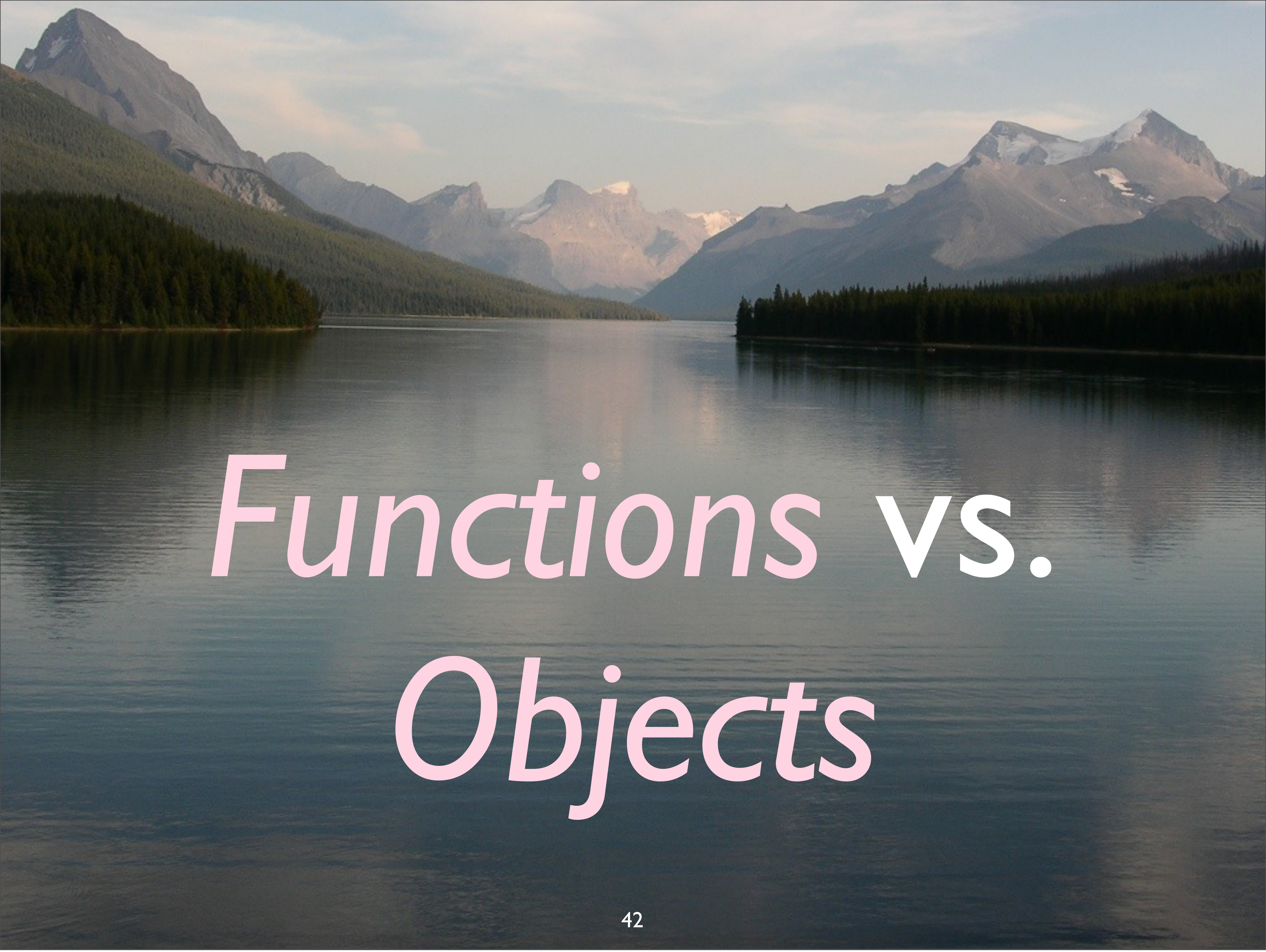
```
case class Complex(real: Double,
                   imag: Double)
{
  def +(that: Complex): Complex =
    Complex(real + that.real,
            imag + that.imag)

  def -(that: Complex): Complex =
    Complex(real - that.real,
            imag - that.imag)

  …
}
```

“operators”

“Operator overloading”

```
var c1 = Complex(1.2, 3.4)
val c2 = Complex(4.3, 2.1)

c1 + c2    // => (5.5, 5.5)
c1 += c2   // same as c1 = c1+c2
c1 - c2    // => (-3.1, 1.3)
```

Example usage



Functions vs. *Objects*

Functions in Scala:

They are *first class*...

and they are *objects*!

Function Literals

```
f: (Double,Int) => String
```

is equivalent to

```
f: Function2[Double,Int,String]
```

(or in Java-speak)

```
Function2<Double,Int,String> f
```

Interlude: Lists

Empty list

“cons” operator (method)

```
val l1 = Nil
val l2 = "c" :: l1
val l3 = "b" :: l2
val l4 = "a" :: l3

// => List(”a”,”b”,”c”)
```

Example

```
def listmap[A,B](l: List[A]) (
 f: A => B): List[B] = l match {

  case head :: tail =>
   f(head) :: listmap(tail)(f)

  case Nil => Nil
}
```

```
def listmap[A,B](l: List[A]) (
 f: A => B): List[B] = l match {
   case head :: tail =>
     f(head) :: listmap(tail)(f)

   case Nil => Nil
}
```

type params

2 arg lists

return type

pattern match

match on non-Nil list

match on Nil

Try it out:

```
val l1 = List(“1”, “2”, “3”)
val l2 = listmap(l1) { s =>
  val i = Integer.parseInt(s)
  i*i
}

// => List(1, 4, 9)
```

2nd argument: function literal

Point-Free Style

... sometimes works.

Using List.map

```
val l2 = List(1, 4, 9)
def square(i: Int) = i * i
l2 map square
// => List(1, 16, 81)
l2 map square map square
// => List(1, 256, 6561)
```

Objects

… can also be *functions*!

When we use case:

“singleton”

“Factory”

```
object Complex {
  def apply(real: Double,
            imag: Double) =
    new Complex(real + that.real,
                imag + that.imag)

  …
}
val c = Complex(1.1, 2.2)
```

```
class Logger(val level:…) {
  def apply(message: String) =
  { // pass to logging system
    log(level, message)
  }
}
val error = new Logger(ERROR)
…
error(“Network error.”)
```

“function object”

Traits

Composable Units of Behavior

Functional Languages

Get composition through higher-order functions.

Java

```
class Queue
 extends Collection
 implements Logging, Filtering
{ … }
```

Java's object model

- *Good*
  - Promotes abstractions.
- *Bad*
  - No *composition* through reusable *mixins*.

Traits

Like interfaces with implementations,

Traits

… or like

abstract classes +

multiple inheritance

(if you prefer).

Example

```
trait Queue[T] {
  def get(): T
  def put(t: T)
}
```

A pure abstraction (in this case...)

```
class StandardQueue[T]
          extends Queue[T] {
  import ...ArrayBuffer
  private val ab =
        new ArrayBuffer[T]
  def put(t: T) = ab += t
  def get() = ab.remove(0)
  …
}
```

Concrete (boring) implementation

Log put

```
trait QueueLogging[T]
 extends Queue[T] {
  abstract override def put(
    t: T) = {
   println("put: "+t)
   super.put(t)
  }
}
```

Log put

```
trait QueueLogging[T]
 extends Queue[T] {
  abstract override def put(
     t: T) = {
    println("put: "+t)
    super.put(t)
   }
}
```

What is "super" bound to??

```
val sq = new StandardQueue[Int]
        with QueueLogging[Int]

sq.put(10)
// => put: 10
sq.put(20)
// => put: 20
```

Example

Mixin composition; no class required

```
val sq = new StandardQueue[Int]
        with QueueLogging[Int]

sq.put(10)
// => put: 10
sq.put(20)
// => put: 20
```

Example

Stackable Traits

Filter put

```
trait QueueFiltering[T]
 extends Queue[T] {
  abstract override def put(
    t: T) = {
   if (veto(t))
     println(t+" rejected!")
   else
     super.put(t)
  }
  def veto(t: T): Boolean
}
```

Filter put

```
trait QueueFiltering[T]
 extends Queue[T] {
  abstract override def put(
    t: T) = {
   if (veto(t))
     println(t+" rejected!")
   else
     super.put(t)
  }
  def veto(t: T): Boolean
}
```

“Veto” puts

```
val sq = new StandardQueue[Int]
     with QueueLogging[Int]
     with QueueFiltering[Int] {
  def veto(t: Int) = t < 0
}
```

Defines “veto”

```
for (i <- -2 to 2) {
    sq.put(i)
}
```

loop from -2 to 2

```
// => -2 rejected!
// => -1 rejected!
// => put: 0
// => put: 1
// => put: 2
```

Filtering occurred before logging

Example use

What if we

reverse the *order*

of the Traits?

```
val sq = new StandardQueue[Int]
     with QueueFiltering[Int]
     with QueueLogging[Int]   {
  def veto(t: Int) = t < 0
}
```

Order switched

```
for (i <- -2 to 2) {
    sq.put(i)
}
```

```
// => put: -2
// => -2 rejected!
// => put: -1
// => -1 rejected!
// => put: 0
// => put: 1
// => put: 2
```

logging comes before filtering!

Loosely speaking, the *precedence* goes *right* to *left*.

"Linearization" algorithm

Case Classes

Recall:

```
class Complex(val real: Double,
              val imag: Double)
{…}

object Complex{
 def apply(r:Double,i:Double) =
    new Complex(r, i)
}
```

This pattern is so common...

Equivalent:

```
case class Complex(
        real: Double, imag: Double)
 {…}
```

You also get an *unapply* method...

… and why is the keyword called case?

Pattern Matching:

```
val c = Complex(…)
c match {
 case Complex(0.0, 0.0) =>
  println(“zero!”)
 case Complex(r, 0.0) =>
  println(“real: ”+r)
 case Complex(r, i) =>
  println(“(”+r+“,”+i+“)”)
}
```

Invokes unapply

For Loops: *Sequence Comprehensions*

```
object CapsStartFor {

 def main(args: Array[String]) = {
  for {
   i <- 0 until args.length
   arg = args(i)
   if (arg(0).isUpperCase)
  }
    println(arg)
}}
// $ scalac CapsStartFor.scala
// $ scala -cp . CapsStartFor aB Ab AB ab
// Ab
// AB
```

“For” can have an arbitrary number of generators, conditions, assignments

Concurrency through *Actors*

Scala's *Actor Model*

- Patterned after *Erlang's*.
- Allows *shared*, *mutable state*.
    - But *discouraged*.

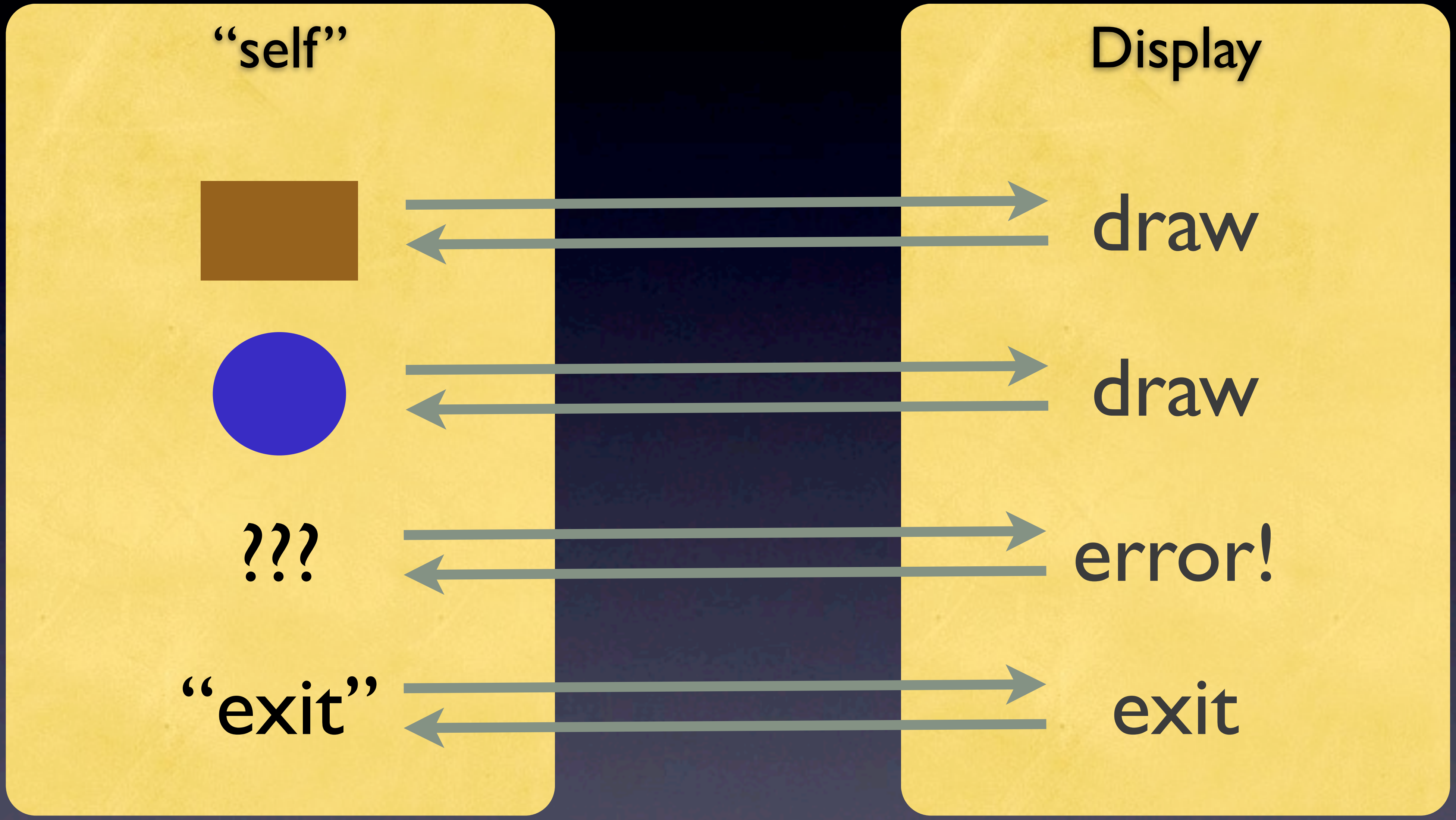
Akka’s *Actors*

akkasource.org

Akka

- Inspired by *Erlang OTP*.
- Clean-room *Actor* lib.
  - Better *performance*.
  - Adds *supervisors, lifecycle management*.

2 Actors:



First, the *Supervisor*.

```
object factory
   extends SupervisorFactory {
 override def getSupervisorConfig =
{…}

val supervisor =
    factory.newSupervisor
supervisor.startSupervisor
```

next slide

```
{
 SupervisorConfig(
  RestartStrategy(OneForOne, 3, 100),
  Supervise(
   new ShapeDrawingActor,
   LifeCycle(Permanent, 100)) :: Nil)
}
```

Next, some *Support Types*.

```
package shapes

case class Point(
 x: Double, y: Double)

abstract class Shape {
 def draw()
}
```

abstract “draw” method

Hierarchy of geometric shapes

```
case class Circle(
 center:Point, radius:Double)
    extends Shape {
 def draw() = …
}

case class Rectangle(
 ll:Point, h:Double, w:Double)
    extends Shape {
 def draw() = …
}
```

concrete "draw" methods

Hierarchy of geometric shapes

Finally, the *Actors*.

```
package shapes
import …akka…actors._, Actor._
object ShapeDrawingActor
        extends Actor {
 def init = {…} // startup
 def receive = {
   // pattern matcher to
   // handle each message
 }
}
```

Actor for drawing shapes

```
receive = {
 case s:Shape =>
   s.draw()
   sender ! "drawn"
 case "exit" =>
   println("exiting...")
   sender ! "bye!"
   // exit
 case msg =>
   println("Error: " + msg)
   sender ! "Unknown: " + msg
}
```

```
import shapes._
import …akka…actors.Actor._,
       …Self

def sendAndReceive(msg: Any) ={
 (ShapeDrawingActor !! msg)
   match {

  case reply => println(reply)
 }
}
```

send and await reply

script to try it out

```
…
sendAndReceive(
  Circle(Point(0.0,0.0), 1.0))
sendAndReceive(
  Rectangle(Point(0.0,0.0), 2, 5))
sendAndReceive(3.14159)
sendAndReceive("exit")

// => Circle(Point(0.0,0.0),1.0)
// => drawn.
// => Rectangle(Point(0.0,0.0),2.0,5.0)
// => drawn.
// => Error: 3.14159
// => Unknown message: 3.14159
// => exiting...
// => bye!
```

```
…
sendAndReceive(
  Circle(Point(0.0,0.0), 1.0))
sendAndReceive(
  Rectangle(Point(0.0,0.0), 2, 5))
sendAndReceive(3.14159)
sendAndReceive("exit")

// => Circle(Point(0.0,0.0),1.0)
// => drawn.
// => Rectangle(Point(0.0,0.0),2.0,5.0)
// => drawn.
// => Error: 3.14159
// => Unknown message: 3.14159
// => exiting...
// => bye!
```

```
…
receive = {
 case s:Shape =>
  s.draw()
  sender ! "drawn"

 case …
 case …
}
```

pattern matching

polymorphism

A powerful combination!

Recap

Scala is...

a *better*
Java and C#,

object-oriented

and

functional,

succinct,

elegant,

yet

powerful.

Thanks!

dean@deanwampler.com

@deanwampler

programmingscala.com
polyglotprogramming.com/talks