A tale of two type-systems

Aesa Kamar

Once upon a time

Combine good parts of Java with Haskell

And then

Cool companies and projects emerge

But now

I want a strongly typed language
with Java-like syntax
with functional elements

Goals

Subtype Polymorphism

Instance Polymorphism

XP with a Compiler

Syntax Check I

```
trait X
trait Y extends X
trait Z[A]
```

Known Types:
- Capitalized
- End of Alphabet
- Aqua

Type Parameters
- Capitalized
- Start of Alphabet
- Teal

Syntax Check II

Type Annotations
- Can be Concrete
- Can be Parameters
- Can be Function Tpes

```
def f[B](xb: X ⇒ B): B
```

Functions can have Type Params

Syntax Check III

Functions can have implicit Arguments/Parameters

```
def g(implicit iy: Y) : Y
```

Syntax Check IV

This is how an ADT looks
- “Sum Type”
- “Coproduct”
- “Algebraic Data Type”

```
trait  Option[A]
object None          extends Option[Nothing]
class  Some[A](a:A) extends Option[A]
```

Colors!
- Trait
- Object
- Class

Syntax Check V

Extension methods via “implicit class”
- Lets you write `a.then(f)`
- Same as f(a)

```
implicit class ThenOps[A](a: A) {
  def then[B](f: A ⇒ B): B = f(a)
}
```

Goals

Subtype Polymorphism

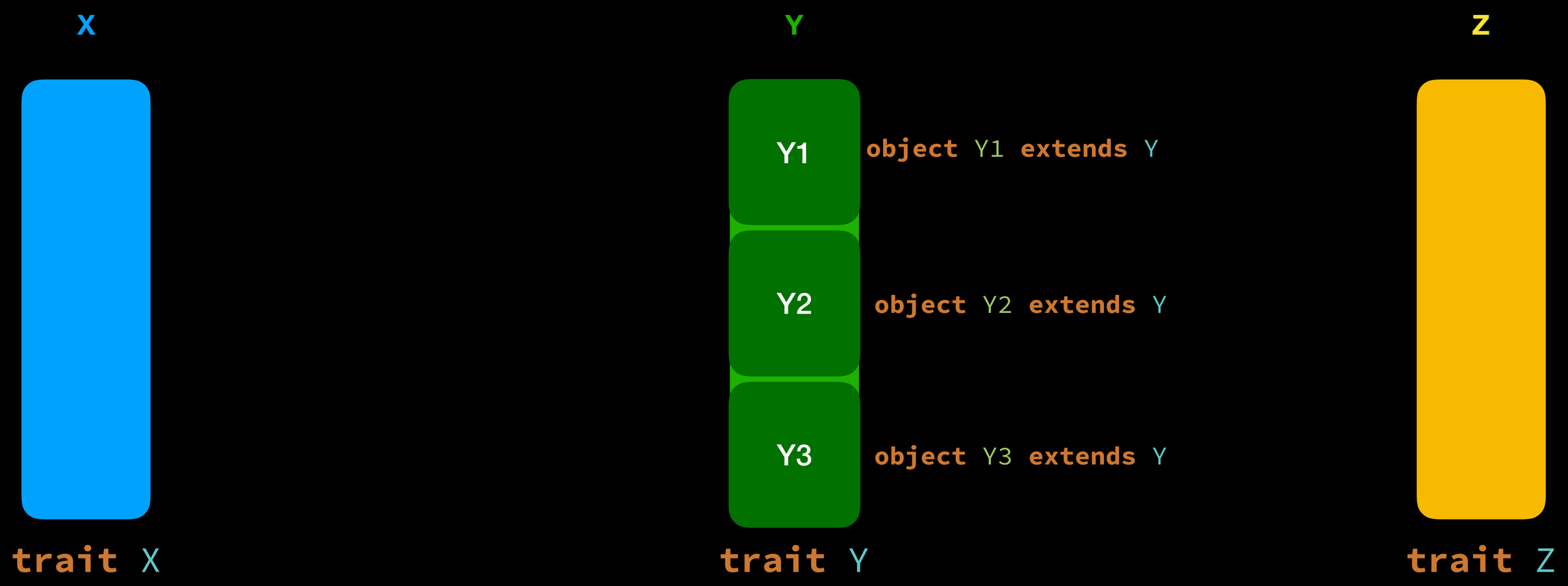
Instance Polymorphism

XP with a Compiler

Types

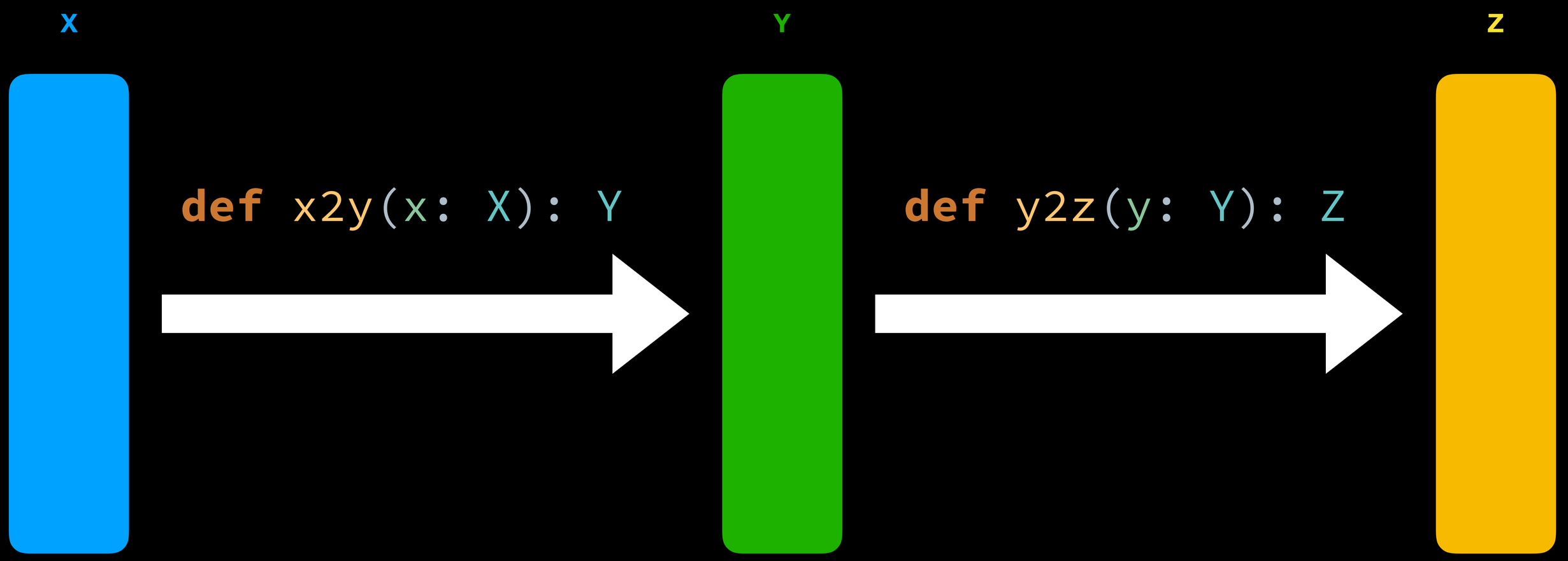
Values : Types :: Elements : Sets

Values : Types :: Regions : Spaces



Types Model the Flow of Computation

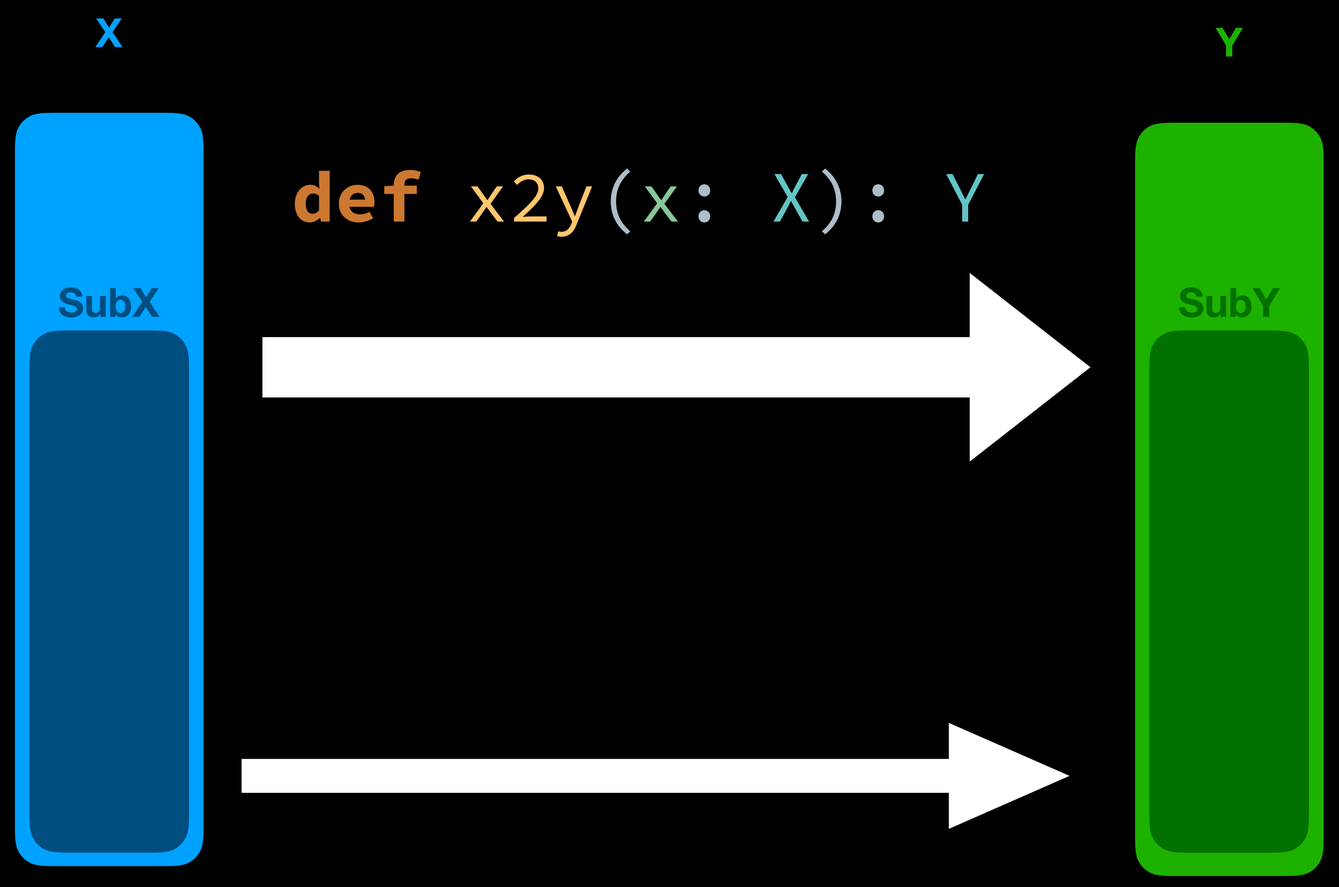
```
val x: X
val xThenYThenZ: Z = x.then(x2y).then(y2z)
```



Subtypes provide constraints

```
trait X
trait SubX extends X
```

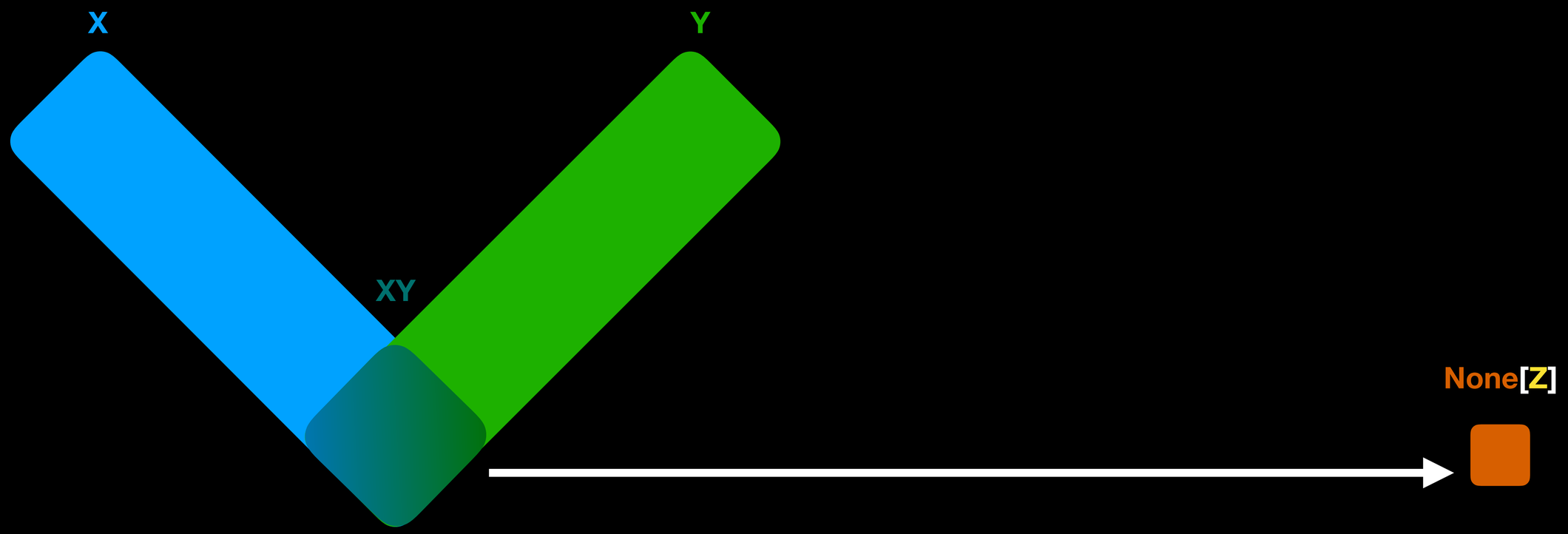
```
trait Y
trait SubY extends Y
```




```
def sub_x2y[A <: X, B <: Y](a :A ): B
def sub_x2y[SX<: X, SY<: Y](sx:SX): SY
```

Type Param names are just names

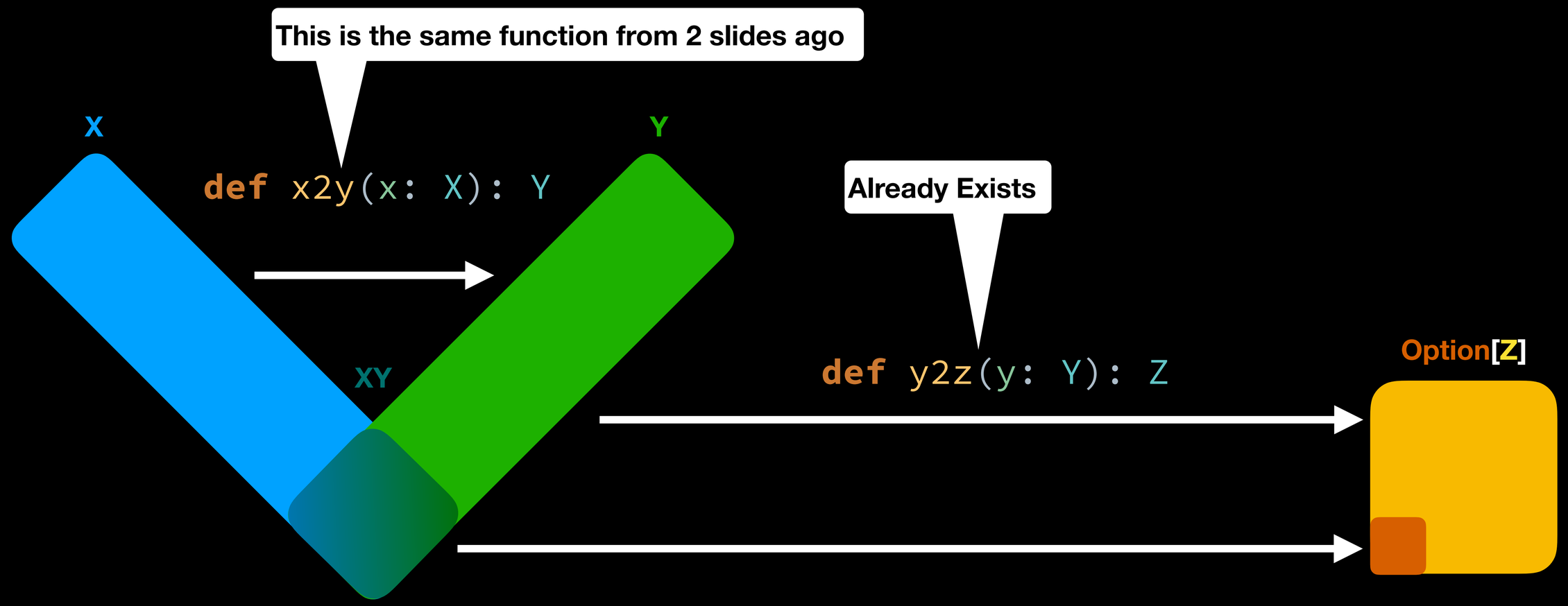
Functions and Types Honor Constraints



Type Params with bounds vs concrete subtypes

```
def xy2z[AB <: X with Y](ab: AB): Option[Z]
def xy2z(ab: X with Y)          : Option[Z]
```

Subtypes are Expressive




```
def xy2z[AB <: X with Y](ab: AB): Option[Z]
```

Subtype vs Instanced?

7:25

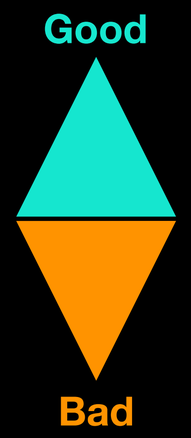
Goals

Subtype Polymorphism

Instance Polymorphism

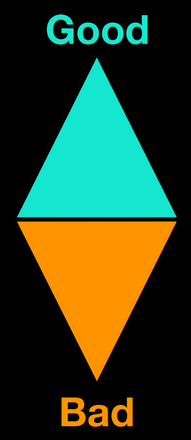
XP with a Compiler

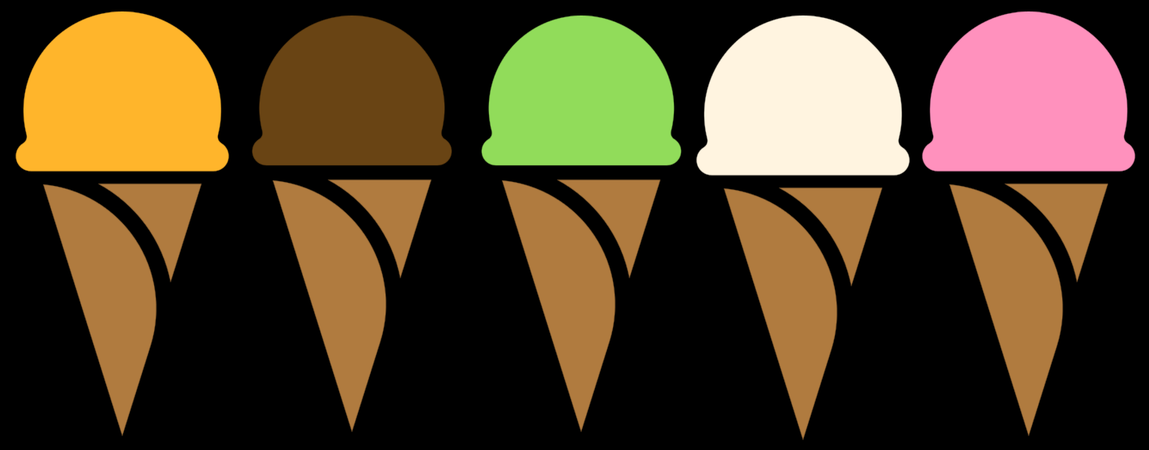
Being Opinionated




```scala
trait Opinion
object Good extends Opinion
object Bad  extends Opinion
```

How Opinionated?

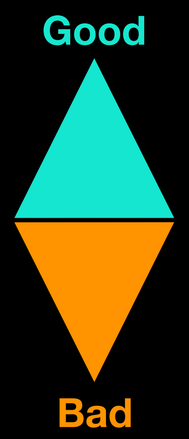




```
trait  IceCream
object Mango      extends IceCream
object Chocolate  extends IceCream
object Pistachio  extends IceCream
object Vanilla    extends IceCream
object Strawberry extends IceCream
```

How to model function that goes from IceCream to Opinion?

With a goal of decoupling.



ADT

```
trait Opinion
object Good extends Opinion
object Bad  extends Opinion
```

Comes from Simulacrum
- **Advisable not to hand write TypeClasses**

Definition

```
@typeclass trait Opinionated[A]{
  def opinion(a: A) : Opinion
}
```

Usage

```
def tellYourFriends[A](a: A)(
    implicit evidenceAIsOpinionated: Opinionated[A])
    : Future[Unit] =
  Future(println(
    s”I feel ${evidenceAIsOpinionated.opinion(a)} about $a"))
```

Syntax for Sanity

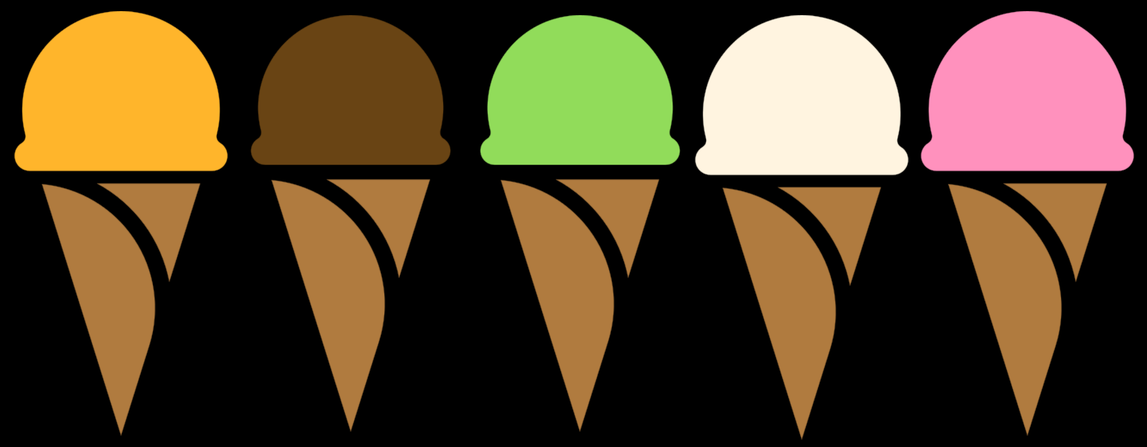
Expanded

```
def tellYourFriends[A](a: A)(
    implicit evidenceAIsOpinionated: Opinionated[A])
    : Future[Unit] =
  Future(println(
    s”I feel ${evidenceAIsOpinionated.opinion(a)} about $a"))
```

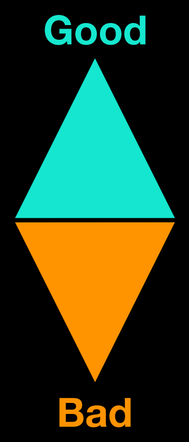
Syntactic Sugar

```
import Opinionated.ops._
def tellYourFriends[A: Opinionated](a: A)
    : Future[Unit] =
  Future(println(s"I feel ${a.opinion} about $a”))
```

Opinionated about IceCream



```scala
@typeclass trait Opinionated[A]{
  def opinion(a: A) : Opinion
}
```




```scala
implicit val iceCreamOpinion: Opinionated[IceCream] = {
  case Mango      ⇒ Good
  case Chocolate  ⇒ Bad
  case Pistachio  ⇒ Good
  case Vanilla    ⇒ Good
  case Strawberry ⇒ Bad
}
```

Conveniently Opinionated

Typeclass Instance

```
implicit val iceCreamOpinion: Opinionated[IceCream] = {
  case Mango      ⇒ Good
  case Chocolate  ⇒ Bad
  case Pistachio  ⇒ Good
  case Vanilla    ⇒ Good
  case Strawberry ⇒ Bad
}
```

Function Usage Site

```
tellYourFriends(Pistachio: IceCream)
```

Function with Typeclass Constraint

```
import Opinionated.ops._
def tellYourFriends[A: Opinionated](a: A)
    : Future[Unit] =
  Future(println(s"I feel ${a.opinion} about $a”))
```

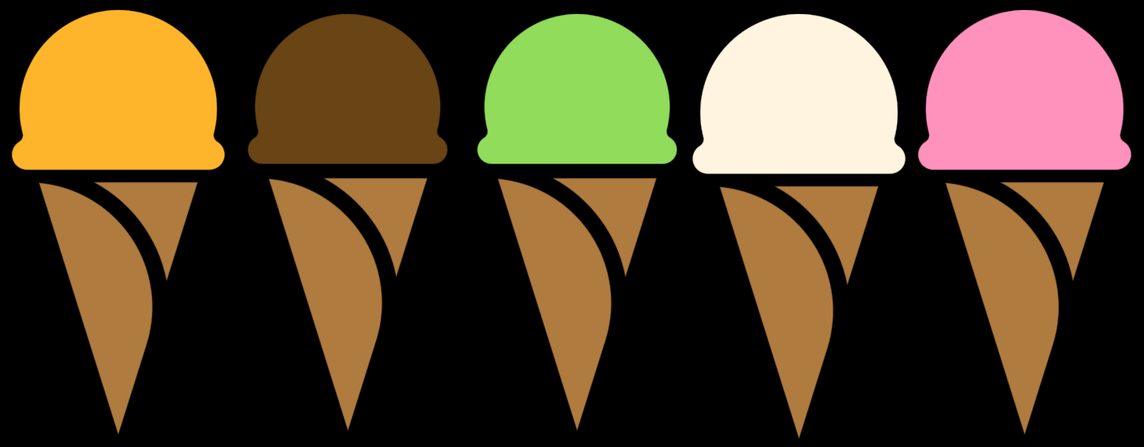
Ecosystems are about Sharing

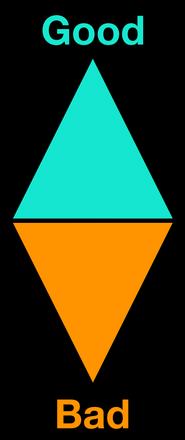
Real *cats* code!

```
@typeclass trait Foldable[F[_]] {
    /**
     * Fold implemented by mapping `A` values into `B` and then
     * combining them using the given `Monoid[B]` instance.
     */
    def foldMap[A, B](fa: F[A])(f: A ⇒ B)(implicit B: Monoid[B]): B =
      foldLeft(fa, B.empty)((b, a) ⇒ B.combine(b, f(a)))
}
```

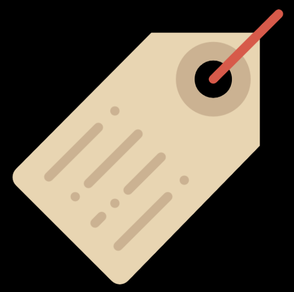
Libraries should be both general
and let your codebase stay flexible

BYO Typeclass Instances

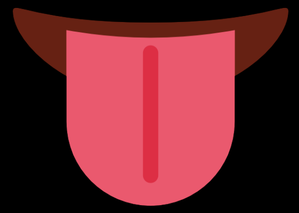




Recipie

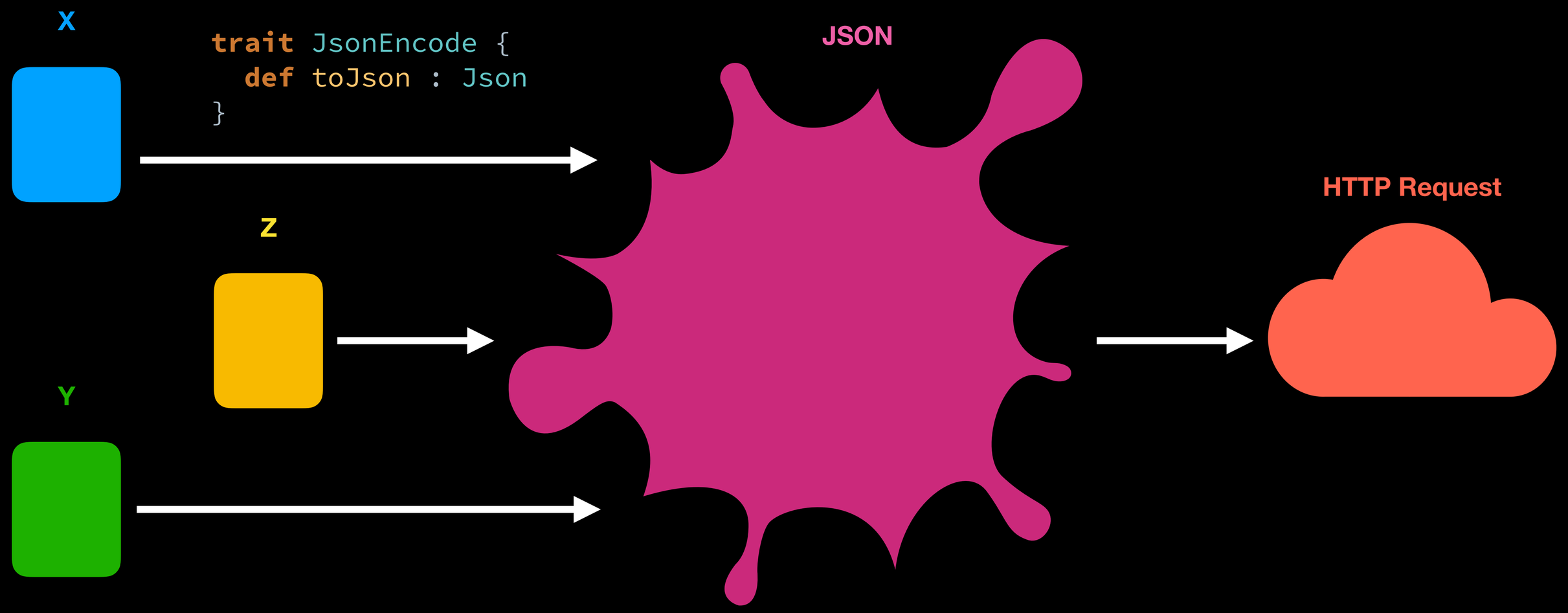


Nutrition Facts



Edibility

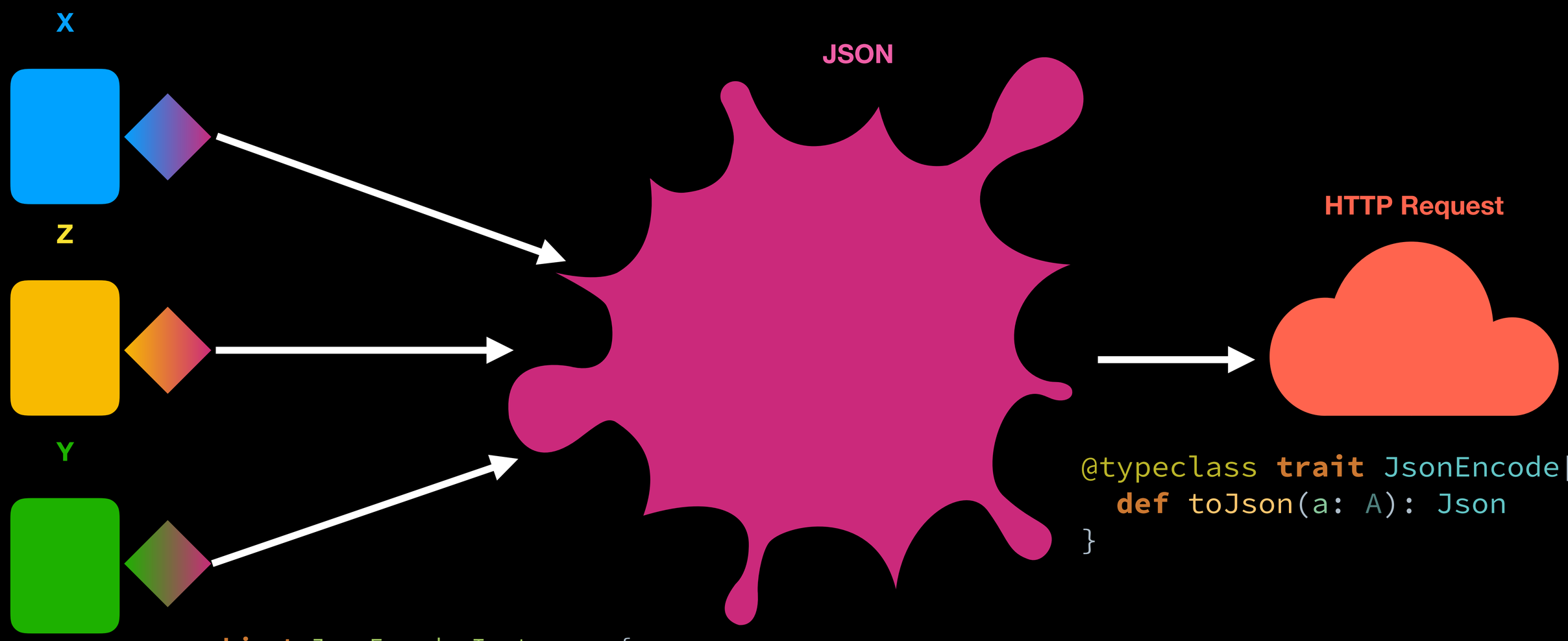
Implementing an interface




```
trait JsonEncode {
  def toJson : Json
}
```

```
sealed trait Json
case class   JsObject(kvps: (JsString, Json)*) extends Json
case class   JsString(s: String)               extends Json
case class   JsNumber[N: Numeric](n: N)        extends Json
case class   JsBool(n: Boolean)                extends Json
```

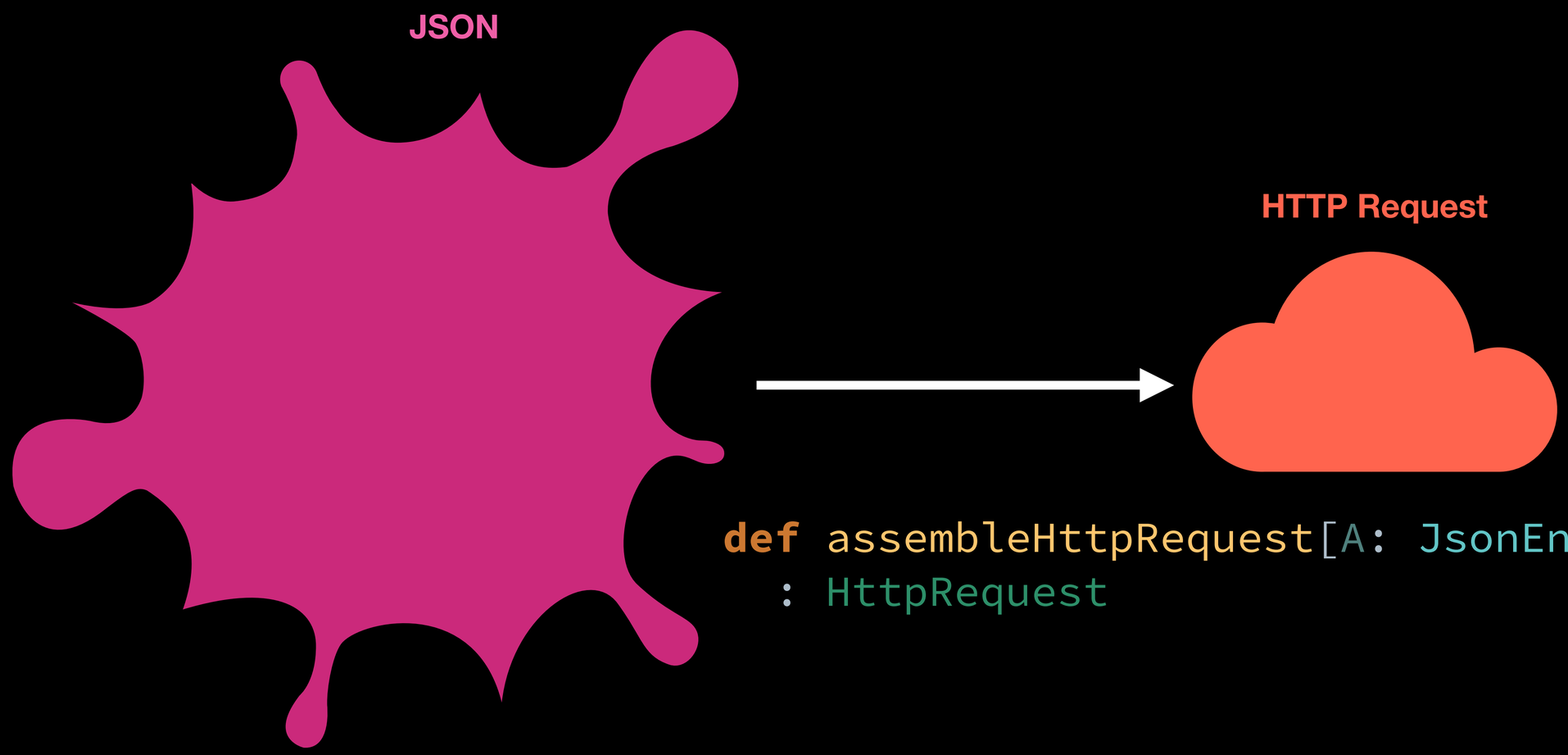
Prove it! Don’t inherit




```scala
@typeclass trait JsonEncode[A] {
  def toJson(a: A): Json
}
```

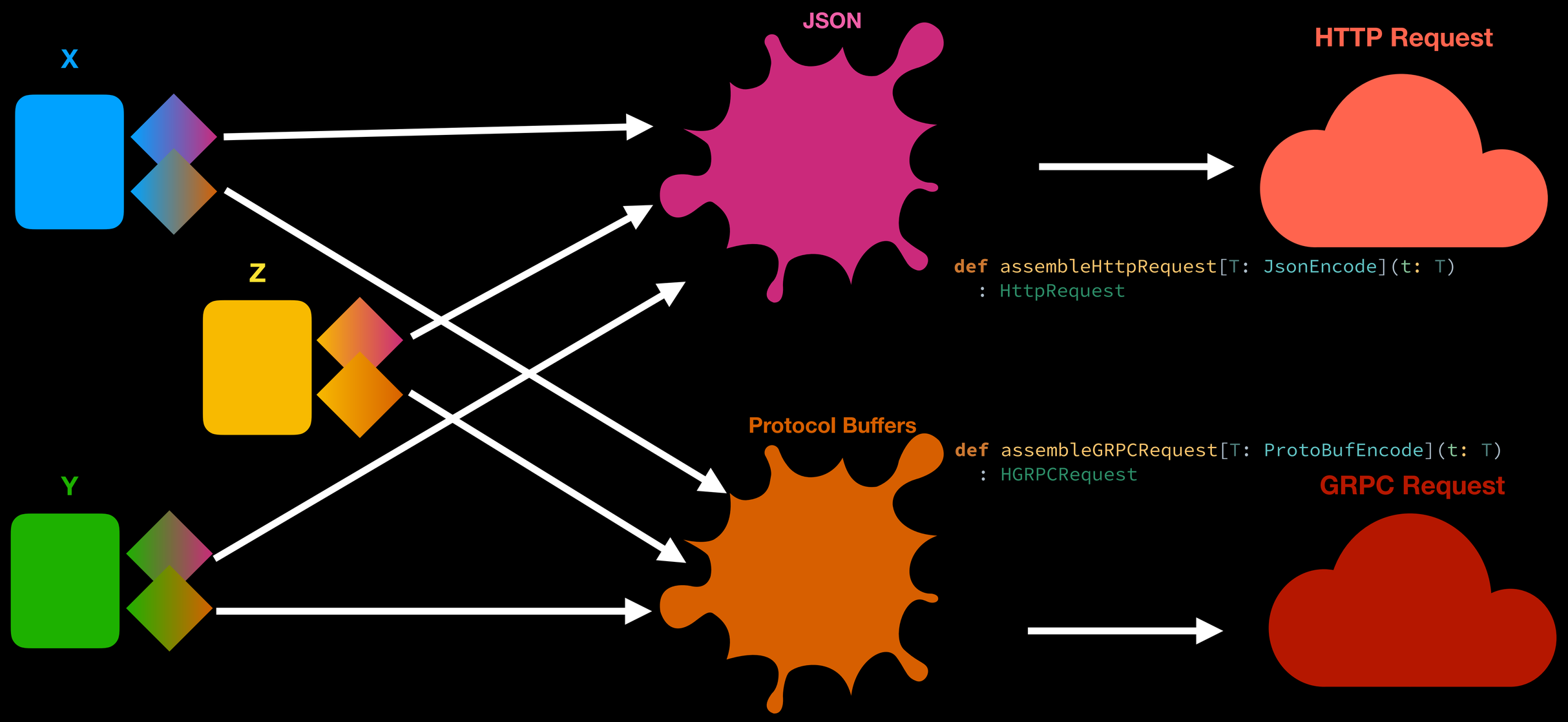
```scala
object JsonEncoderInstances {
  implicit val xJsonEncode: JsonEncode[X] = (_: X) ⇒ JsObject(JsString("type") → JsString("X"))
  implicit val yJsonEncode: JsonEncode[Y] = (_: Y) ⇒ JsObject(JsString("type") → JsString("Y"))
  implicit val zJsonEncode: JsonEncode[Z] = (_: Z) ⇒ JsObject(JsString("type") → JsString(“Z"))
}
```

Using Type Instances




```
def assembleHttpRequest[A: JsonEncode](a: A)
  : HttpRequest
```

Type instance constraints provide flexibility



Goals

Subtype Polymorphism

Instance Polymorphism

XP with a Compiler

When do I use them?

How should I organize my library?:

- **Organized by behaviors?**
  - **Use Type Instances**

- **Organized by domain-object?**
  - **Use Subtypal Inheritance**

Subtype vs Instanced?

7:40

THANK YOU