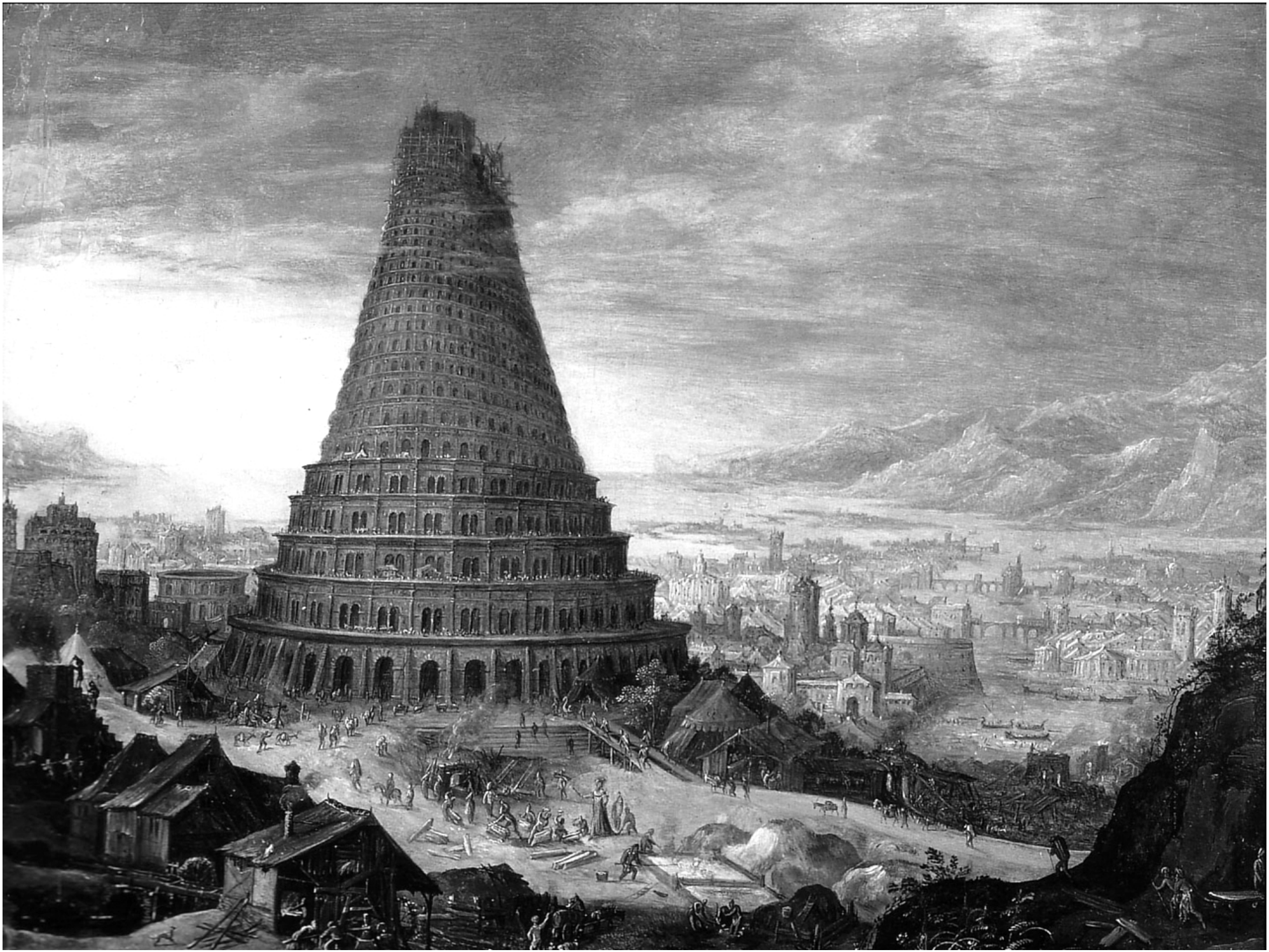
Introduction to Scala

Aleksandar Prokopec

EPFL



Scala

Pragmatic

runs on the JVM

Martin Odersky

Seamless Java interoperability

Since 2003

Production ready

Statically typed

Hybrid

Statically typed

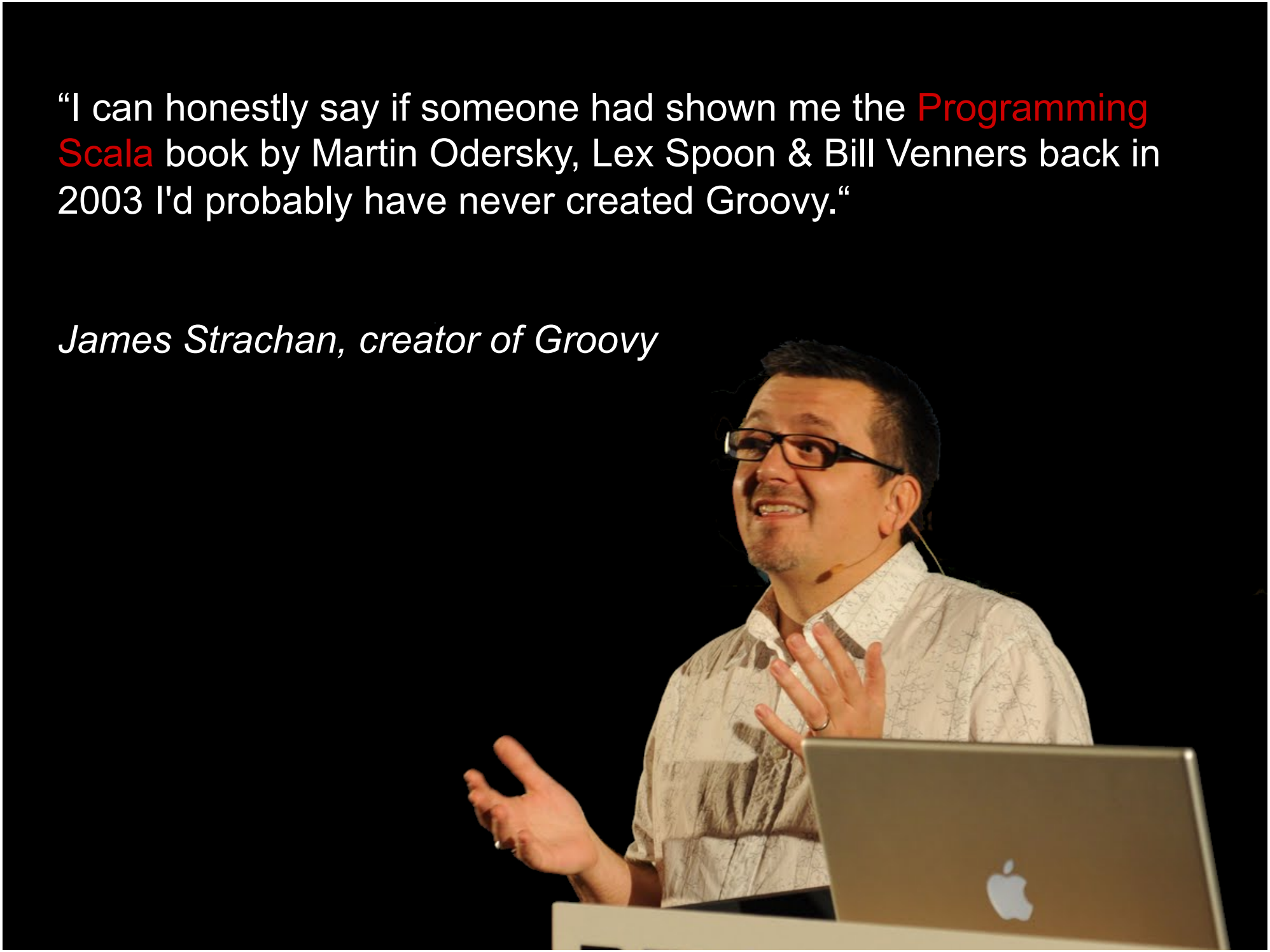
runs on the JVM

Scala programs are fast.


OOP + FP

“I can honestly say if someone had shown me the Programming Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy.“

James Strachan, creator of Groovy



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

James Gosling

Pragmatic

Scala is lightweight.

```
println(“Hello world!”)
```

REPL

evaluating expressions on the fly

```
scala> println(“Hello world!”)
Hello world!
```

Compiled version

```
object MyApp extends App {
  println(“Hello world!”)
}
```

Singleton objects

no more static methods

```
object MyApp extends App {
  println(“Hello world!”)
}
```

Declaring methods

```
object MyApp {
  def main(args: Array[String]) {
    println("Hello world!")
  }
}
```

Declaring variables

```
object MyApp {
  def main(args: Array[String]) {
    var user: String = args(0)
    println("Hello, "+user+"!")
  }
}
```

Declaring values

prevents accidental changes

```
object MyApp {
  def main(args: Array[String]) {
    val user: String = args(0)
    println("Hello, "+user+"!")
  }
}
```

Local type inference

less… “typing”

```
object MyApp {
  def main(args: Array[String]) {
    val user = args(0)
    println(“Hello, ”+user+“!”)
  }
}
```

Local type inference

less… “typing”

```
class StringArrayFactory {
  def create: Array[String] = {
    val array: Array[String] =
      Array[String](“1”, “2”, “3”)
    array
  }
}
```

Local type inference

less… “typing”

```
class StringArrayFactory {
  def create = {
    val array = Array(“1”, “2”, “3”)
    array
  }
}
```

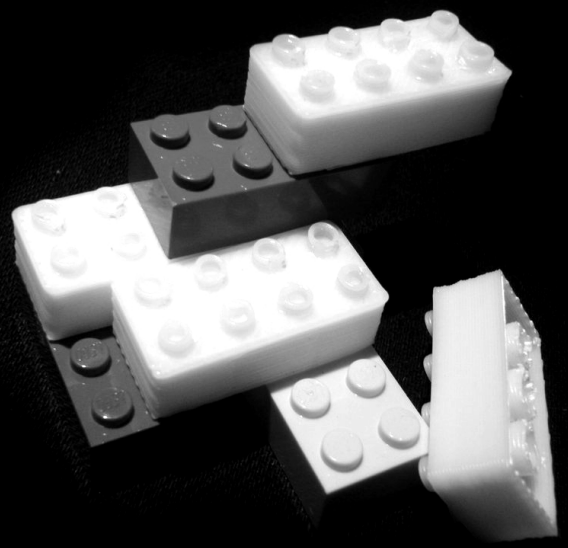
Declaring classes

…concisely

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

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

Scala is object-oriented.



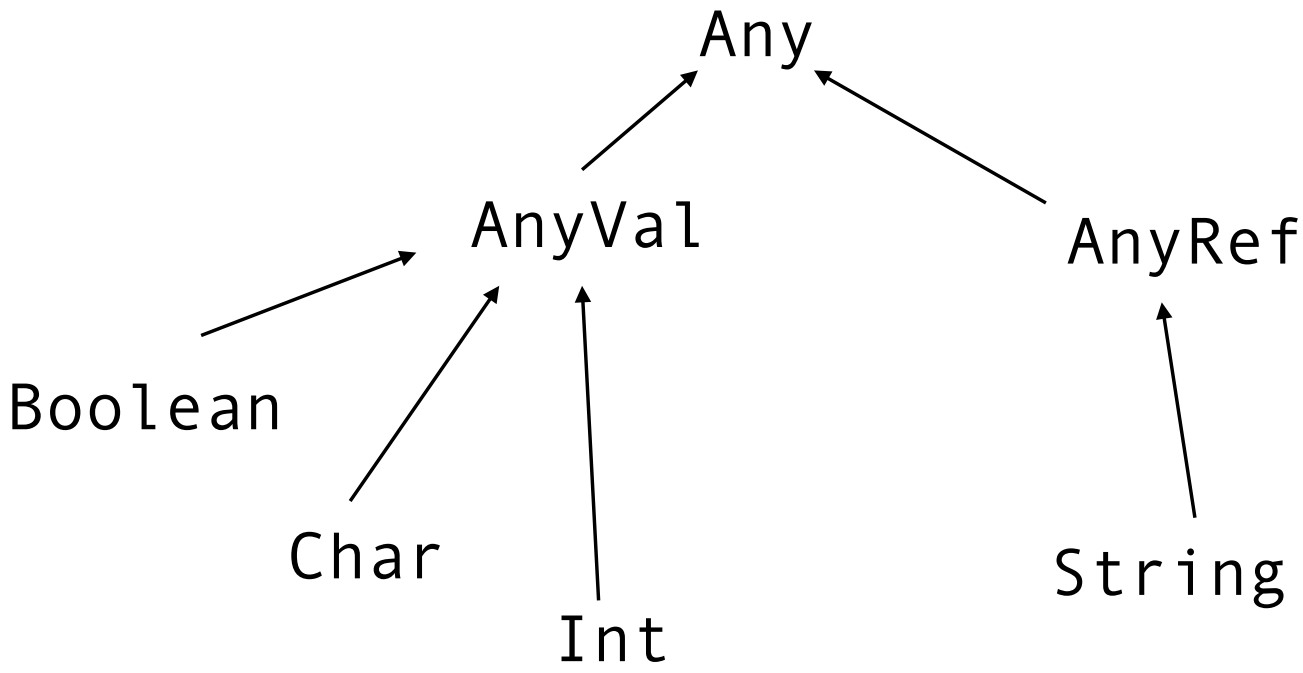
Object-oriented

everything's an object

```
object Foo {
  val b = new ArrayBuffer[Any]
}
```

Object-oriented

everything's an object



Object-oriented

everything's an object

```
object Foo {
  val b = new ArrayBuffer[Any]
  b += 1
  b += 1.toString
  b += Foo
  println(b)
}
```

Operator overloading

operators are methods

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

```
Array(1, 2, 3) ++ Array(4, 5, 6)
Array(1, 2, 3).++(Array(4, 5, 6))
```

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

Declaring traits

traits are like interfaces

```
trait Iterator[T] {
  def next(): T
  def hasNext: Boolean
}
```

Declaring traits

traits are rich

```
trait Iterator[T] {
  def next(): T
  def hasNext: Boolean
  def printAll() =
    while (hasNext) println(next())
}
```

Multiple inheritance

traits are composable

```
trait Animal
```

Multiple inheritance

traits are composable

```
trait Animal
trait Mammal extends Animal {
  def think() = println(“hm...”)
}
```

Multiple inheritance

traits are composable

```
trait Animal
trait Mammal extends Animal {
  def think() = println(“hm...”)
}
trait Bird extends Animal {
  def layEgg() = System.createEgg()
}
```

Mixin composition

traits are composable

```
trait Animal
trait Mammal extends Animal {
  def think() = println("hm...")
}
trait Bird extends Animal {
  def layEgg() = System.createEgg()
}
class Platypus extends Bird with Mammal
```

Dynamic mixin composition

…or composition “on the fly”

```
trait Animal
trait Reptile extends Animal {
  def layInTheSun: Unit = {}
}
class Dog(name: String) extends Mammal

new Dog(“Nera”) with Reptile
```

Cake pattern

Self-types

to express requirements

```
trait Logging {
  def log(msg: String)
}

trait AnsweringMachine {
  self: Logging with DAO with Protocol =>
  log("Initializing.")
  ...
}
```

Cake pattern

layers above layers

```
trait ConsoleLogging {
  def log(msg: String) = println(msg)
}

class LocalAnsweringMachine
extends AnsweringMachine
    with ConsoleLogging
    with H2DAO
    with JabberProtocol
```

Scala is functional.

First class functions

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

First class functions

functions are objects too

```
val doub: Int => Int = (x: Int) => x * 2

doub(1)
→ 2
```

First class functions

as higher order parameters

```
val doub = (x: Int) => x * 2

List(1, 2, 3).map(doub)
→ List(2, 4, 6)
```

Functions with sugar

make code sweeter

```
List[Int](1, 2, 3).map((x: Int) => x *
2)

// more type inference
List(1, 2, 3).map(x => x * 2)

// or even shorter
List(1, 2, 3).map(_ * 2)
```

Closures

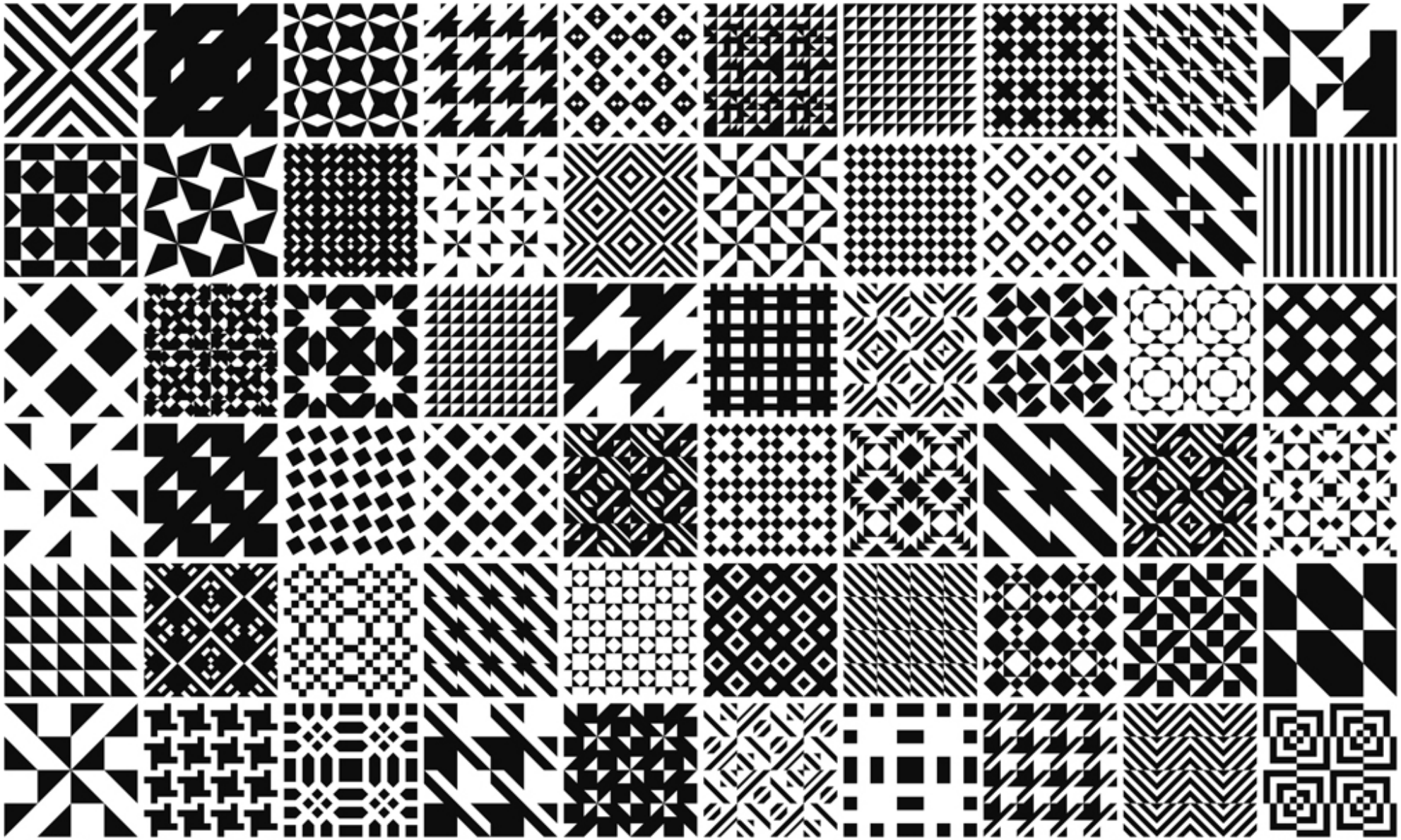
functions that “capture” their environment

```
var step = 1
val inc = x => x + step

inc(5)
→ 6
step = 2
inc(5)
→ 7
```

First class functions

because you write them in Java all the time

```
// Java
button.addMouseListener(new MouseAdapter() {
  public void mouseEntered(MouseEvent e) {
    System.out.println(e);
  }
}

// Scala
listenTo(button)
reactions += { case e => println(e) }
```



Pattern matching

Pattern matching

…is concise

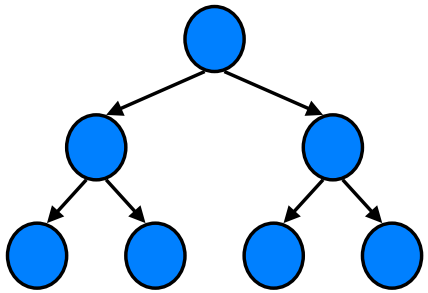
```
// Scala
reactions += {
  case m: MouseEntered =>
    println(“I see it!”)
  case m: MouseExited =>
    println(“Lost it.”)
  case m: MouseClicked =>
    println(“Poked!”)
}
```

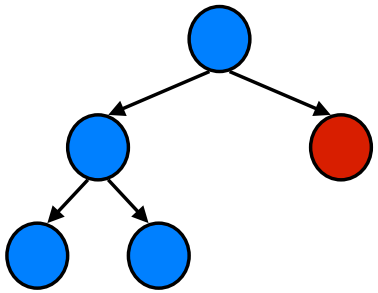
```
// Java
button.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {
    System.out.println(“I see it!”);
  }
  public void mouseExited(MouseEvent e) {
    System.out.println(“Lost it.”);
  }
  public void mouseClicked(MouseEvent e) {
    System.out.println(“Poked!”);
  }
}

// ...alternative - isinstanceof
```

Pattern matching

…is precise

```
trait Tree
case class Node(l: Tree, r: Tree)
  extends Tree
case object Leaf
  extends Tree
```





Pattern matching

…is precise

```
def b(t: Tree): Int = t match {
  case Node(Leaf, Node(_, _)) |
       Node(Node(_, _), Leaf) => -1
  case Node(l, r) =>
    val (ld, rd) = (b(l), b(r))
    if (ld == rd) ld + 1 else -1
  case Leaf => 0
  case _ => error("Unknown tree!")
}
```

Pattern matching

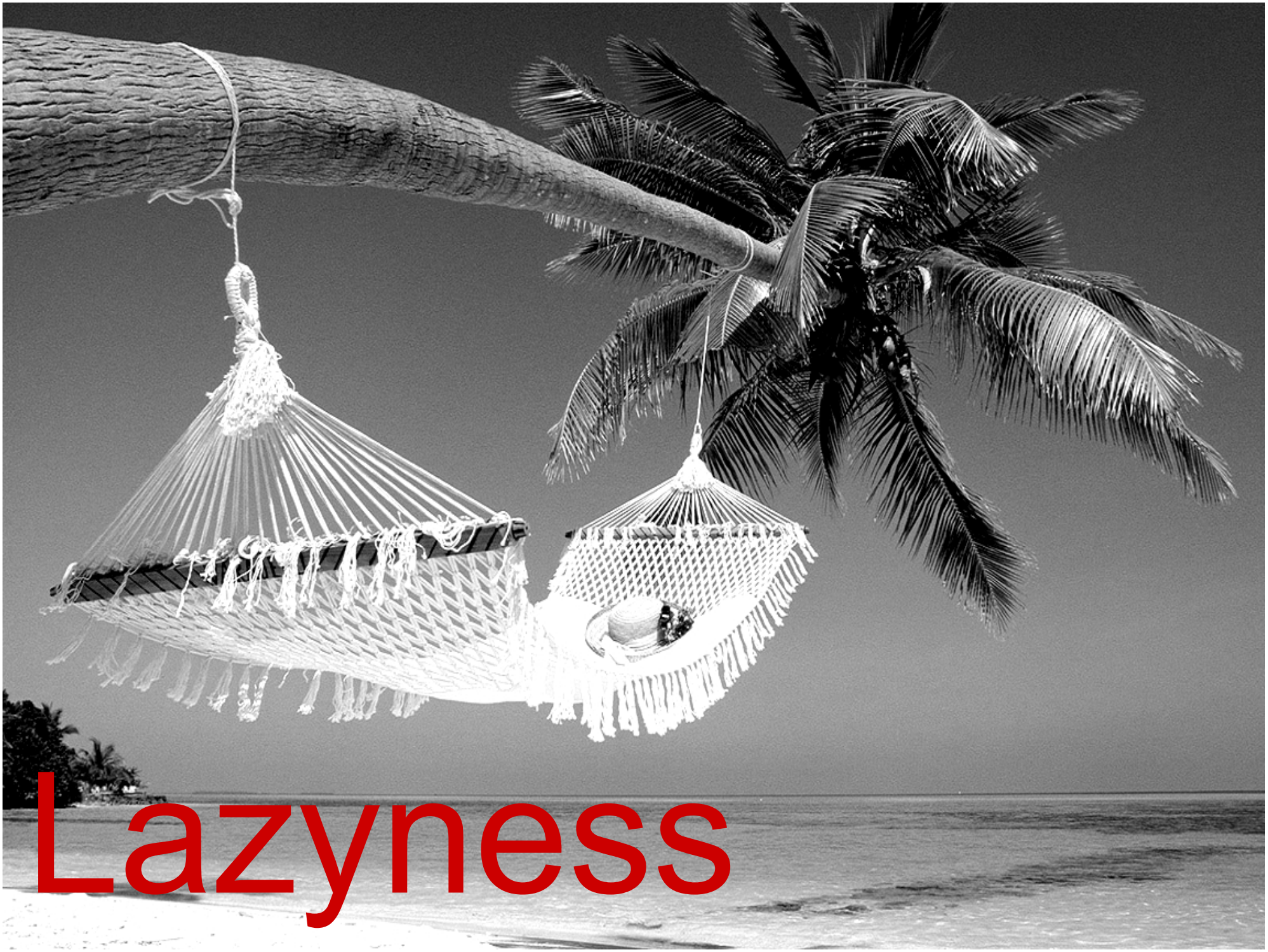
…is exhaustive

```
sealed trait Tree
...
def b(t: Tree): Int = t match {
  case Node(Leaf, Node(_, _)) |
       Node(Node(_, _), Leaf) => -1
  case Node(l, r) =>
    val (ld, rd) = (b(l), b(r))
    if (ld == rd) ld + 1 else -1
  case Leaf => 0
}
```

Pattern matching

…is extensible

```
def matchingMeSoftly(a: Any): Any =
  a match {
    case 11 => “eleven”
    case s: String => “’%s’”.format(s)
    case <tag>{t}</tag> => t
    case Array(1, 2, 3) => “1, 2, 3”
    case head :: tail => tail
    case _ => null
  }
```


Lazyness

lazy values

don’t compute if there’s no demand

```
class User(id: Int) {
  lazy val followernum =
    from(followers)(f =>
      where(id === f.fid)
      compute(countDistinct(f.fid))
    )
}
```

Call by name

evaluate only when you have to

```
def withErrorOut(body: =>Unit) = {
  val old = Console.out
  Console.setOut(Console.err)
  try body
  finally Console.setOut(old)
}
...
withErrorOut {
  if (n < 0) println("n too small")
}
```

Streams

lazy lists

$$e = \sum 1/n!$$

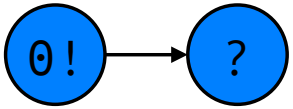
$$= 1/0! + 1/1! + 1/2! + 1/3! + 1/4! + \ldots$$

Streams

lazy lists

$$e = \sum 1/n!$$

$$= 1/0! + 1/1! + 1/2! + 1/3! + 1/4! + \ldots$$

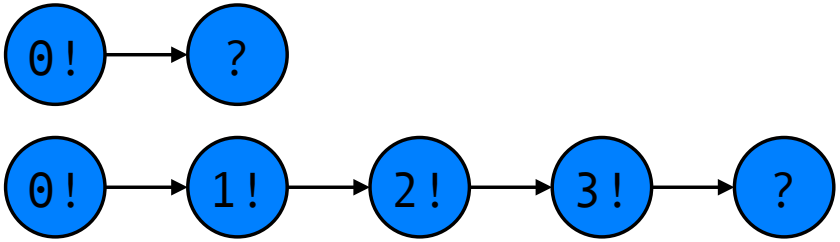


Streams

lazy lists

$$e = \sum 1/n!$$

$$= 1/0! + 1/1! + 1/2! + 1/3! + 1/4! + \ldots$$

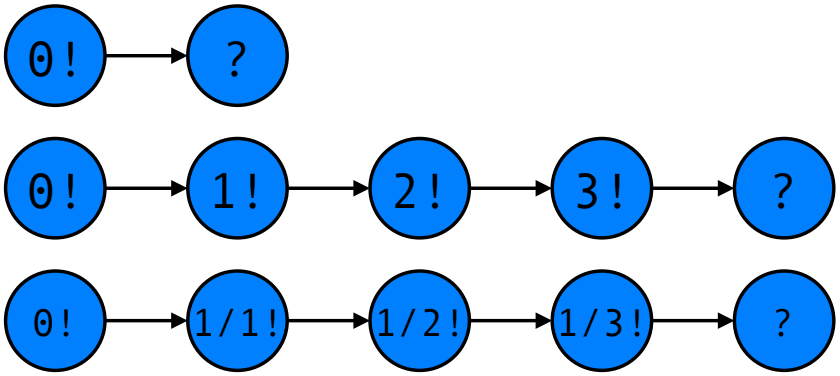


Streams

lazy lists

$$e = \sum 1/n!$$

$$= 1/0! + 1/1! + 1/2! + 1/3! + 1/4! + \ldots$$



Streams

lazy lists

```
def fact(n: Int, p: Int): Stream[Int] =
  p #:: fact(n + 1, p * (n + 1))

val factorials = fact(0, 1)

val e = factorials.map(1./
_).take(10).sum
```

Scala is expressive.

for comprehensions

traverse anything

```
for (x <- List(1, 2, 3)) println(x)
List(1, 2, 3).foreach(x => println(x))

for (x <- 0 until 10) println(x)
(0 until 10).foreach(x => println(x))
Range(0, 10, 1).foreach(x => println(x))
```

for comprehensions

map anything

```
for (x <- List(1, 2, 3)) yield x * 2
List(1, 2, 3).map(x => x * 2)

for (x <- List(1, 2); y <- List(1, 2))
  yield x * y
List(1, 2).flatMap(x =>
  List(1, 2).map(y => x * y)
)
→ List(1, 2, 2, 4)
```

for comprehensions

like SQL queries

```
for {
  p <- people
  if p.age > 25
  s <- schools
  if p.degree == s.degree
} yield (p, s)

// pairs of people older than 25 and
// schools they possibly attended
```

Collections

easy to create

```
val phonebook = Map(
  “Jean” -> “123456”,
  “Damien” -> “666666”)

val meetings = ArrayBuffer(
  “Dante”, “Damien”, “Sophie”)

println(phonebook(meetings(1)))
```

Collections

high-level combinators

```
// Java
boolean isOk = true
for (int i = 0; i < name.length(); i++)
{
  if (isLetterOrDigit(name.charAt(i)) {
    isOk = false;
    break;
  }
}
```

Collections

high-level combinators

```
// Scala
name.forall(_.isLetterOrDigit)
```

Collections

high-level combinators

```
// count the total number of different
// surnames shared by at least 2 adults
people
```

Collections

high-level combinators

```
// count the total number of different
// surnames shared by at least 2 adults
people.filter(_.age >= 18)
```

Collections

high-level combinators

```
// count the total number of different
// surnames shared by at least 2 adults
people.filter(_.age >= 18)
  .groupBy(_.surname): Map[String, List[Person]]
```

Collections

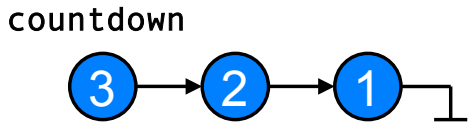
high-level combinators

```
// count the total number of different
// surnames shared by at least 2 adults
people.filter(_.age >= 18)
  .groupBy(_.surname): Map[String, List[Person]]
  .count { case (s, l) => l.size >= 2 }
```

Lists

an immutable sequence

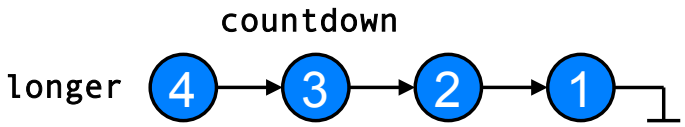
```
val countdown = List(3, 2, 1)
```



Lists

an immutable sequence

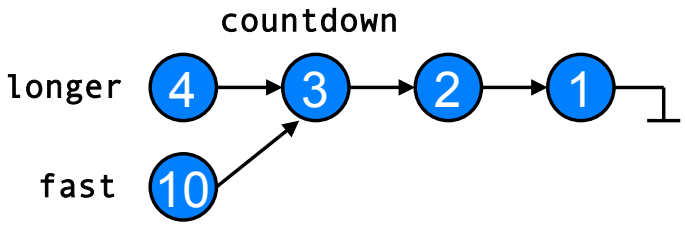
```
val countdown = List(3, 2, 1)
val longer = 4 :: countdown
```



Lists

an immutable sequence

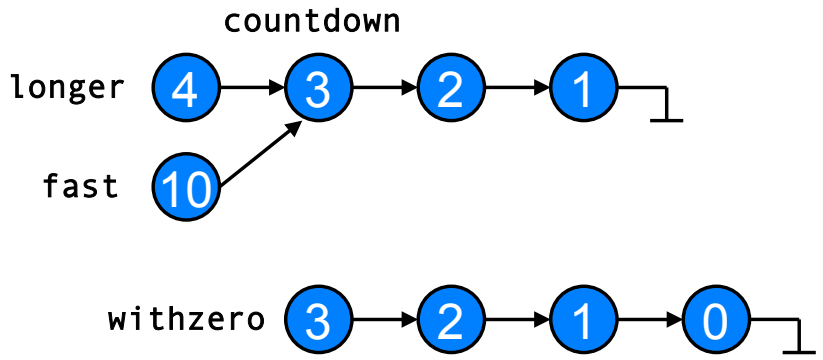
```
val countdown = List(3, 2, 1)
val longer = 4 :: countdown
val fast = 10 :: countdown
```



Lists

an immutable sequence

```
val countdown = List(3, 2, 1)
val longer = 4 :: countdown
val fast = 10 :: countdown
val withzero = countdown ::: List(0)
```



Buffers

mutable sequences

```
val b = ArrayBuffer(1, 2, 3)
b += 4
b += 5
b += 6
→ ArrayBuffer(1, 2, 3, 4, 5, 6)
```

Maps

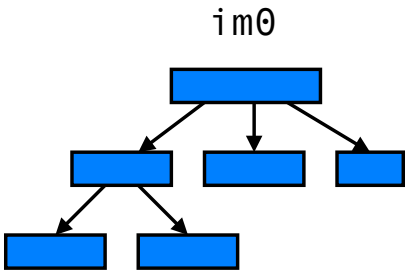
mutable or immutable, sorted or unsorted

```
import collection._

val m = mutable.Map(“Heidfeld” -> 1,
  “Schumacher” -> 2)
m += “Hakkinen” -> 3

val im = immutable.Map(“Schumacher” ->
1)
```

Hash tries

persistence through efficient structural sharing

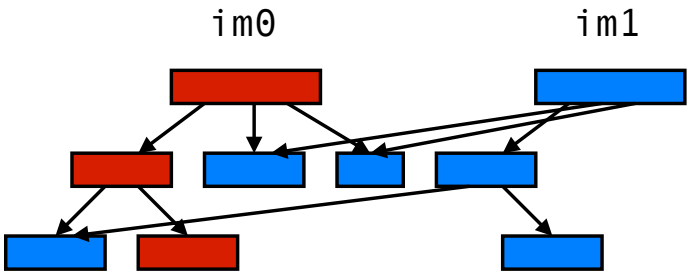
```
val im0: Map[Int, Int] = ...
```



Hash tries

persistence through efficient structural sharing

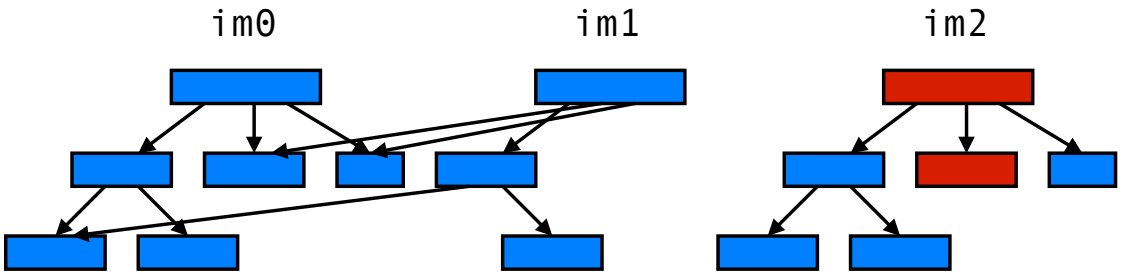
```
val im0: Map[Int, Int] = ...
val im1 = im0 + (1 -> 1)
```



Hash tries

persistence through efficient structural sharing

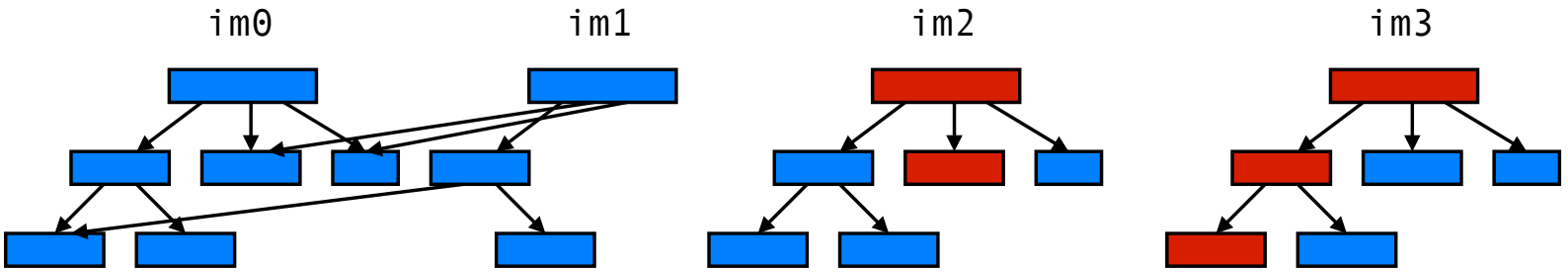
```
val im0: Map[Int, Int] = ...
val im1 = im0 + (1 -> 1)
val im2 = im1 + (2 -> 2)
```



Hash tries

persistence through efficient structural sharing

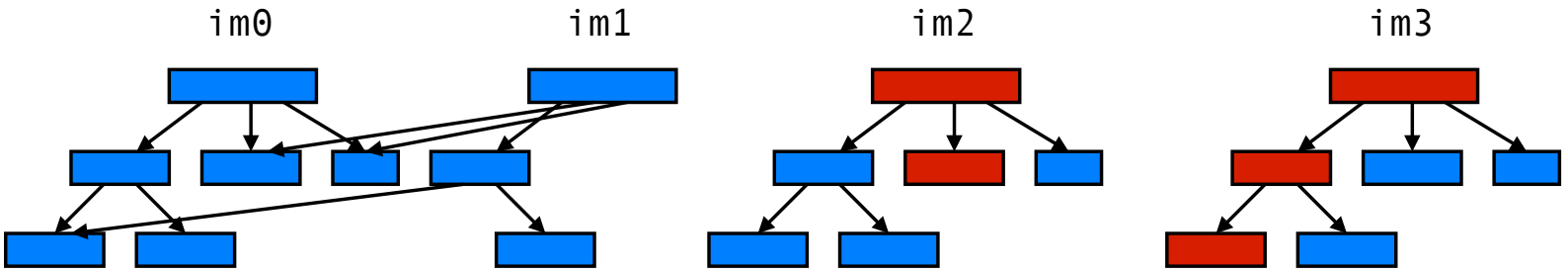
```
val im0: Map[Int, Int] = ...
val im1 = im0 + (1 -> 1)
val im2 = im1 + (2 -> 2)
val im3 = im2 + (3 -> 6)
```



Hash tries

persistence through efficient structural sharing

2x-3x slower lookup
2x faster iteration



Parallel collections

parallelize bulk operations on collections

```
def cntEqlen(m: Map[String, String]) = {
  m.par.count {
    case (n, s) => n.length == s.length
  }
}

// be careful with side-effects
```

Scala is extensible.

One ring to rule them all.

actors

road to safer concurrency

```
val a = actor {
  react {
    case i: Int => println(i)
  }
}
...
a ! 5
```

Custom control flow

it’s all about control

```
def myWhile(c: =>Boolean)(b: =>Unit) {
  if (c) {
    b
    myWhile(c)(b)
  }
}
```

Custom control flow

it’s all about control

```
@tailrec
def myWhile(c: =>Boolean)(b: =>Unit) {
  if (c) {
    b
    myWhile(c)(b)
  }
}
```

Custom control flow

it's all about control

```
@tailrec
def myWhile(c: =>Boolean)(b: =>Unit) {
  if (c) {
    b
    myWhile(c)(b)
  }
}
var i = 0
myWhile (i < 5) {
  i += 1
}
```

ARM

automatic resource management

```
withFile (“~/.bashrc”) { f =>
  for (l <- f.lines) {
    if (“#”.r.findFirstIn(l) != None)
      println(l)
  }
}
```

ScalaTest

behavioral testing framework

```
"A list" should {

  "be a double reverse of itself" in {
    val ls = List(1, 2, 3, 4, 5, 6)
    ls.reverse.reverse should equal (ls)
  }

}
```

BaySick

Basic DSL in Scala

```
10 PRINT “Baysick Lunar Lander v0.9”
20 LET ('dist := 100)
30 LET ('v := 1)
40 LET ('fuel := 1000)
50 LET ('mass := 1000)
...
```

implicit conversions

augmenting types with new operations

```
'a'.toUpperCase
```

implicit conversions

augmenting types with new operations

```
implicit def charOps(c: Char) = new {
  def toUpperCase =
    if (c >= ‘a’ && c <= ‘z’)
      (c - 32).toChar
    else c
}

...
‘a’.toUpperCase
```

Pimp my library

implicit conversions

pimping your libraries since 2006

```
import scalaj.collection._

val list = new java.util.ArrayList[Int]
list.add(1)
list.add(2)
list.add(3)
...
for (x <- list) yield x * 2
```

implicit conversions

pimping your libraries since 2006

```
import scalaj.collection._

val list = new java.util.ArrayList[Int]
list.add(1)
list.add(2)
list.add(3)
...
for (x <- list) yield x * 2
// list.map(x => x * 2)
```

implicit conversions

pimping your libraries since 2006

```
import scalaj.collection._

val list = new java.util.ArrayList[Int]
list.add(1)
list.add(2)
list.add(3)
...
for (x <- list) yield x * 2
// jlist2slist(list).map(x => x * 2)
```

implicit arguments

restricting operations on types

```
val is = SortedSet(1, 2, 3)

case class Man(id: Int)
...
implicit object MenOrd extends Ordering[Man] {
  def compare(x: Man, y: Man) = x.id - y.id
}

val ms = SortedSet(Person(1), Person(2))
```

STM

software transactional memory

```
val account1 = cell[Int]
val account2 = cell[Int]

atomic { implicit txn =>
  if (account2() >= 50) {
    account1 += 50
    account2 -= 50
  }
}
```

Scala has support.

Building

Ant
SBT
Maven

SBT

simple build tool

```
> compile
...
> ~compile
...
> test
...
> package
```

SBT

project definition written in Scala

```
val scalatest =
  “org.scala-tools.testing” %
  “scalatest” % “0.9.5”

...

> reload
[INFO] Building project ...
[INFO]    using sbt.Default ...
> update
```

Tools: IDEs

Eclipse
Netbeans
IntelliJ IDEA
Emacs
ENSIME
JEdit

Web frameworks

Used by...


twitter


foursquare


LinkedIn


EDF
Electricité
de France


Novell


GridGain
OPEN CLOUD PLATFORM


SONY


SIEMENS

Thank you!