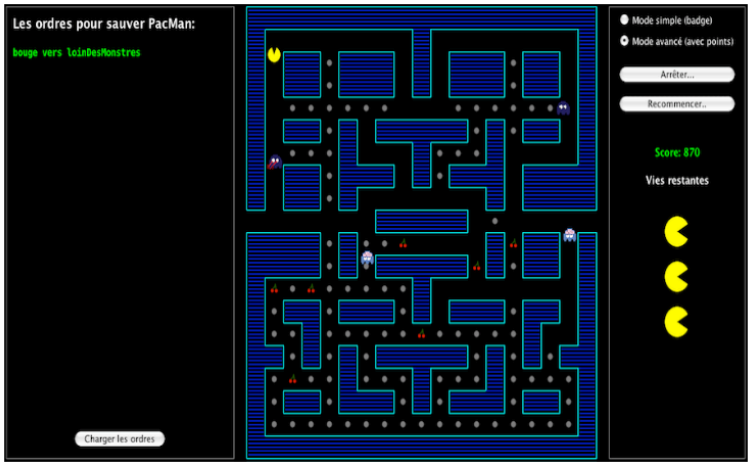
面向对象分析与设计

齐琦
海南大学

编程设计项目

- 小组组成情况
  - 编程设计内容

- 编程设计参考项目
  - Pacman游戏 （https://github.com/ScalaPino/pacman）



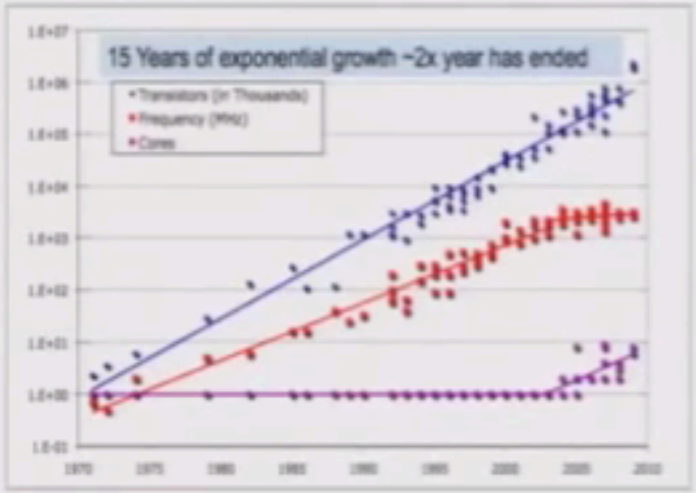
今天的内容

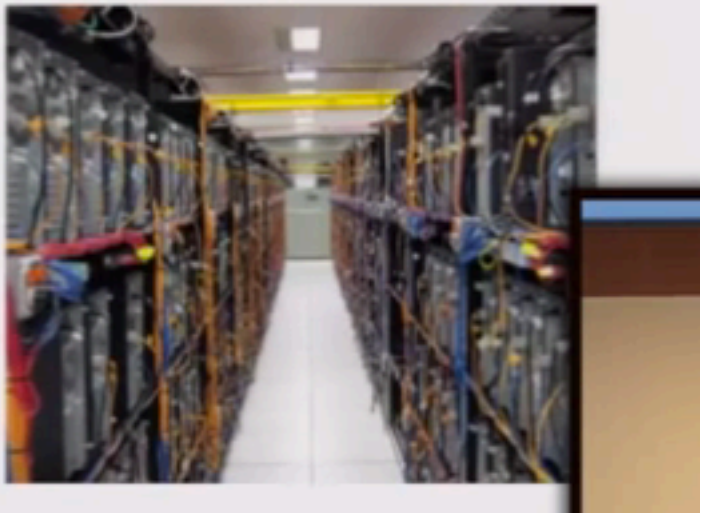
- 继续Scala的基本介绍
- 纵览一下Scala的（简单）基本构件
- 演示几个Scala的程序

软件（编程模式）变革的催化器

A Catalyzer

- Two forces driving software complexity:
  - Multicore (= parallel programming)
  - Cloud computing (= distributed programming)
- Current languages and frameworks have trouble keeping up (locks/threads don't scale)
- Need better tools with the right level of abstraction





主要的羁绊是可被改变的状态变量

Parallel – how to make use of multicore, GPUs, clusters?

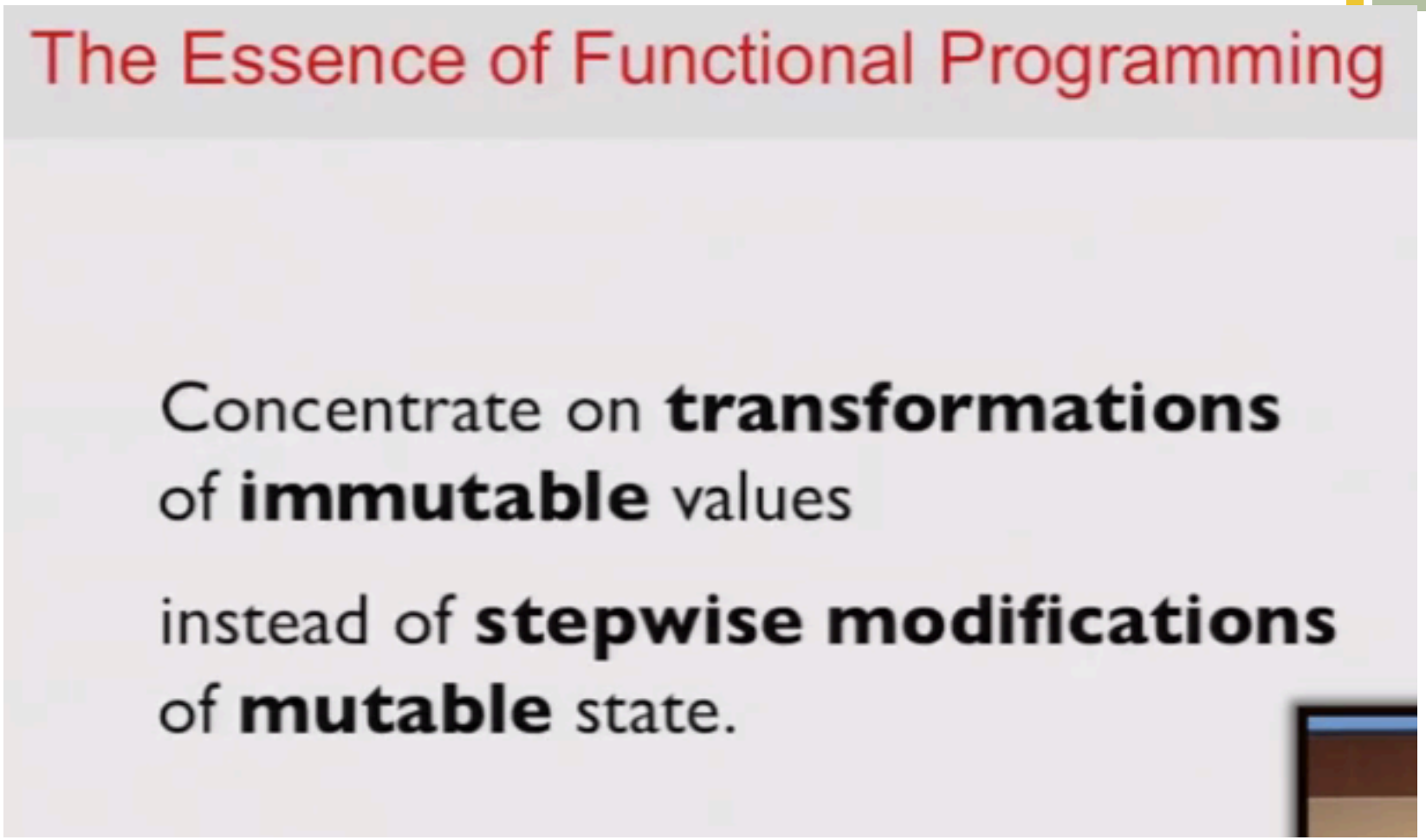
Async – how to deal with asynchronous events

Distributed – how to deal with delays and failures?

Mutable state is a liability for each one of these!

- Cache coherence
- Races
- Versioning

函数编程的本质：对不变值的操作和转移



对象概念仍然需要

What about Objects?

So, should we forget OO and all program in functional programming languages?

No! What the industry learned about OO decomposition in analysis and design stays valid.

Central question: What goes where?

In the end, need to put things somewhere, or risk unmanageable global namespace.

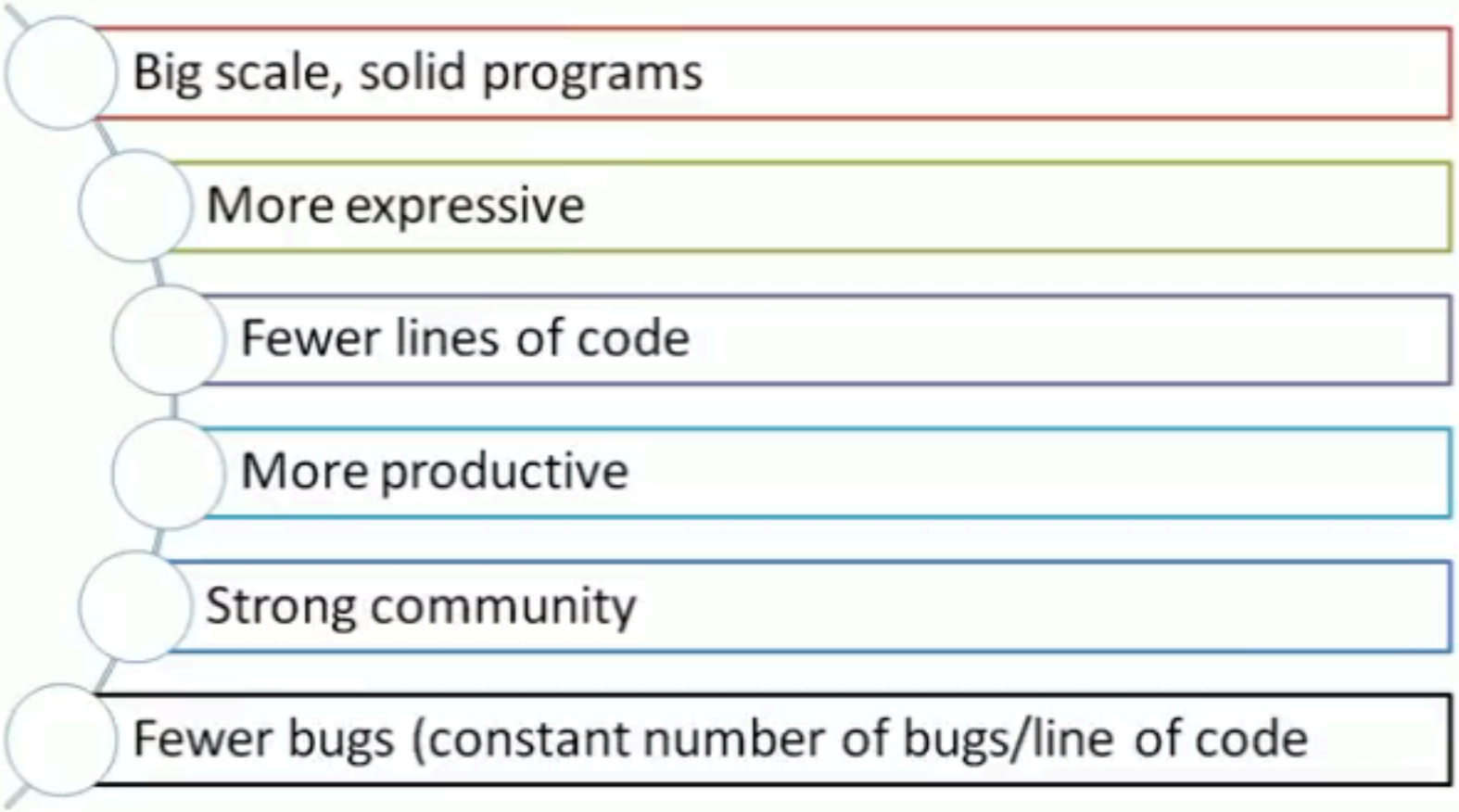
工业界应用广泛


Scala


Genentech
IN BUSINESS FOR LIFE
LinkedIn
Walmart
box
JUNIPER NETWORKS
NCL
NORWEGIAN CRUISE LINE
workday.
DIRECTV
GILT GROUPE
Angie's list.
white pages
here
Scotiabank
Saks Fifth Avenue
hp
THE HUFFINGTON POST
TOMAX retail.net


coursera
simpleBI
foursquare
primetalk
theguardian
flowdock
abiquo

Why In the Enterprise?



现代软件开发者所选择的10大技术之一

SOURCE: Typesafe

July 23, 2014 17:54 ET

Scala Named a "Top 10 Technology for Modern Developers"

RebelLabs Study Based on Developer Feedback and Market Data Affirms Popularity of Scala as Powerful Language for the JVM

SAN FRANCISCO, CA--(Marketwired - July 23, 2014) - Typesafe, provider of the world's leading Reactive platform and the company behind Play Framework, Akka, and Scala, today announced that Scala has been selected as one of the "10 Kick-Ass Technologies Modern Developers Love" in an independent study conducted by RebelLabs, the research and content think tank of ZeroTurnaround.

The study set out to reveal the technologies most "loved and chosen specifically by geeks" based on selection criteria of "Market Data" (raw % of market, relative size of market, level of fragmentation), "Developer Feedback" (i.e. explicit responses from surveys), "Amount of Buzz" (press coverage, social media share, community presence), and "Anecdotal Evidence" (personal conversations, stories, gut feeling). In being recognized, Scala joined other hot developer technologies that got a nod, including (in alphabetical order): Confluence, Git, Gradle, Groovy, IntelliJ IDEA, Jenkins, JIRA, MongoDB and Tomcat + TomEE.

"Scala has evolved a lot since its inception in 2004," said Martin Odersky, creator of Scala. "Its user base has grown to a size I could have never imagined back then, and has been picked up in industries ranging from major enterprises and financial institutions to startups. Being recognized by RebelLabs is a huge honor and validates the work we're doing to create an elegant and powerful programming language."

According to the study, when developers were asked "which alternative JVM (Java Virtual Machine) language they would be most interested in learning," 47% listed Scala as their next choice. The study cites high interest in Scala's sbt, and Typesafe's "focus and the community backing of the Scala ecosystem of tools, including Akka and Play" among key contributing factors for the rise of Scala's popularity as a language.

Additional Resources:

- Full Report: 10 Kick-Ass Technologies Modern Developers Love
- Typesafe Reactive Platform / Scala

Scala的风格

Bridging the Gap

- Scala acts as a bridge between the two paradigms.
- To do this, it tries to be:
  - orthogonal
  - expressive
  - un-opinionated
- It naturally adapts to many different programming styles.

A Question of Style

Because Scala admits such a broad range of styles, the question is which one to pick?

Certainly, Scala is:

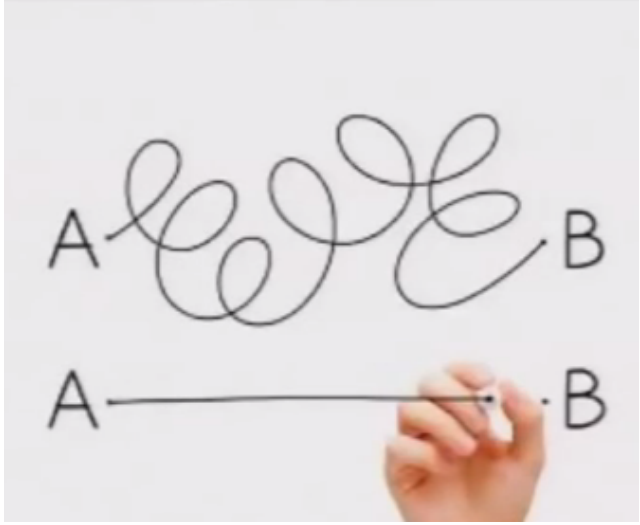
- Not a better Java
- Not a Haskell on the JVM

But what is it then?

I expect that a new fusion of functional and object-oriented will emerge.

Scala 风格

#1 Keep it Simple



#2 Don't pack too much in one expression

- I sometimes see stuff like this:

```
jp.getRawClasspath.filter(
  _.getEntryKind == IClasspathEntry.CPE_SOURCE).
  iterator.flatMap(entry =>
    flatten(ResourcesPlugin.getWorkspace.
      getRoot.findMember(entry.getPath)))
```

- It's amazing what you can get done in a single statement.
- But that does not mean you have to do it.

#3 Prefer Functional

Prefer Functional ...

- By default:
  - use vals, not vars.
  - use recursions or combinators, not loops.
  - use immutable collections
  - concentrate on transformations, not CRUD.
- When to deviate from the default:
  - Sometimes, mutable gives better performance.
  - Sometimes (but not that often!) it adds convenience.

#4 ... But don't diabolize local state

Why does mutable state lead to complexity?

It interacts with different program parts in ways that are hard to track.

=> Local state is less harmful than global state.

More local state examples

The canonical functional solution uses a `foldLeft`:

```
val (totalPrice, totalDiscount) =
  items.foldLeft((0.0, 0.0)) {
  case ((tprice, tdiscount), item) =>
    (tprice + item.price,
     tdiscount + item.discount)
  }
}
```

Whereas the canonical imperative solution looks like this:

```
var totalPrice, totalDiscount = 0.0
for (item <- items) {
  totalPrice += item.price
  totalDiscount += item.discount
}
```

#5 Careful with mutable objects

Mutable objects tend to encapsulate global state.

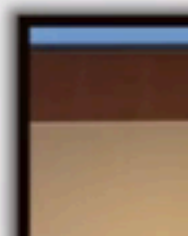
"Encapsulate" sounds good, but it does not make the global state go away!

=> Still a lot of potential for complex entanglements.

#6 Don't stop improving too early

- You often can shrink code by a factor of 10, and make it more legible at the same time.
- But the cleanest and most elegant solutions do not always come to mind at once.
- That's OK. It's great to experience the pleasure to find a better solution multiple times.

So, keep going ...



SCALA 特点

- 结合面向对象和函数式编程模式，抽象性能好，组件重用性强
  - 每个值是一个对象
  - 每个函数也是一个值，函数也可作为输入输出参数
- 类对象组合灵活（composition by mixing with traits）
- 数据类型解析，通过模式匹配（case class, pattern matching）
- 类型自动分析，软件健壮性好（type inference）
- 面向JVM，和Java可以自由结合联合使用

SCALA优势

- 并行分布计算（parallel, concurrent, distributed computing）
  - 数据集合上的并行计算
  - 对未来事件的抽象，支持异步并发处理（Futures, Promises, async）
  - 基于消息传递的并行分布计算（Actors，Akka）
- 构建组件系统，可扩展性强（component systems, scalable）
- 面向JVM，可自由利用Java库
- 语法简洁，开发效率高，软件测试和可维护性强

软件安装

- 开发环境 Scala-IDE(Eclipse) http://scala-ide.org
- Typesafe reactive platform (Web-based Tool; Templates)
  http://www.typesafe.com/platform
- SBT(Scala Building Tool; Command-line-based)
  http://www.scala-sbt.org/release/docs/Getting-Started/Setup.html
- 相关网页
  - http://www.scala-lang.org
  - http://www.typesafe.com
  - http://akka.io
  - http://www.playframework.com

如何开始一个SCALA项目

1. 建立项目模版
2. Scala Eclipse-based IDE导入项目进行编辑
3. SBT 修改更新项目设置
4. 重复2，3步进行维护调整

自学参考

- Scala Standard Library API http://www.scala-lang.org/api/
- Scala School, a tutorial by Twitter http://twitter.github.com/scala_school/
- A Tour of Scala http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html
- Scala Overview on StackOverflow http://stackoverflow.com/tags/scala/info
- Scala By Example http://www.scala-lang.org/docu/files/ScalaByExample.pdf
- Scala Cheatsheet http://docs.scala-lang.org/cheatsheets/
- Books

Programming in Scala. Martin Odersky, Lex Spoon and Bill Venners. 2nd edition. Artima 2010

Scala for the Impatient. Cay Horstmann. Addison-Wesley 2012

Scala in Depth. Joshua D. Suereth. Manning 2012.

Scala编程的基本构件

齐琦

(Slides adopted from the "Scala the simple parts" by Martin Odersky)

Scala 社区的成长

Scala's user community is pretty large for its age group.

~ 100'000 developers

~ 200'000 subscribers to Coursera online courses.
#13 in RedMonk Language Rankings

Many successful rollouts and happy users.

But Scala is also discussed more controversially than usual for a language at its stage of adoption.

Why?

同样有着争议

Internal controversies: Different communities don't agree what programming in Scala should be.

External complaints:

"Scala is too academic"

"Scala has sold out to industry"

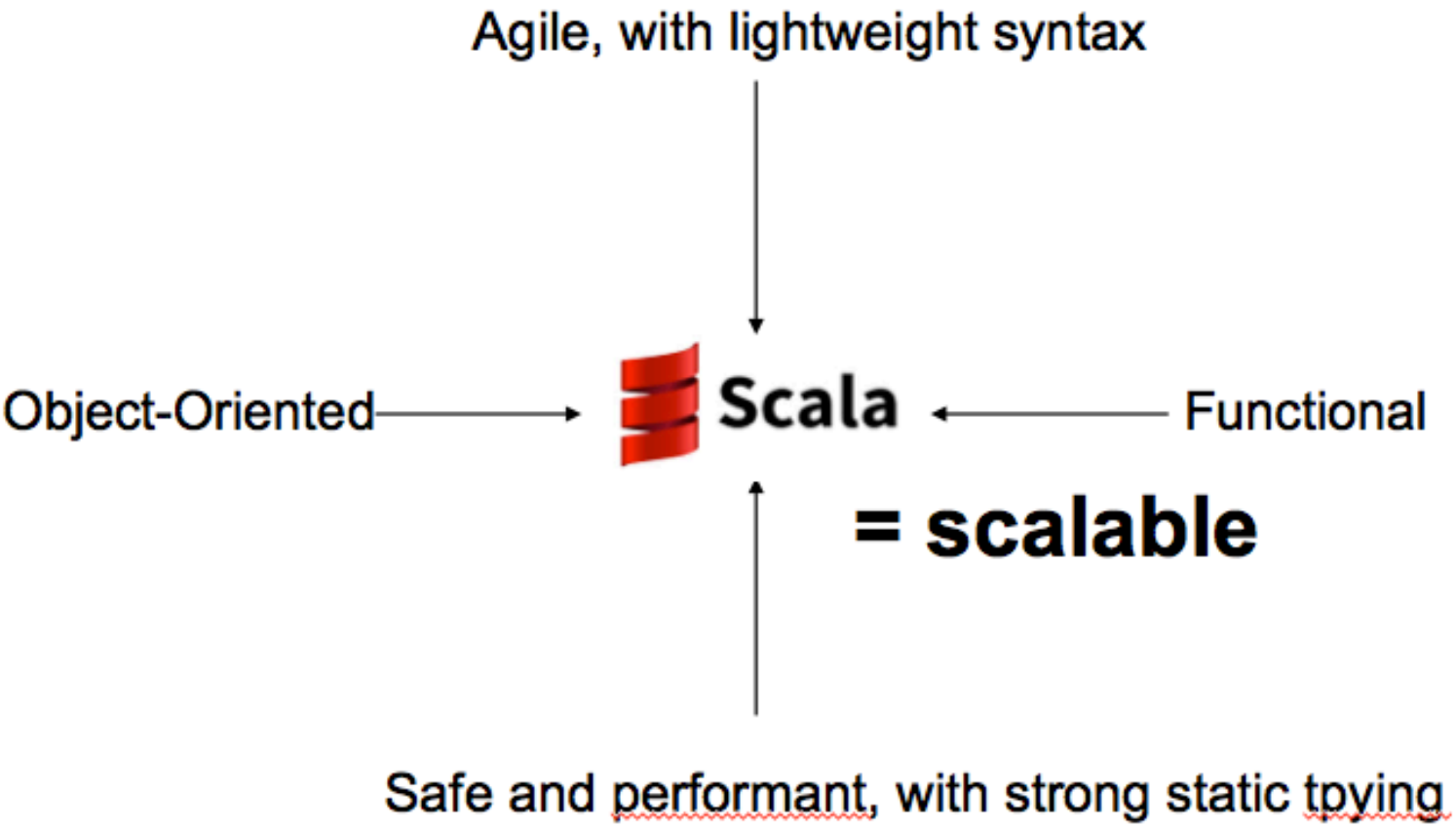
"Scala's types are too hard"

"Scala's types are not strict enough"

"Scala is everything and the kitchen sink"

Signs that we have not made clear enough what the essence of programming in Scala is.

可扩展的语言(Scalable)



What is "Scalable"?

1st meaning: "Growable"

- can be molded into new languages by adding libraries (domain specific or general)

See: "Growing a language" (Guy Steele, 1998)

2nd meaning: "Enabling Growth"

- can be used for small as well as large systems
- allows for smooth growth from small to large.

A Growable Language

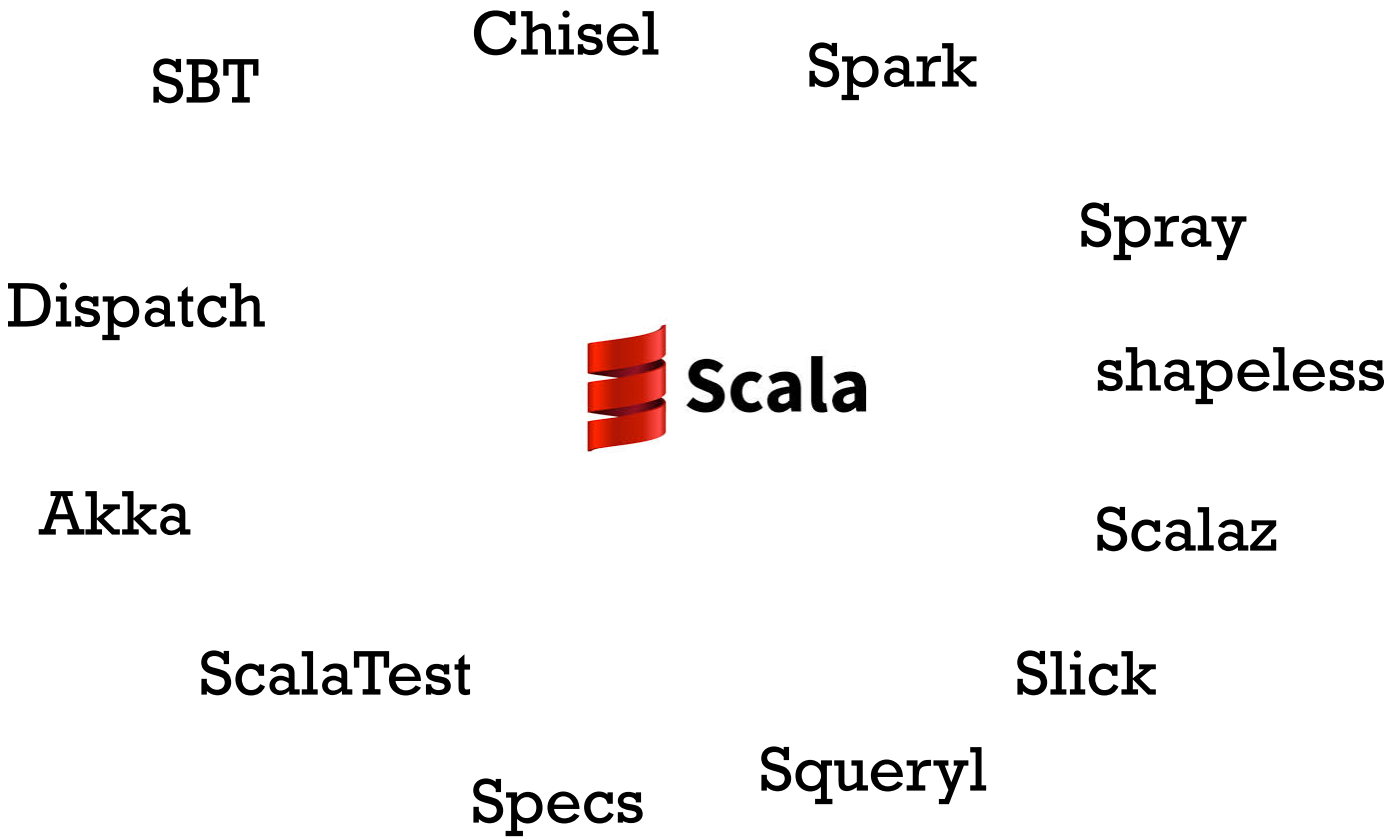
- Flexible Syntax
- Flexible Types
- User-definable operators
- Higher-order functions
- Implicits

...

Make it relatively easy to build new DSLs on top of Scala

And where this fails, we can always use the macro system (even though so far it's labeled experimental)

A Growable Language



Growable = Good?

In fact, it's a double edged sword.

- DSLs can fracture the user community ("The Lisp curse")
- Besides, no language is liked by everyone, no matter whether its a DSL or general purpose.
- Host languages get the blame for the DSLs they embed.

Growable is great for experimentation.

But it demands conformity and discipline for large scale production use.

A Language For Growth

- Can start with a one-liner.
- Can experiment quickly.
- Can grow without fearing to fall off the cliff.
- Scala deployments now go into the millions of lines of code.
  - Language works for very large programs
  - Tools are challenged (build times!) but are catching up.

"A large system is one where you do not know that some of its components even exist"

What Enables Growth?

Unique combination of Object/Oriented and Functional

Large systems rely on both.

Unfortunately, there's no established term for this

object/functional?

Would prefer it like this

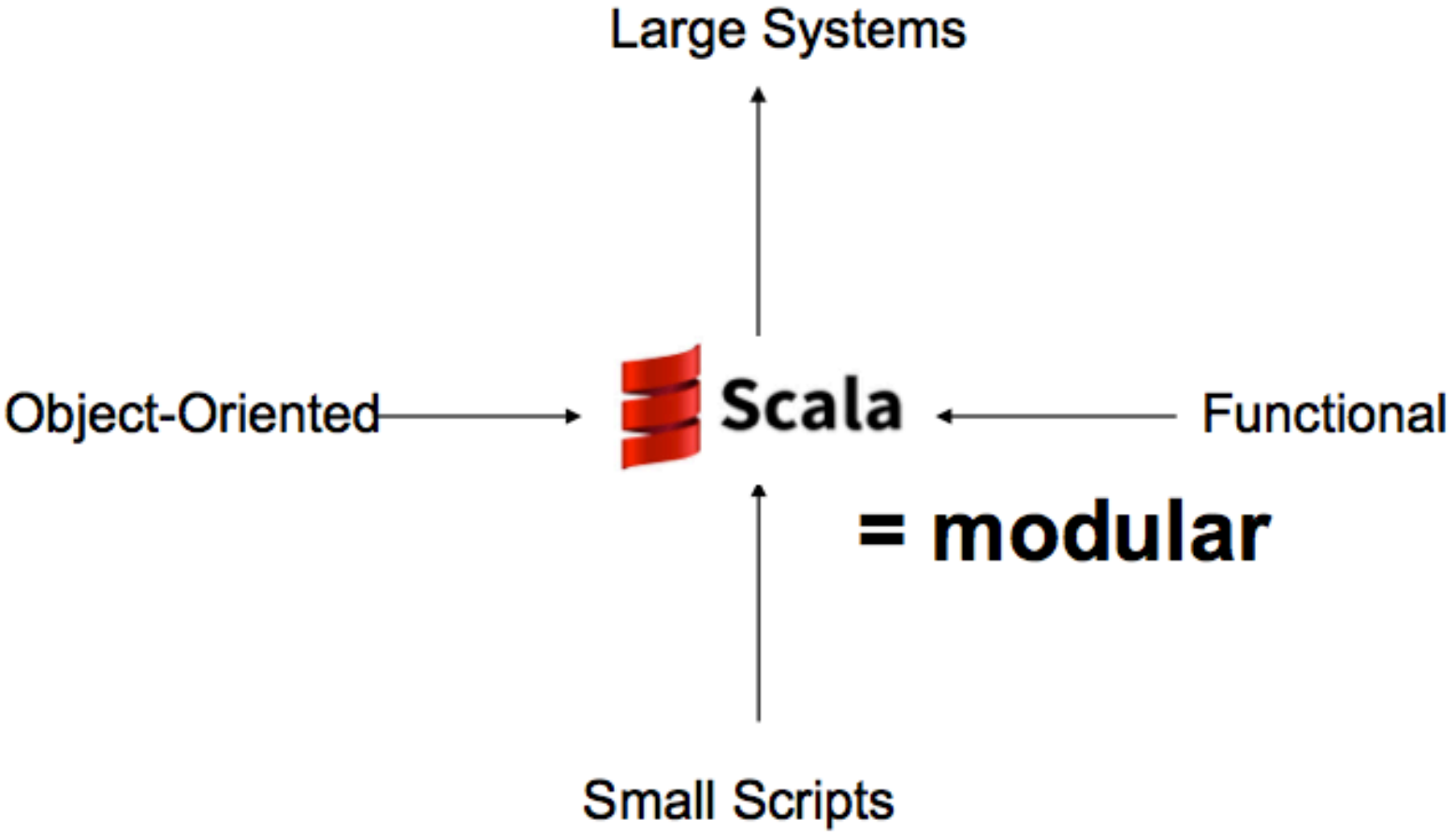
But unfortunately it's often more like this

How many FP people see OOP

How many OOP people see FP

And that's where we are ☺

另一个角度看：一种模块化的语言(a modular language)



（模块化的编程）Modular Programming

Systems should be composed from modules.

Modules should be *simple parts*（简单的构件） that can be combined in *many ways* to give interesting results.

- (Simple: encapsulates one functionality)

But that's old hat!

- Should we go back to Modula-2?
- Modula-2 was limited by the Von-Neumann Bottleneck (see John Backus' Turing award lecture).
- Today's systems need richer models and implementations.

FP is Essential for Modular Programming（函数式编程本质上体现了模块化编程）

Read:

"Why Functional Programming Matters"

(John Hughes, 1985).

Paraphrasing:

"Functional Programming is good because it leads to modules that can be combined freely."

Functions and Modules

Functional does not always imply *Modular*.

Some concepts in functional languages are at odds with modularity:

- Haskell's type classes
- OCaml's records and variants
- Dynamic typing? (can argue about this one)

Objects and Modules

Object-oriented languages are in a sense the successors of classical modular languages.

But *Object-Oriented* does not always imply *Modular* either.

Non-modular concepts in OOP languages:

- Smalltalk's virtual image.
- Monkey-patching
- Mutable state makes transformations hard.
- Weak composition facilities require external DI frameworks.
- Weak decomposition facilities encourage mixing domain models with their applications.

朝向模块化编程方向的努力

- 函数式编程(Functional programming)本质上反映了模块化编程，但是并不完全兼容。
- 面向对象编程语言从某种意义上说也是经典的模块化编程语言的后代，但也存在一些概念与之不相一致。
- Scala 把面向对象和函数式编程方式相结合的同时，努力实现一致的模块化的编程模式。
- 良好的模块化编程模式的好处之一就是具有良好的可扩展性。

Scala – The Simple Parts（简单的（基本的）构件）

Before discussing library modules, let's start with the simple parts in the language itself.

They correspond directly to the *fundamental actions* that together cover the essence of programming in Scala

compose – match – group – recurse – abstract – aggregate – mutate

As always: Simple ≠ Easy !

1. Compose(组合)

Everything is an expression
→ everything can be composed with everything else.

```
val category = if (age >= 18) "grownup" else "minor"

val result = tag match {
case “email” =>
        try getEmail()
        catch handleIOException
  case “postal” =>
        scanLetter()
}
```

2. Match（匹配）

Pattern matching（模式匹配，函数式的） decomposes data.
It's the dual of composition.

```
trait Expr
case class Number(n: Int) extends Expr
case class Plus(l: Expr, r: Expr) extends Expr

def eval(e: Expr): Int = e match {
  case Number(n)  => n
  case Plus(l, r) => eval(l) + eval(r)
}
```

Simple & flexible, even if a bit verbose.

The traditional OO alternative

```
trait Expr {
def eval: Int
}
case class Number(n: Int) extends Expr {
def eval = n
}
case class Plus(l: Expr, r: Expr) extends Expr {
def eval = l.eval + r.eval
}
```

OK if operations are fixed and few.

But mixes data model with "business" logic.

3. Group(分组)

Everything can be grouped and nested.

Static scoping discipline.

Two name spaces: Terms and Types.

Same rules for each.

```
def solutions(target: Int): Stream[Path] = {
  def isSolution(path: Path) =
    path.endState.contains(target)
  allPaths.filter(isSolution)
}
```

4. Recurse（递归）

Recursion let's us compose to arbitrary depths.

This is almost always better than a loop.

Tail-recursive functions are guaranteed to be efficient.

```
@tailrec
def sameLength(xs: List[T], ys: List[U]): Boolean =
  if (xs.isEmpty) ys.isEmpty
  else ys.nonEmpty && sameLength(xs.tail, ys.tail)
```

5. Abstract（抽象）

Functions are abstracted expressions.

Functions are themselves values.

Can be named or anonymous.

- **def isMinor(p: Person) = p.age < 18**
- **val (minors, adults) = people.partition(isMinor)**
- **val infants = minors.filter(_.age <= 3)**

(this one is pretty standard by now)

(even though scope rules keep getting messed up sometimes)

6. Aggregate(聚合)

Collection aggregate data.

Transform instead of *CRUD(create, read, update, delete).*

Uniform set of operations

Very simple to use

Learn one – apply everywhere

```
val people: Array[Person] = Array(Person("Bob", 22), Person("Carla", 7))
                                                  > people  : Array[fm.Person] = Array(Person(Bob,22), Perso
people.map(_.name)                                > res0: Array[String] = Array(Bob, Carla)

val nums = Set(1, 4, 5, 7)                        > nums  : Set[Int] = Set(1, 4, 5, 7)
nums.map(_ / 2)                                   > res1: Set[Int] = Set(0, 2, 3)

val roman = Map("I" -> 1, "V" -> 5, "X" -> 10)
                                                  > roman  : Map[String,Int] = Map(I -> 1, V -> 5, X -> 10)
roman map { case (l, d) => (d, l) }               > res2: Map[Int,String] = Map(1 -> I, 5 -> V, 10 -> X)
```

7. Mutate(转变)

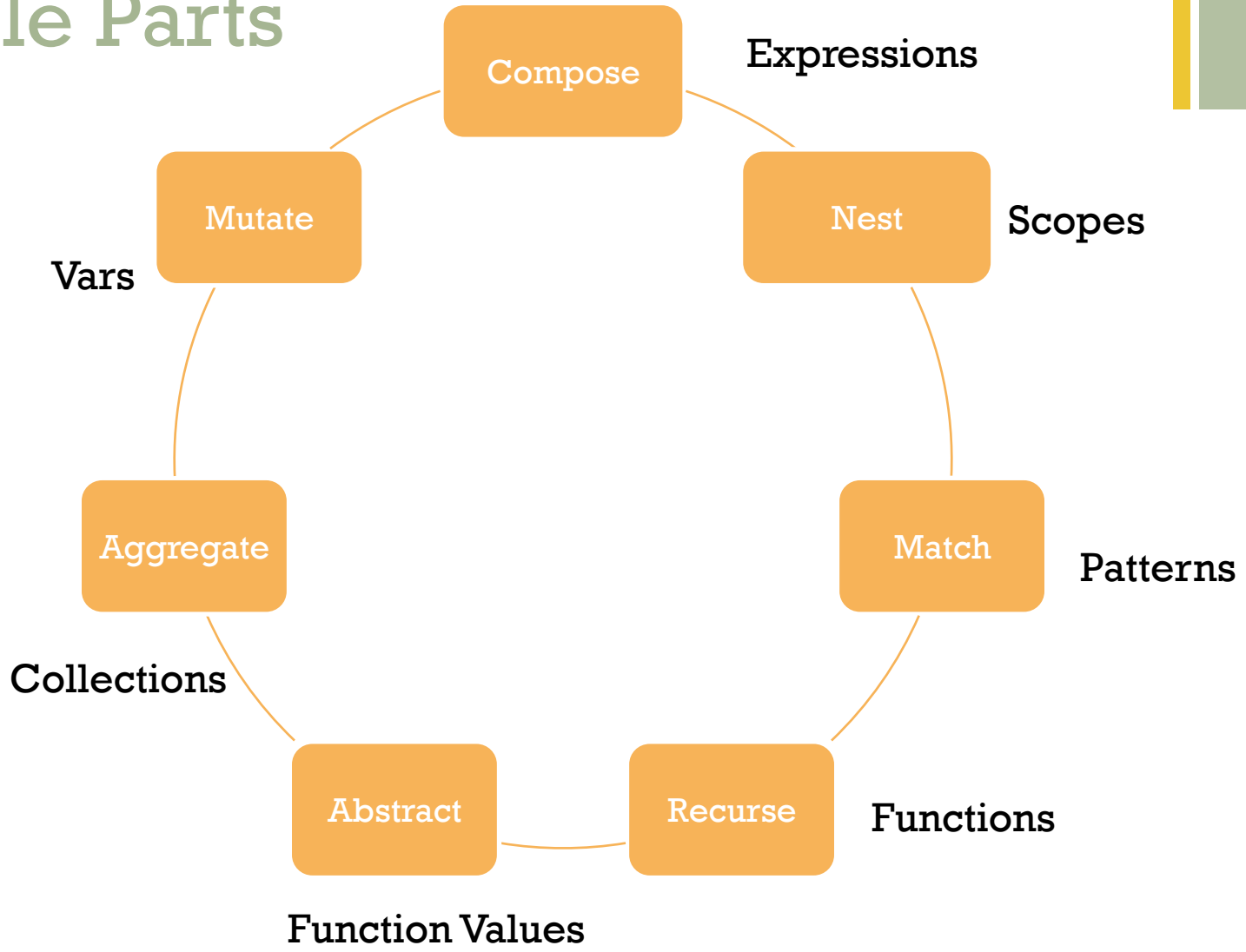
It's the last point on the list and should be used with restraint.

But are vars and mutation not anti-modular?

- Indeed, global state often leads to hidden dependencies.
- But used-wisely, mutable state can cut down on boilerplate and increase efficiency and clarity.

- **Boilerplate** is any text that is or can be reused in new contexts or applications without being greatly changed from the original.
样板（**boilerplate**）是指一系列可被重用的文本。

From Fundamental Actions to Simple Parts



Modules（模块）

Modules can take a large number of forms

- A function
- An object
- A class
- An actor
- A stream transform
- A microservice

Modular programming is putting the focus on how modules can be combined, not so much what they do.

In Scala, modules talk about values as well as types.

Features For Modular Programming

1. Our Vocabulary: Rich types with static checking and functional semantics
   - gives us the domains of discourse,
   - gives us the means to guarantee encapsulation,
   - see: "On the Criteria for Decomposing Systems into Modules" (David Parnas, 1972).

2. Start with Objects — atomic modules

3. Parameterize with Classes — templates to create modules dynamically

4. Mix it up with Traits — mixable slices of behavior

5. Abstract By Name

Members of a class or trait can be concrete or abstract.

Example: A Graph Library

```
trait Graphs {
  type Node
  type Edge
  def pred(e: Edge): Node
  def succ(e: Edge): Node
  type Graph <: GraphSig
  def newGraph(nodes: Set[Node], edges: Set[Edge]): Graph

  trait GraphSig {
    def nodes: Set[Node]
    def edges: Set[Edge]
    def outgoing(n: Node): Set[Edge]
    def incoming(n: Node): Set[Edge]
    def sources: Set[Node]
  }
}
```

Where To Use Abstraction?

Simple rule:

- Define what you know, leave abstract what you don't.
- Works universally for values, methods, and types.

```
trait AbstractModel extends Graphs {
  class Graph(val nodes: Set[Node],
              val edges: Set[Edge]) extends GraphSig {
    def outgoing(n: Node) = edges filter (pred(_) == n)
    def incoming(n: Node) = edges filter (succ(_) == n)
    lazy val sources = nodes filter (incoming(_).isEmpty)
  }
  def newGraph(nodes: Set[Node], edges: Set[Edge]) =
    new Graph(nodes, edges)
}
```

Encapsulation(封装包囊) = Parameterization（参数设定）

Two sides of the coin:

1. Hide an implementation
2. Parameterize an abstraction

```
trait ConcreteModel extends Graphs {
  type Node = Person
  type Edge = (Person, Person)
  def succ(e: Edge) = e._1
  def pred(e: Edge) = e._2
}

class MyGraph extends AbstractModel with ConcreteModel
```

在Scala-IDE里运行这行程序。

6. Abstract By Position

Parameterize classes and traits.

- `class List[+T]` (apple is a fruit; a list of apples is a list of fruits too.)
- `class Set[T]`
- `class Function1[-T, +R]`

- **List[Number]**
- **Set[String]**
- **Function1[String, Int]**

Variance expressed by +/- annotations

A good way to explain variance is by mapping to abstract types.

Modelling Parameterized Types

```
class Set[T] { ... }        →  class Set { type $T }

Set[String]                 →  Set { type $T = String }
```

```
class List[+T] { ... }      →  class List { type $T }
List[Number]                →  List { type $T <: Number }
```

Parameters(参数) → Abstract members（抽象成员）

Arguments（参数实体化）→ Refinements （进一步明确）

7. Keep Boilerplate Implicit

Implicit parameters are a rather simple concept

But they are surprisingly versatile(多用途的)!

Can represent a *typeclass*:

- `def min(x: A, b: A)(implicit cmp: Ordering[A]): A`

Implicit Parameters

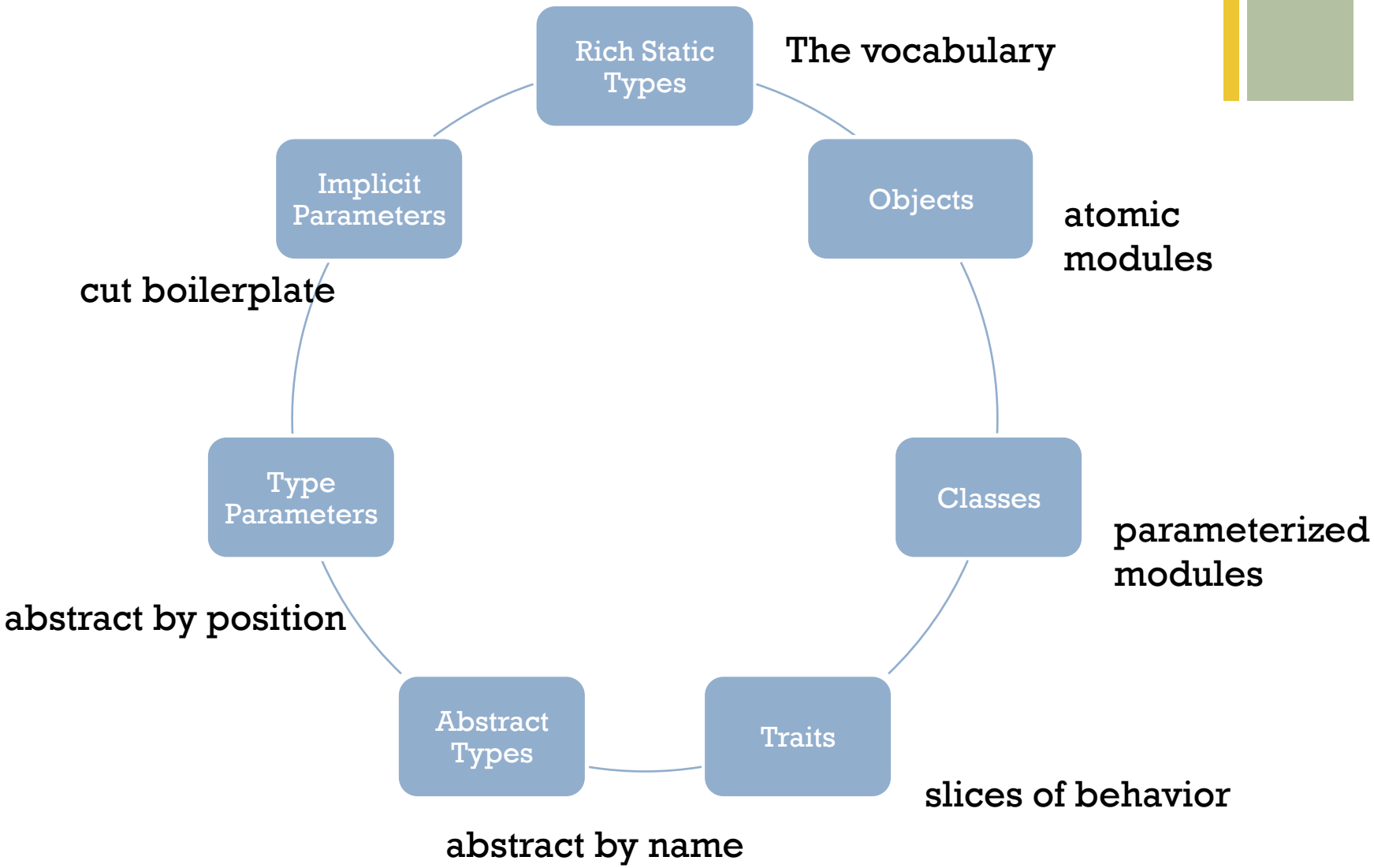
Can represent a *context*:

- **def typed(tree: untpd.Tree, expected: Type)(implicit ctx: Context): Type**
- **def compile(cmdLine: String)**
  (implicit defaultOptions: List[String]): Unit

Can represent a *capability*:

- **def accessProfile(id: CustomerId)**
  (implicit admin: AdminRights): Info

Module Parts



程序展示

Scala List

```
val nums = (1 to 10).toList

val total = nums.
  filter(x => x % 2 == 0).
  map(x => x * x).
  foldLeft(0)((a, b) => a + b)

val allByAllSum = nums.
  flatMap (n1 => nums.map (n2 => n1 * n2)).
  foldLeft(0)((a, b) => a + b)
```

程序展示

Scala List Using For

```
val nums = (1 to 10).toList

val totalf = (for {
    n <- nums if n % 2 == 0
} yield n * n).sum

val allByAllSumf = (for {
    n1 <- nums
    n2 <- nums
} yield n1 * n2).sum
```

程序展示

Scala Future Using For

```
val usdQuote = future { connection.getCurrentValue(USD) }
val chfQuote = future { connection.getCurrentValue(CHF) }

val purchase = for {
    usd <- usdQuote
    chf <- chfQuote
    if isProfitable(usd, chf)
} yield connection.buy(amount, chf)

purchase onSuccess {
    case _ => println("Purchased " + amount + " CHF")
}
```

程序展示

Scala Try Using For

```
import scala.util.{Try, Success, Failure}

  def divide: Try[Int] = {
     val dividend = Try(Console.readLine("Enter an Int that you'd like to divide:\n").toInt)
     val divisor = Try(Console.readLine("Enter an Int that you'd like to divide by:\n").toInt)

     val problem = for {
        x <- dividend
        y <- divisor
     } yield x/y

     problem match {
        case Success(v) =>
           println("Result of " + dividend.get + "/"+ divisor.get +" is: " + v)
           Success(v)
        case Failure(e) =>
           println("You must've divided by zero or entered something that's not an Int. Try again!")
           println("Info from the exception: " + e.getMessage)
           divide
      }
}
```