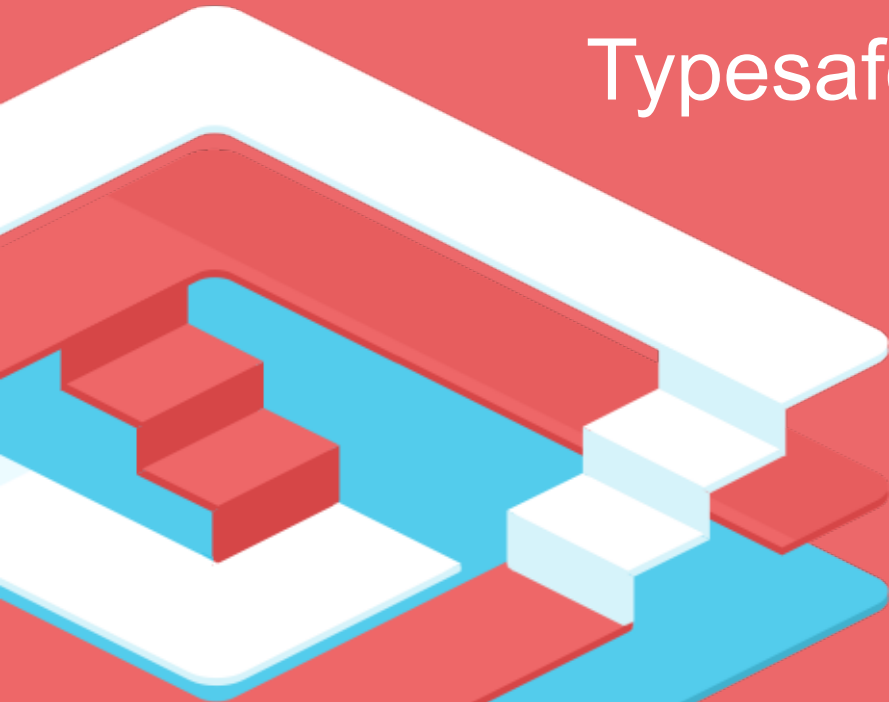


Scala - The Simple Parts

Martin Odersky
Typesafe and EPFL



10 Years of Scala

Subject: **ANNOUNCEMENT: The Scala Programming Language**

Newsgroups: **gmane.comp.lang.scala**

Date: 2004-01-20 17:25:18 GMT (10 years, 15 weeks, 6 days, 3 hours and 12 minutes ago)



We'd like to announce availability of the first implementation of the Scala programming language. Scala smoothly integrates object-oriented and functional programming. It is designed to express common programming patterns in a concise, elegant, and type-safe way. Scala introduces several innovative language constructs. For instance:

- Abstract types and mixin composition unify ideas from object and module systems.
- Pattern matching over class hierarchies unifies functional and object-oriented data access. It greatly simplifies the processing of XML trees.
- A flexible syntax and type system enables the construction of advanced libraries and new domain specific languages.

At the same time, Scala is compatible with Java. Java libraries and frameworks can be used without glue code or additional declarations.

The current implementation of Scala runs on Java VM. It requires JDK 1.4 and can run on Windows, MacOS, Linux, Solaris, and most other operating systems. A .net version of Scala is currently under development.

For further information and downloads, please visit:



Grown Up?

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?

Controversies

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”

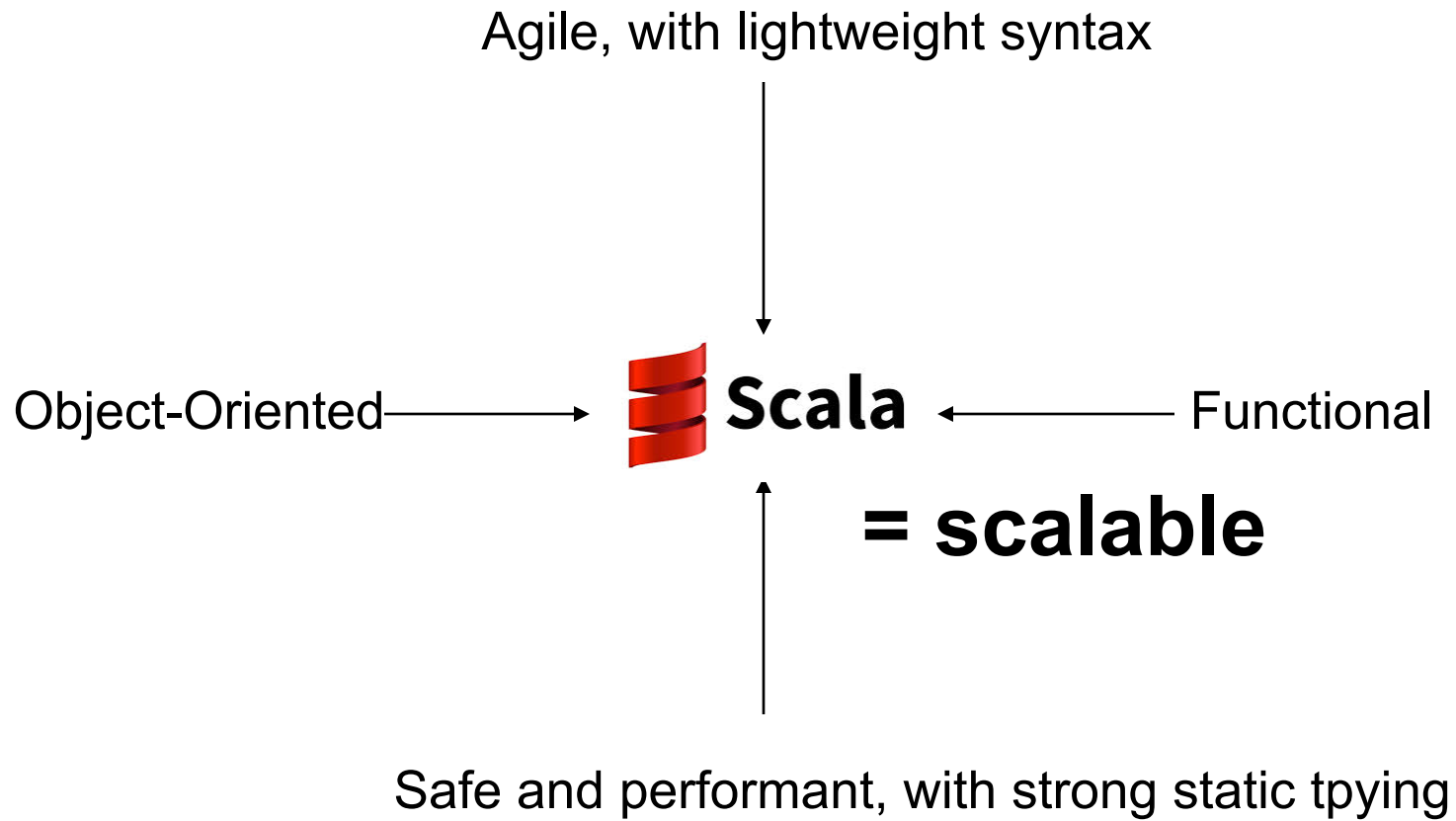
“Scala's types are too hard”

“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.

The Picture So Far



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

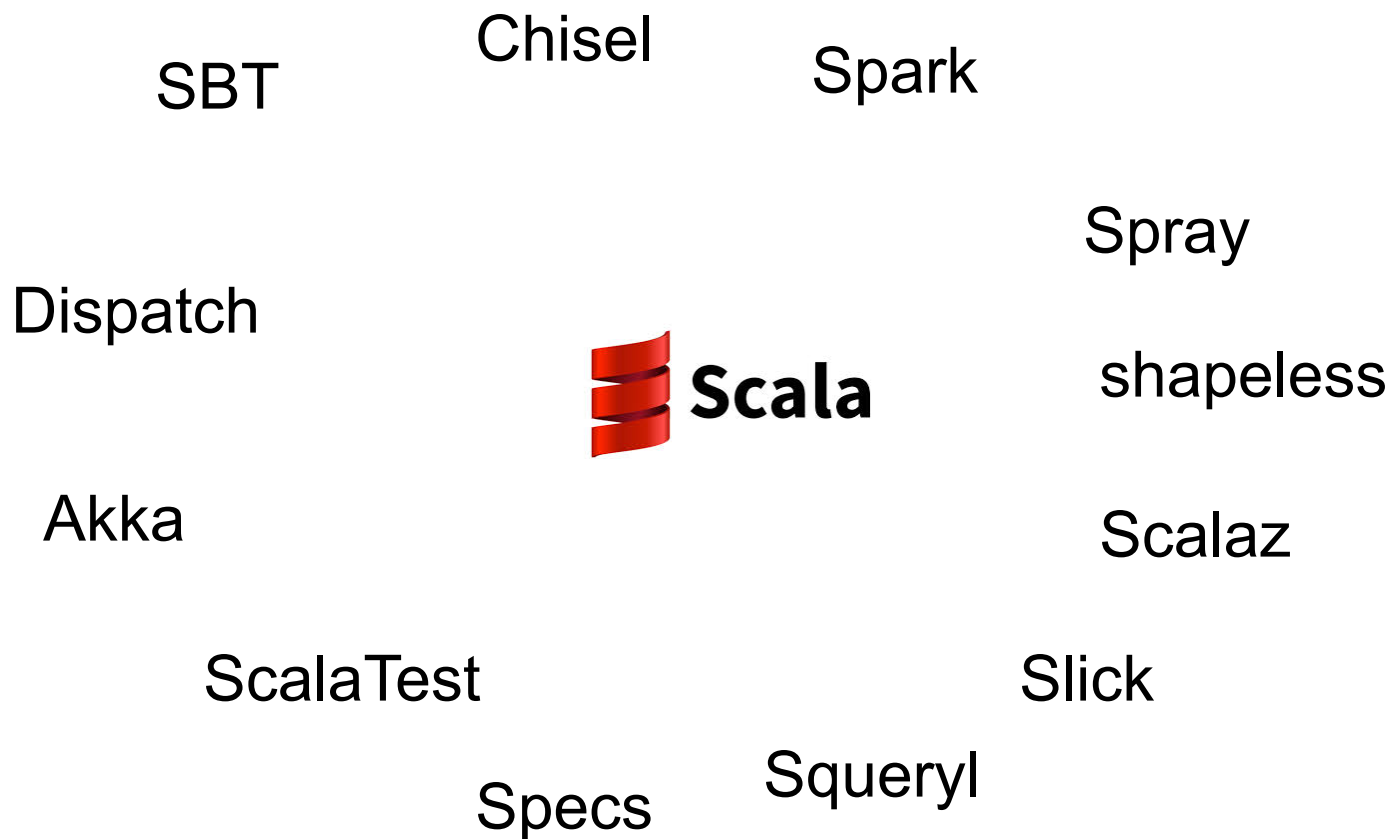
- Flexible Syntax
- Flexible Types
- User-definable operators
- Higher-order functions
- Implicits

...



- 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 😊

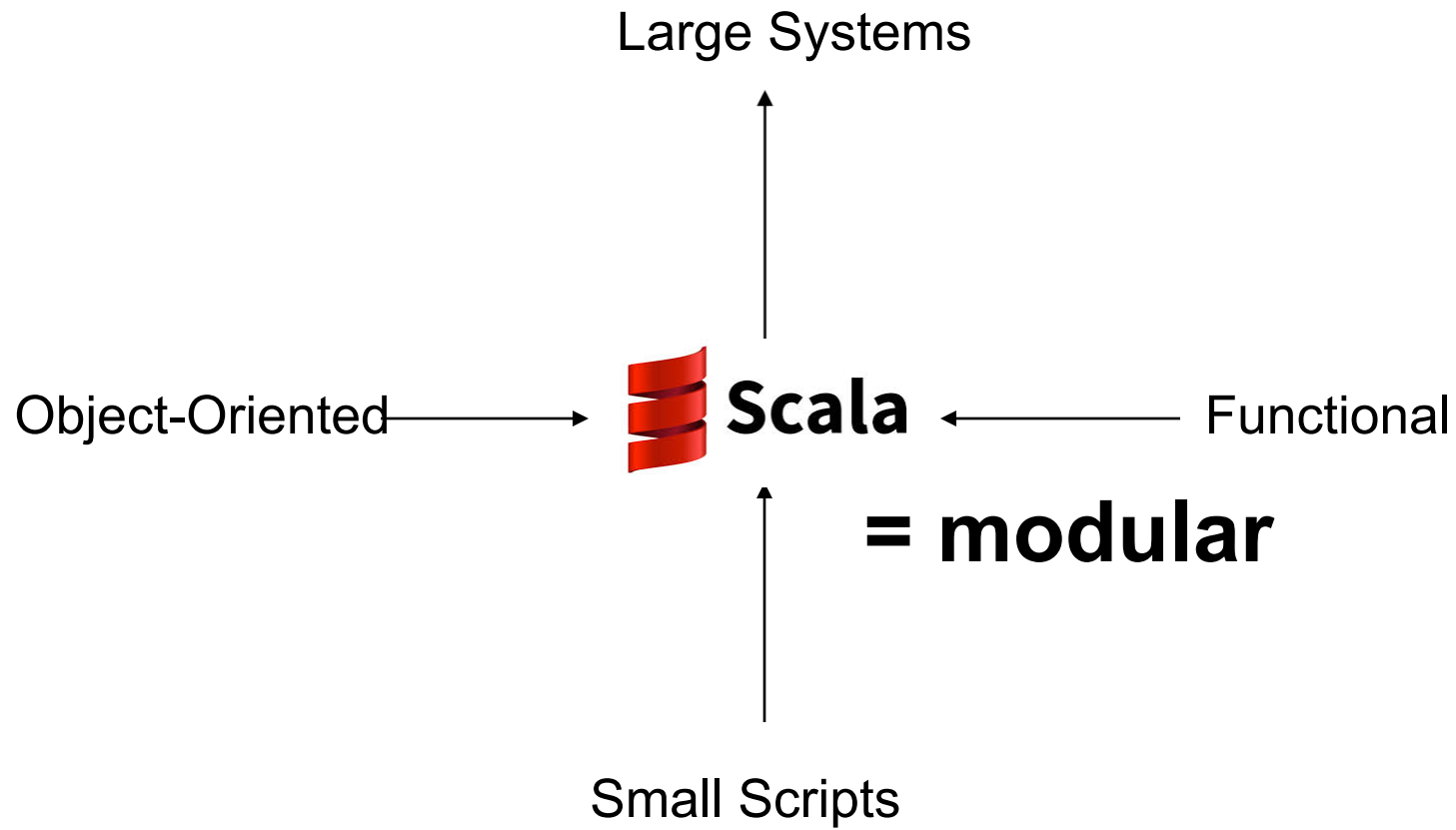
Scala's Role in History 😊

2003 - A drunken Martin Odersky sees a Reese's Peanut Butter Cup ad featuring somebody's peanut butter getting on somebody else's chocolate and has an idea. He creates Scala, a language that unifies constructs from both object oriented and functional languages. This pisses off both groups and each promptly declares jihad.

(from:James Iry: A Brief, Incomplete, and Mostly Wrong History of Programming Languages)



Another View: A Modular Language



Modular Programming

- Systems should be composed from modules.
- Modules should be *simple parts* 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

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:
 - Aggregation constructs may be lacking or 2nd class
 - Sometimes, assumes global namespace (e.g. type classes)
 - 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:
 - Monkey-patching
 - Mutable state makes transformations hard.
 - Weak composition facilities require external DI frameworks.
 - Weak decomposition facilities encourage mixing domain models with their applications.

Scala – The Simple Parts

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

Here are seven simple building blocks that can be combined in flexible ways.

Together, they cover much of Scala's programming in the small.

As always: Simple \neq Easy !

#1 Expressions

Everything is an expression

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

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

#2: Scopes

- Everything can be 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)  
}
```

Tip: 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.

Tip: Find meaningful names!

- There's a lot of value in meaningful names.
- Easy to add them using inline vals and defs.

```
val sources = jp.getRawClasspath.filter(  
  _.getEntryKind == IClasspathEntry.CPE_SOURCE)  
def workspaceRoot =  
  ResourcesPlugin.getWorkspace.getRoot  
def filesOfEntry(entry: Set[File]) =  
  flatten(workspaceRoot.findMember(entry.getPath))  
sources.iterator flatMap filesOfEntry
```


#3: Patterns and Case Classes

```
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 in the small

But mixes data model with “business” logic

#4: Recursion

- Recursion is almost always better than a loop.
- Simple fallback: Tail-recursive functions
- Guaranteed to be efficient

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

#5: Function Values

- Functions are 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 Collections

- Extensible set of collection types
- Uniform operations
- Transforms instead of CRUD
- Very simple to use
- Learn one – apply everywhere!

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

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

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

Collection Objection

“The type of map is ugly / a lie”

▶ def lengthCompare(len: Int): Int

Compares the length of this mutable indexed sequence to a test value.

▶ def map[B](f: (A) ⇒ B): Array[B]

[use case] Builds a new collection by applying a function to all elements of this array.

▶ def max: A

[use case] Finds the largest element.

Collection Objection

“The type of map is ugly / a lie”

def **lengthCompare**(len: [Int](#)): [Int](#)

Compares the length of this mutable indexed sequence to a test value.

def **map**[B](f: (A) ⇒ B): [Array](#)[B]

[use case]

Builds a new collection by applying a function to all elements of this array.

B the element type of the returned collection.

f the function to apply to each element.

returns a new array resulting from applying the given function *f* to each element of this array and collecting the results.

Implicit information This member is added by an implicit conversion from [Array](#)[T] to [ArrayOps](#)[T] performed by method [genericArrayOps](#) in [scala.Predef](#).

Definition Classes [TraversableLike](#) → [GenTraversableLike](#) → [FilterMonadic](#)

▼ Full Signature

def **map**[B, That](f: (T) ⇒ B)(implicit bf: [CanBuildFrom](#)[[Array](#)[T], B, That]): That

Why CanBuildFrom?

- Why not define it like this?

```
class Functor[F[_], T] {  
  def map[U](f: T => U): F[U]  
}
```

- Does not work for arrays, since we need a class-tag to build a new array.
- More generally, does not work in any case where we need some additional information to build a new collection.
- This is precisely what's achieved by CanBuildFrom.

#7 Vars

- Are vars not anti-modular?
- Indeed, global mutable state often leads to hidden dependencies.
- But used-wisely, mutable state can cut down on annoying boilerplate and increase clarity.

Where I use State

In dotc, a newly developed compiler for Scala:

- caching *lazy vals, memoized functions, interned names, LRU caches.*
- persisting *once a value is stable, store it in an object.*
- copy on write *avoid copying untpd.Tree to tpd.Tree.*
- fresh values *fresh names, unique ids*
- typer state *2 vars: current constraint & current diagnostics (versioned, explorable).*

Why Not Use a Monad?

The fundamentalist functional approach would mandate that typer state is represented as a monad.

Instead of now:

```
def typed(tree: untpd.Tree, expected: Type): tpd.Tree
def isSubType(tp1: Type, tp2: Type): Boolean
```

we'd write:

```
def typed(tree: untpd.Tree, expected: Type):
  TyperState[tpd.Tree]
def isSubType(tp1: Type, tp2: Type):
  TyperState[Boolean]
```

Why Not Use a Monad?

Instead of now:

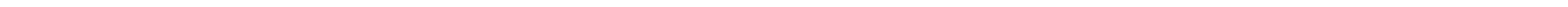
```
if (isSubType(t1, t2) && isSubType(t2, t3)) result
```

we'd write:

```
for {  
  c1 <- isSubType(t1, t2)  
  c2 <- isSubType(t2, t3)  
  if c1 && c2  
} yield result
```

Why would this be better?

A Question of Typing

Clojure	Scala	Haskell	Idris	Coq
				
<i>syntax</i>	<i>arguments</i>	<i>effects</i>	<i>values</i>	<i>correctness</i>

Statically checked properties

None of the 5 languages above is “right”.
It’s all a question of tradeoffs.

Forms of Modules

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**.

Two stages: **programming** and **deployment**.

I am going to concentrate on the first.

Scala's Modular Roots

Modula-2	First language I programmed intensively First language for which I wrote a compiler.
Modula-3	Introduced universal subtyping
Haskell	Type classes → Implicits
SML modules	

<i>Object</i>	\cong	<i>Structure</i>
<i>Class</i>	\cong	<i>Functor</i>
<i>Trait</i>	\cong	<i>Signature</i>
<i>Abstract Type</i>	\cong	<i>Abstract Type</i>
<i>Refinement</i>	\cong	<i>Sharing Constraint</i>

Features Supporting Modular Programming

1. Rich types with functional semantics.

- gives us the domains of discourse

2. Static typing.

- Gives us the means to guarantee encapsulation
- Read

“On the Criteria for Decomposing Systems into Modules”
(David Parnas, 1972)

3. Objects

4. Classes and Traits

#5 Abstract Types

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 = Parameterization!

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
```

#6 Parameterized Types

```
class List[+T]  
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] { ... }  
Set[String]
```

```
→ class Set { type $T }  
→ Set { type $T = String }
```

```
class List[+T] { ... }  
List[Number]
```

```
→ class List { type $T }  
→ List { type $T <: Number }
```

Parameters → Abstract members

Arguments → Refinements

#7 Implicit Parameters

Implicit parameters are a simple concept
But they are surprisingly versatile.

Can represent a *typeclass*:

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

#7 Implicit Parameters

Can represent a *context*

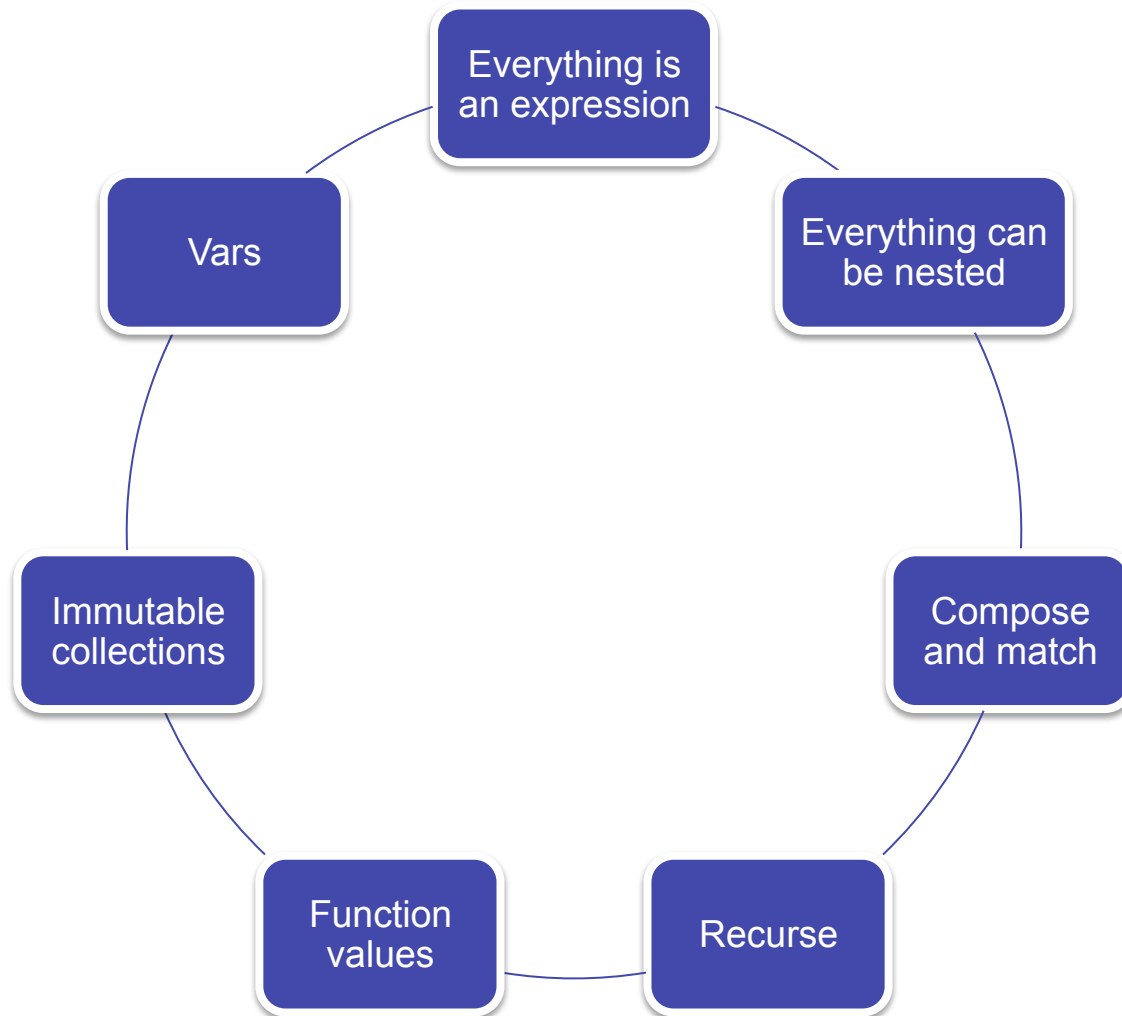
```
def typed(tree: untpd.Tree, expected: Type)
  (implicit ctx: Context): Type
```

```
def compile(cmdLine: String)
  (implicit defaultOptions: List[String]): Unit
```

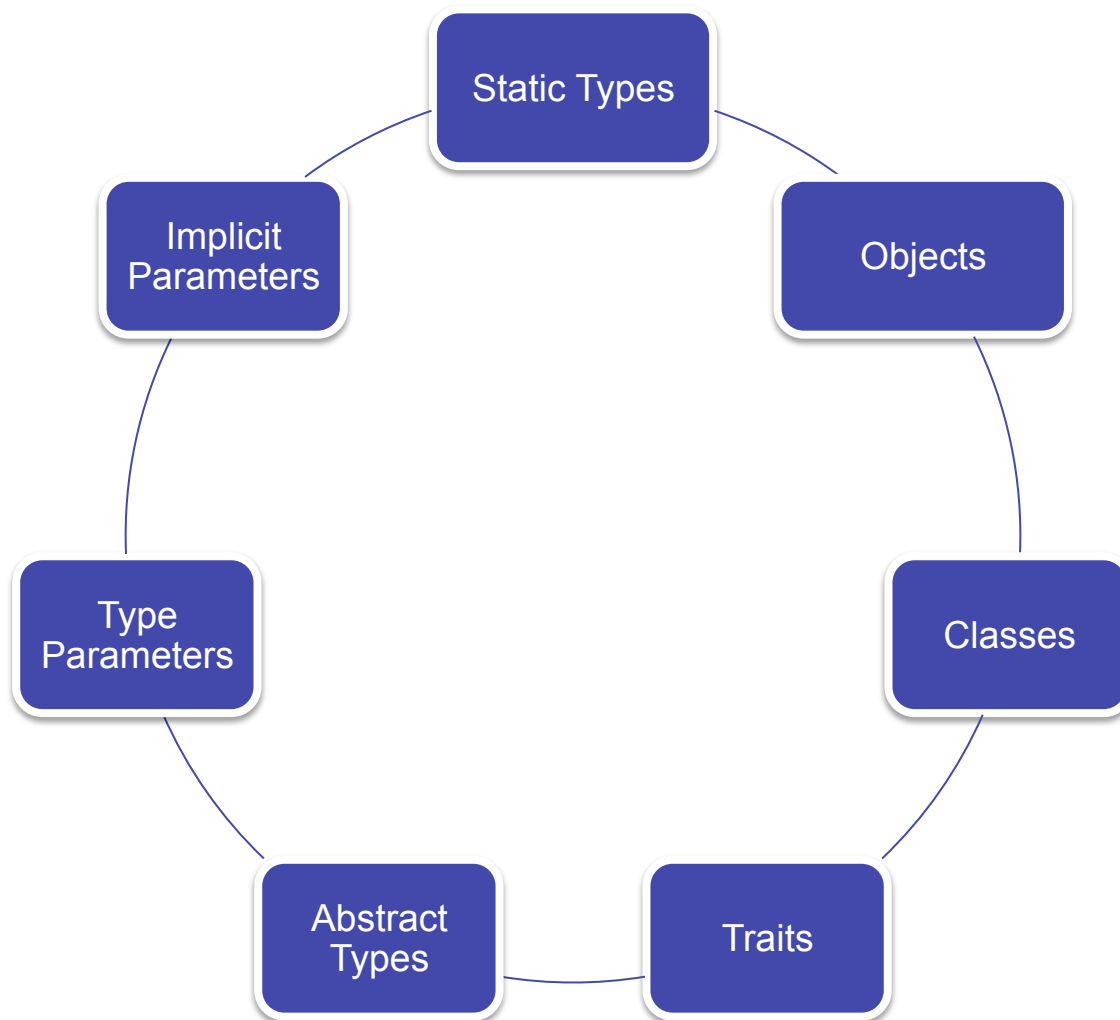
Can represent a *capability*:

```
def accessProfile(id: CustomerId)
  (implicit admin: AdminRights): Info
```

Simple Parts Summary



Module Parts Summary

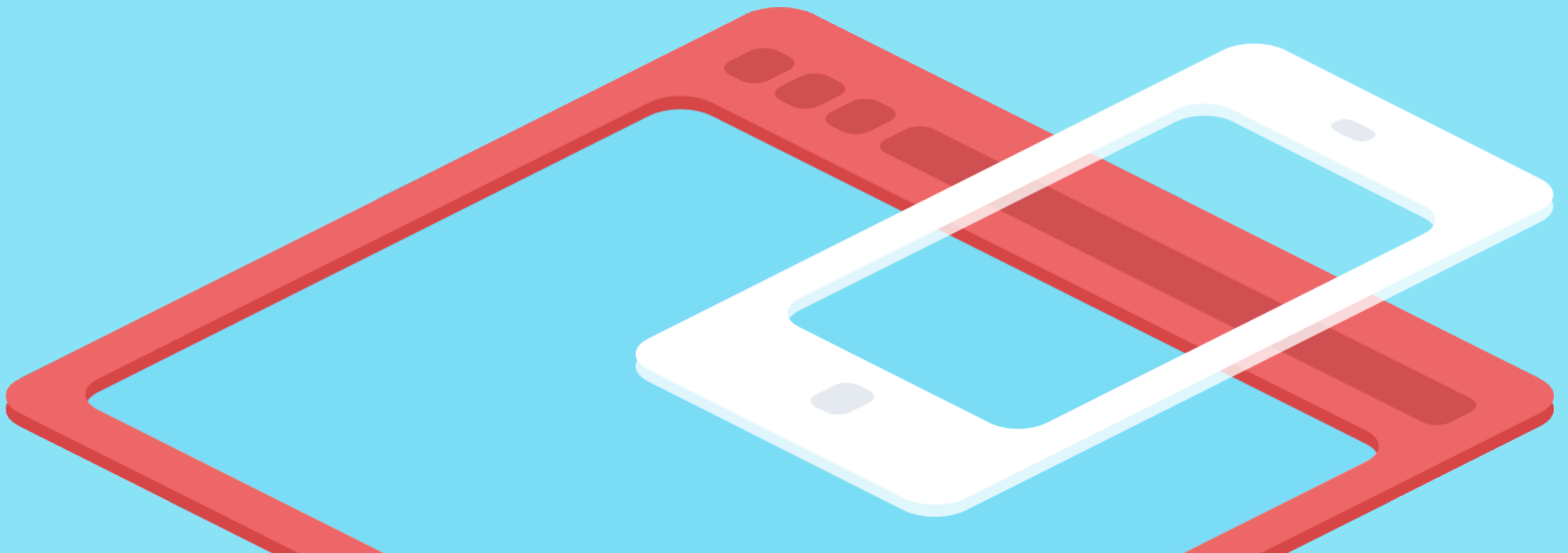


Other Parts

- Here are parts that are either not as simple or do not work as seamlessly with the core:
 - Implicit conversions
 - Existential types
 - Structural types
 - Higher-kinded types
 - Macros
- All of them are under language feature flags or experimental flags.
- This makes it clear they are outside the core.
- My advice: Avoid, unless you have a clear use case
 - e.g, you use Scala as a host for a DSL or other language.

Thank You

Follow us on twitter:
@typesafe



Simple Parts Summary

Language

- Expressions
- Scopes and Nesting
- Case Classes and Patterns
- Recursion
- Function Values
- Collections
- Vars

Library

- Static Types
- Objects
- Classes
- Traits
- Abstract Types
- Type Parameters
- Implicit Parameters

A fairly modest (some might say: boring) set of parts that can be combined in flexible ways.

Caveat: This is my selection, not everyone needs to agree.