OOP versus Type Classes

How to stop worrying and start loving both parametricity and the vtable 😅

Alexandru Nedelcu @ Scala Love in the City — 2021


Type Classes
OOP
Scala

Agenda

For the brave …

- Abstraction
- What is OOP?
- What are Type Classes?
- Ideological Clash
- Conversions (OOP to Type Classes, Type Classes to OOP)
- Recipes & Best Practices

Abstraction

SortedSet[A]

List[A]

Array[A]

Vector[A]

String

Option[A]

Future[A]

Try[A]

Long

IO[A]

Either[E,A]

Abstraction

Definition

- *“to draw away, withdraw, remove”*
- *“to consider as a general object or idea without regard to matter”*
- *“a member of an idealized subgroup when contemplated according to the abstracted quality which defines the subgroup”*

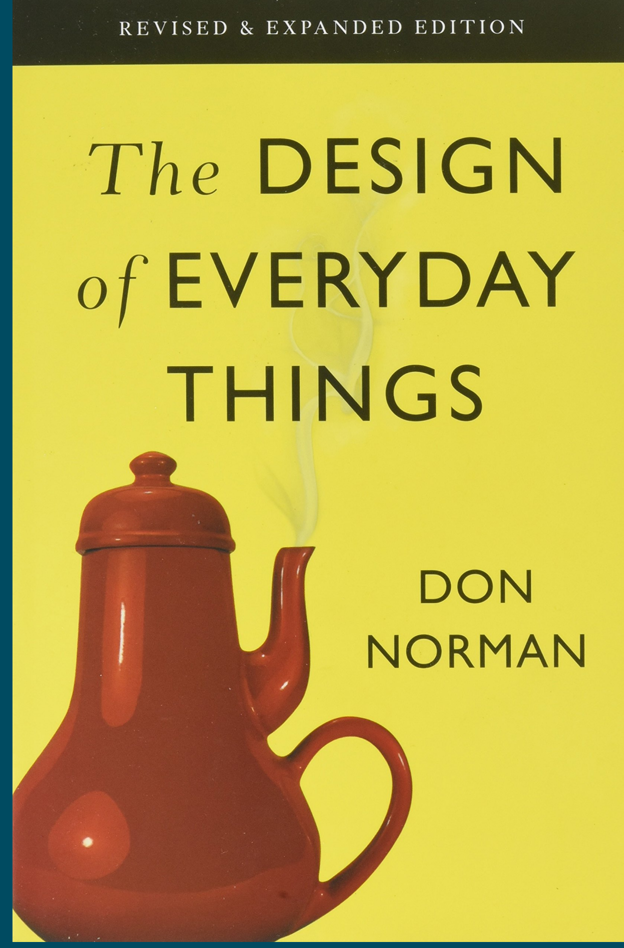
Abstraction

In the context of software development …

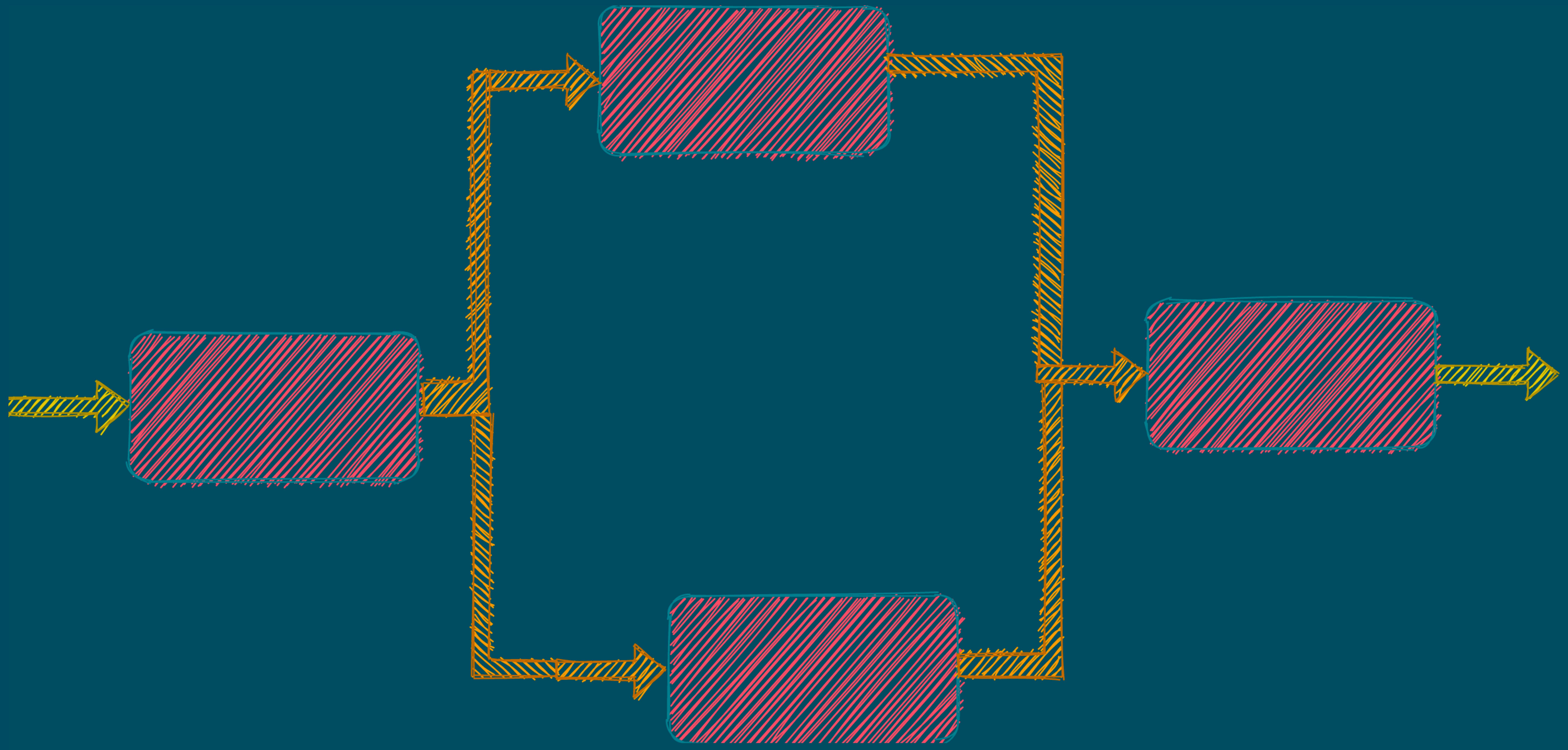
- **idealization**, removing details that aren't relevant, working with idealized models that focus on what's important
- **generalization**, looking at what objects or systems have in common that's of interest, such that we can transfer knowledge, recipes, proofs

Managing complexity is like ...

Black Box Abstraction



Black Box Abstraction



$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[loop above, "1_A"] \arrow[dr, "g \circ f"'] & B \arrow[d, "g"] \arrow[loop above, "1_B"] \\
& C \arrow[loop below, "1_C"]
\end{tikzcd}
$$

Abstraction

- **OOP ➡** best for black boxes
- **(Static) FP ➡** best for composing (white?) boxes

What is OOP?

What is OOP?

- **Subtype polymorphism**, via single (dynamic) dispatch
- **Encapsulation** (hiding implementation details)
- **Inheritance** of classes or prototypes

What is OOP?

```
SortedSet[A] <: Set[A]
```

Liskov substitution principle

Are OOP and FP orthogonal?

Spoiler: Yes, but with caveats!

Are OOP and FP orthogonal?

```scala
case class Customer(
  name: FullName,
  emailAddress: EmailAddress,
  pass: PasswordHash
)
```

Are OOP and FP orthogonal?

```scala
trait Iterator[+A] {
  def hasNext: Boolean
  def next(): A
}
```

Are OOP and FP orthogonal?

```
trait Iterator[+A] {
  def hasNext: IO[Boolean]
  def next: IO[A]
}
```

"FP removes one important dimension of complexity – To understand a program part (a function), you need no longer account for the possible executions that can lead to that program part"

Martin Odersky

Are OOP and FP orthogonal?

```scala
final class Metrics(counter: AtomicLong) {
  def touch(): Long =
    counter.incrementAndGet()
}

// FP version
final class Metrics(ref: Ref[IO, Long]) {
  def touch: IO[Long] = ???
}
```

Are OOP and FP orthogonal?

```scala
object Metrics {
  // Note the internals are now exposed
  def touch(ref: Ref[IO, Long]): IO[Long] = ???
}
```

What are Type Classes?

What are Type Classes?

Parametric Polymorphism

```scala
def identity[A](a: A): A = a

// Compare and contrast with this one — how
// many implementations can this have?

def foo(a: String): String
```

What are Type Classes?

Ad-hoc Polymorphism

```scala
def sum[A](list: List[A]): A = ???

// Should work for integers
sum(List(1, 2, 3)) //= 6

// Should work for strings
sum(List("Hello, ", "World")) //= Hello, World

// Should work for empty lists
sum(List.empty[Int]) //= 0
sum(List.empty[String]) //= ""
```

What are Type Classes?

OOP wannabe?

```scala
trait Combine {
  def ++(other: Combine): Combine
}
```

What are Type Classes?

OOP wannabe?

```scala
trait Combine {
  def ++(other: Combine): Combine
}

"Hello, " ++ 10
```

Fail

Liskov is sad 😢

What are Type Classes?

OOP wannabe — one more try

```
trait Combine[Self] { self: Self =>
  def ++(other: Self): Self
}

class String extends Combine[String] { ... }

def sum[A](list: List[Combine[A]]): A = ???
```

What are Type Classes?

OOP wannabe — one more try

```scala
trait Combine[Self] { self: Self =>
  def ++(other: Self): Self
}

class String extends Combine[String] { ... }

def sum[A](list: List[Combine[A]]): A = ???
```

Fail

OOP developer is sad 😢

What are Type Classes?

Ad-hoc Polymorphism for the win

```
trait Combinable[A] {
  def combine(x1: A, x2: A): A
  def empty: A
}

def sum[A](list: List[A], fns: Combinable[A]): A =
  list.foldLeft(fns.empty)(fns.combine)
```

What are Type Classes?

Ad-hoc Polymorphism for the win

```scala
// Oops, the jig is up
trait Monoid[A] {
  def combine(x1: A, x2: A): A
  def empty: A
}

object Monoid {
  // Visible globally.
  // WARN: multiple monoids are possible for integers ;-)
  implicit object intSumInstance extends Monoid[Int] {
    def combine(x1: Int, x2: Int) = x1 + x2
    def empty = 0
  }
}
```

What are Type Classes?

Ad-hoc Polymorphism for the win

combine(x, combine(y, z)) = combine(combine(x, y), z)

combine(x, empty) = combine(empty, x) = x

What are Type Classes?

Ad-hoc Polymorphism for the win

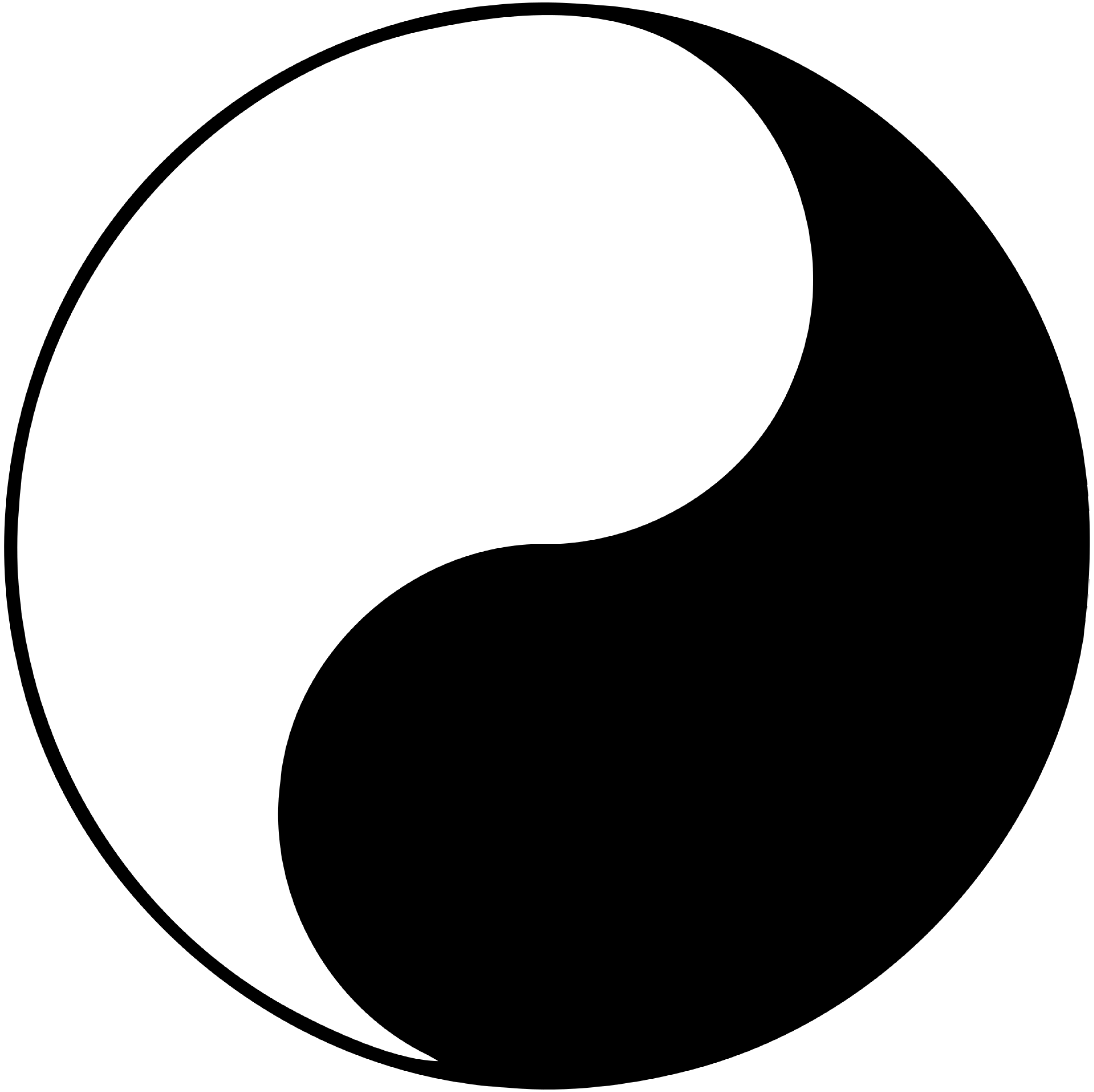
```scala
def sum[A](list: List[A])(implicit m: Monoid[A]): A =
  list.foldLeft(m.empty)(m.combine)

// Or with some syntactic sugar
def sum[A : Monoid](list: List[A]): A = ???
```

With parametric polymorphism,
types dictate the implementation!

This intuition, that the signature describes precisely what the implementation does, is what static FP developers call “**parametricity**”

Ideological clash



Ideological clash

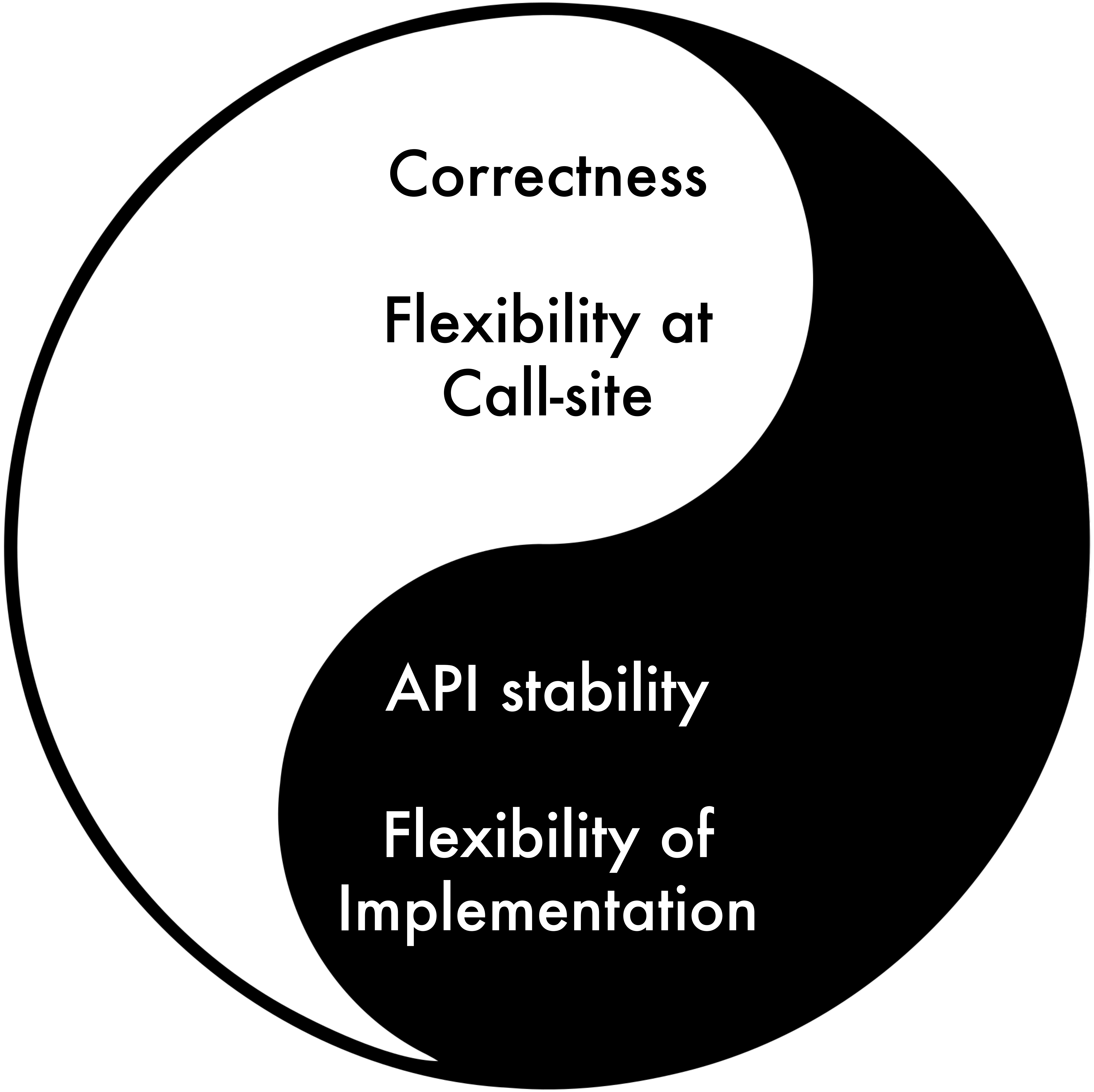
OOP values

- flexibility of implementation
- backwards compatibility
- black boxes
- resource management

Ideological clash

FP values

- flexibility at the call site
- correctness
- dumb data structures
- dealing with data
- composition


Correctness
Flexibility at Call-site
API stability
Flexibility of Implementation

Ideological clash

Degenerate cases

```
trait Actor {
  def send(message: Any): Unit
}
```

```
def identity[A](a: A): A
```

Type Class Superpowers

Type Class Superpowers

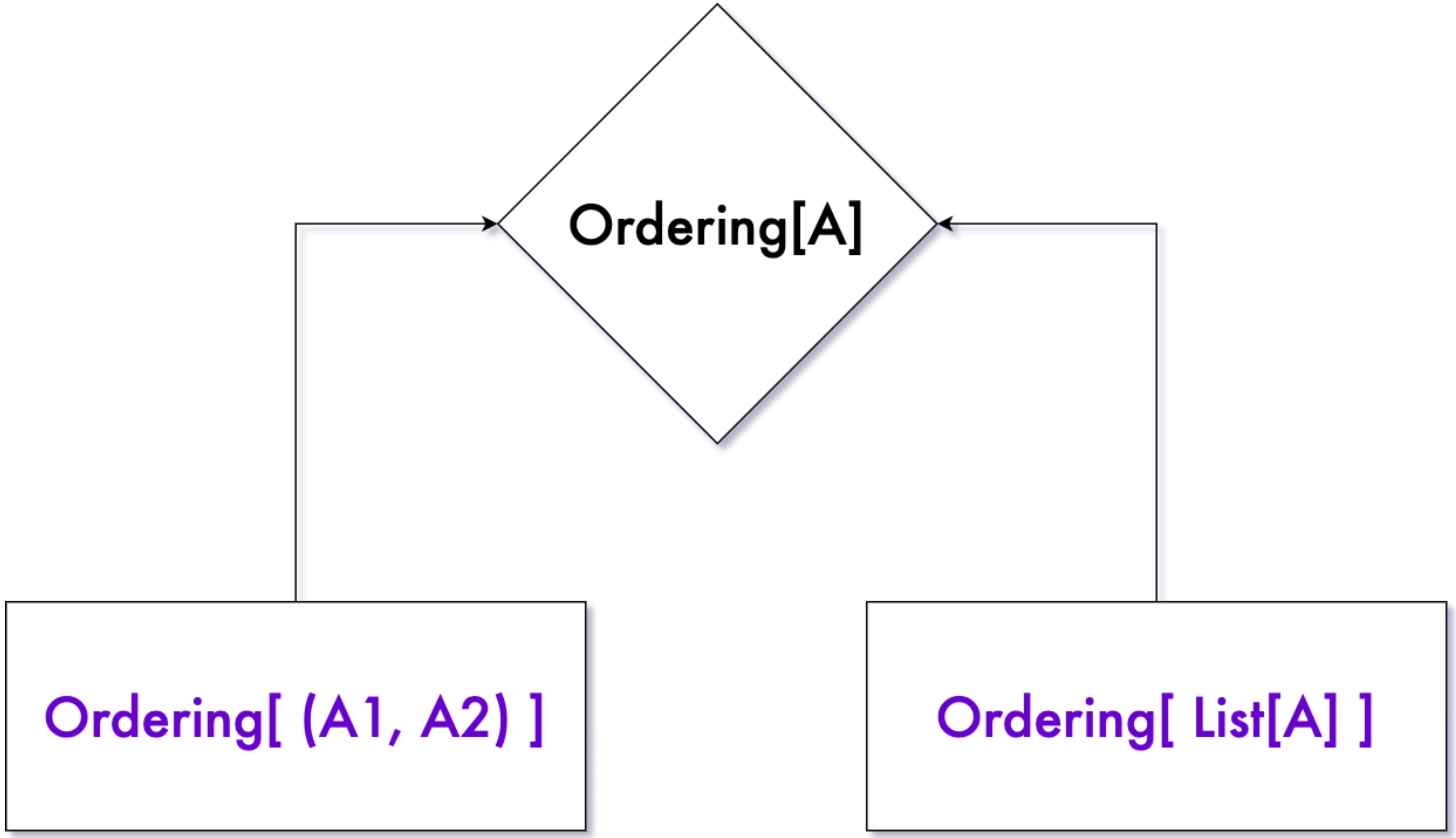
"Extend" types we don't control

```scala
case class Age(value: Int)

//...
implicit val ord: Ordering[Age] =
  (x: Age, y: Age) => x.value.compareTo(y.value)
```

Type Class Superpowers

Automatic instance derivation



Type Class Superpowers

"Extend" types you don't control

```scala
implicit def ordTuple2[A, B](implicit
  A: Ordering[A],
  B: Ordering[B]): Ordering[(A, B)] = {

  case ((a1, b1), (a2, b2)) =>
    A.compare(a1, a2) match {
      case 0 => B.compare(b1, b2)
      case v => v
    }
}
```

Type Class Superpowers

"Extend" types you don't control

```scala
implicit def ordList[A](implicit A: Ordering[A]): Ordering[List[A]] =
  (xs: List[A], ys: List[A]) => {
    xs.zip(ys)
      .collectFirst {
        case (x, y) if A.compare(x, y) != 0 =>
          A.compare(x, y)
      }
      .getOrElse(xs.length.compareTo(ys.length))
  }
```

Converting between styles

Converting OOP to Type Clases

```scala
// OOP-style interface
trait JSONSerialization {
  def toJSON: JsValue
}

// Type-class
trait JSONSerialization[A] {
  def toJSON(a: A): JsValue
}
```

Converting OOP to Type Clases

```scala
// OOP
trait Iterator[+A] {
  def hasNext: Boolean
  def next(): A
}

// Type Class
trait Iterator[F[_]] {
  type Cursor[_]
  def start[A](collection: F[A]): Cursor[A]
  def hasNext[A](cursor: Cursor[A]): Boolean
  def next(cursor: Cursor[A]): (A, Cursor[A])
}
```

Converting OOP to Type Clases

```scala
trait Iterator[F[_]] {
  type Cursor[A]
  def start[A](collection: F[A]): IO[Cursor[A]]
  def hasNext[A](cursor: Cursor[A]): IO[Boolean]
  def next(cursor: Cursor[A]): IO[A]
}

// Perfectly equivalent to this OOP class:
trait Iterator[+A] {
  def hasNext: IO[Boolean]
  def next: IO[A]
}
```

Converting OOP to Type Clases

- Sometimes Type Classes expose internals wide open
- Type Classes introduce the need to have Higher-Kinded Types (HKTs)

Type Classes → OOP

Converting Type Clases to OOP

```scala
trait FlatMap[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}

trait Monad[F[_]] extends FlatMap[F] {
  def pure[A](a: A): F[A]
}
```

Converting Type Clases to OOP

```scala
trait FlatMap[A] {
  def flatMap[B](f: A => FlatMap[B]): FlatMap[B]
}

// WRONG! We can't compose monadic types like this
Option(1).flatMap(x => Future(x + 1))
```

Converting Type Clases to OOP

```scala
trait FlatMap[A] {
  def flatMap[B](f: A => FlatMap[B]): FlatMap[B]
}

// WRONG! We can't compose monadic types like this
Option(1).flatMap(x => Future(x + 1))
```

Fail

Liskov is sad 😢

Converting Type Clases to OOP

```
trait FlatMap[A, Self[_] <: FlatMap[A, Self]] {
  self: Self[_] =>

  def flatMap[B](f: A => Self[B]): Self[B]
}
```

Converting Type Clases to OOP

```
trait FlatMap[A, Self[_]] <: FlatMap[A, Self]] {
  self: Self[_] =>

  def flatMap[B](f: A => Self[B]): Self[B]
}
```

Fail

OOP developer is sad 😢

Type Classes → OOP
is not always possible!

A “*design pattern*” is usually a name for an abstraction that your programming language doesn’t let you turn into a library.

Recipes & Best Practices

Recipes & Best Practices

Agenda

- Use Type Classes for data serialization
- Use Type Classes for expressing data constructors (factories)
- Use Type Classes for reusability of dumb data structures
- Type Class instances must be coherent (globally unique)
- Avoid "orphaned" Type Class instances (but do what you must)
- Type Classes must not keep state
- Use OOP for managing resources / information hiding
- Use Principle of Least Power — Default to OOP 🤕

Use Type Classes for data serialization

Recipes & Best Practices

Use Type Classes for data serialization

```scala
trait LogShow[A] {
  def logShow(a: A): LogMessage
}
```

Use Type Classes for expressing data constructors (factories)

Recipes & Best Practices

Use Type Classes for expressing data constructors (factories)

```scala
def sequence(list: List[IO[A]]): IO[List[A]] = ???

def sequence(list: Iterable[IO[A]]): IO[???]
```

Recipes & Best Practices

Use Type Classes for expressing data constructors (factories)

```scala
trait CollectionBuilder[Coll[_]] {
  // Buffer is used for building the collection,
  // it can be dirty / mutable
  type Buffer[A]
  // We need a way to iterate over the collection
  def iterable[A](coll: Coll[A]): Iterable[A]
  // Buffer data constructor
  def newBuffer[A]: Buffer[A]
  def append[A](buf: Buffer[A], elem: A): Buffer[A]
  def build[A](buf: Buffer[A]): Coll[A]
}
```

Recipes & Best Practices

Use Type Classes for expressing data constructors (factories)

```scala
object CollectionBuilder {
  // Sample instance
  implicit object forList extends CollectionBuilder[List] {
    type Buffer[A] = ListBuffer[A]

    def iterable[A](coll: List[A]) = coll
    def newBuffer[A] = ListBuffer.empty[A]
    def append[A](buf: Buffer[A], elem: A) = buf += elem
    def build[A](buf: Buffer[A]) = buf.toList
  }
}
```

Recipes & Best Practices

Use Type Classes for expressing data constructors (factories)

```scala
def sequence[Coll[_]](list: Coll[IO[A]])
  (implicit cb: CollectionBuilder[Coll]): IO[Coll[A]] = {

  cb.iterable(list)
    .foldLeft(IO(cb.newBuffer[A])) { (acc, e) =>
      acc.map(cb.append(_, e))
    }
    .map(cb.build)
}
```

Use Type Classes for reusability of dumb data structures

Recipes & Best Practices

Use Type Classes for reusability of dumb data structures

```scala
case class BinaryTree[+A](
  value: A,
  left: Option[BinaryTree[A]],
  right: Option[BinaryTree[A]]
)
```

Recipes & Best Practices

Use Type Classes for reusability of dumb data structures

```scala
object SortedSet {
  def fromList[A: Ordering](list: List[A]): BinaryTree[A] = ???
  def contains[A: Ordering](set: BinaryTree[A], value: A): Boolean = ???
}

// Second variant
object InefficientSet {
  def fromList[A](list: List[A]): BinaryTree[A] = ???
  def contains(set: BinaryTree[A], value: A): Boolean
}
```

Type Class instances must be coherent (globally unique)

Recipes & Best Practices

Type Class instances must be coherent (globally unique)

```
val set: BinaryTree[Int] = SortedSet.fromList(???)
{
  import my.pkg.implicits._
  SortedSet.contains(10)
}
```

Recipes & Best Practices

Type Class instances must be coherent (globally unique)

Correctness issue!

It's fine to have exceptions,
if well encapsulated

Caveat: dumb data structures can be misleading

Sometimes invariants set by the used functions are too important.

Recipes & Best Practices

Caveat: dumb data structures can be misleading

```
// Inefficient
InefficientSet.contains(
  SortedSet.fromList(???),
  111
)
// Malfunction
SortedSet.contains(
  InefficientSet.fromList(???),
  222
)
```

Recipes & Best Practices

Caveat: dumb data structures can be misleading

```scala
// Notice the Ordering restriction:
case class SortedSet[+A : Ordering](
  value: A,
  left: Option[SortedSet[A]],
  right: Option[SortedSet[A]]
)
```

Type Classes must not keep state

Recipes & Best Practices

Type Classes must not keep state

```
trait RegistrationService[F[_]] {
  def registerUser(user: User): F[Unit]
}
```

Recipes & Best Practices

Type Classes must not keep state

```scala
trait RegistrationService[F[_]] {
  def registerUser(user: User): F[Unit]
}

object RegistrationService {
  def apply[F[_]: Monad](
    db: UserDB[F],
    es: EmailService[F]): RegistrationService[F] = ???
}
```

Recipes & Best Practices

Type Classes must not keep state

```
trait RegistrationService[F[_], -Env] {
  def registerUser(user: User, env: Env): F[Unit]
}
```

Recipes & Best Practices

Type Classes must not keep state

```scala
trait RegistrationService[F[_], -Env] {
  def registerUser(user: User, env: Env): F[Unit]
}

object RegistrationService {
  implicit def instance[F[_]: Monad]
  : RegistrationService[F, UserDBEnv[F] with EmailServiceEnv[F]] = ???
}

trait UserDBEnv[F[_]] { def udb: UserDB[F] }
trait EmailServiceEnv[F[_]] { def es: EmailService[F] }
```

Recipes & Best Practices

Type Classes must not keep state

```
object RegistrationService {

  implicit def fake[F[_]]: RegistrationService[F, Unit] = ???
}
```

Recipes & Best Practices

Type Classes must not keep state

```scala
trait RegistrationService[Env[_[_]]] {
  def registerUser[F[_]: Monad](env: Env[F], user: User): F[Unit]
}

object RegistrationService {
  type Env[F[_]] = UserDBEnv[F] with EmailServiceEnv[F]
  implicit val instance: RegistrationService[Env] = ???
}
```

Use OOP for managing resources / information hiding

Recipes & Best Practices

Use OOP for managing resources / information hiding

```scala
trait RegistrationService[F[_]] {
  def registerUser(user: User): F[Unit]
}

object RegistrationService {
  def apply[F[_]: Monad](
    db: UserDB[F],
    es: EmailService[F]): RegistrationService[F] = ???
}
```

Recipes & Best Practices

Use OOP for managing resources / information hiding

```scala
trait RegistrationService[F[_]] {
  def registerUser(user: User): F[Unit]
}

object RegistrationService {
  def apply[F[_]: Monad](
    db: UserDB[F],
    es: EmailService[F]): RegistrationService[F] = ???
}
```

Liskov is happy 😁

Use Principle of Least Power 🤕

Use the Principle of Least Power 🤕

- You have a **Complexity Budget**
- **All abstraction has a cost!**
  (used tooling, learning curve, comprehension)

Use the Principle of Least Power 🤕

- Don't use a type class, if an OOP class or a higher-order function would do
  - Scala is not Haskell
- All type classes need to be defended
- All type parameters need to be defended
- All abstractions need to be defended
  - Beware false abstractions

Next Steps, Questions?

- Follow my blog: alexn.org
- Checkout the Typelevel ecosystem, Cats, Cats-Effect, Monix
- Join Typelevel's Gitter to talk about FP: gitter.im/typelevel/cats