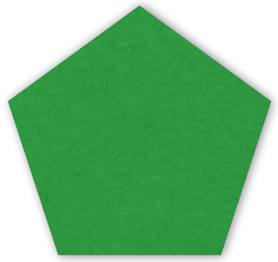
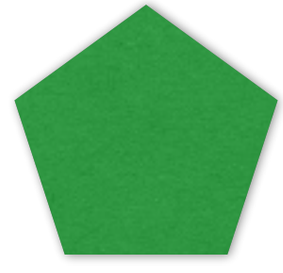
Shapes 

You are at Scala Syd
This is June in 2014
My name is Maxwell Swadling

Data types

- All data types have a "shape" 
- We are going to solve the farm yard problem
- Computer science problem
- We have a bag of animals on our farm
- We want to encode them in JSON

Dog 

```
case class Dog(name: String)
```

- Constructors:
  - List("Dog")
- Fields:
  - List(("name", "String"))

Animal 🐴

🐴

```
sealed trait Animal
```

🐕

```
case class Dog(name: String)
```

🐦

```
case class Parrot(
    name: String
  , age: Int)
```

- Constructors:
  - List("Dog", "Parrot")
- Fields:
  - Dog ->
    List(("name", "String"))
  - Parrot ->
    List(("name", "String")
        ,("age", "Int"))

What kind of shapes exist

```
sealed trait Shape

case class Void() extends Shape

case class Sum(name: String, left: Shape, right: Shape)
extends Shape

case class Product(left: Shape, right: Shape) extends Shape

case class UnitC() extends Shape
case class Value[A](recordName: String, a: A) extends Shape
```

A Dog's Shape

```
sealed trait Shape

case class Void() extends Shape

case class Sum(name: String, left:
Shape, right: Shape) extends Shape

case class Product(left: Shape,
right: Shape) extends Shape

case class UnitC() extends Shape
case class Value[A](recordName:
String, a: A) extends Shape
```

```
case class Dog(name: String)

def from(d: Dog): Shape =
  Sum(
    Product(Value("name", dog.name)
            ,Void())
  , Void()
  )
```

What's Shibe's Shape

```
case class Dog(name: String)

def from(d: Dog): Shape =
  Sum(
    Product(Value("name", dog.name)
            ,Void())
  , Void()
  )

from(Dog("Kabosu"))
res1: Shape =
 Sum(Dog,Product(Value(name,Kabosu),Void()),Void())
```

Functions over Shapes

```
def toJSON(x: Shape): String = x match {
  case Void() => "{}"
  case Sum(n, l, r) => s"[\"$n\", ${toJSON(l)}, ${toJSON(r)}]"
  case Product(l, r) => s"[${toJSON(l)}, ${toJSON(r)}]"
  case UnitC() => "{}"
  case Value(n, x) => "\"" ++ x.toString ++ "\""
}
```

```
toJSON(from(Dog("Kabosu")))
res13: String = ["Dog", ["Kabosu", {}], {}]
```

Labels	✅
Total Decode Function	⛔

shapeless

Coproduct

```
sealed trait Shape

case class Void() extends Shape

case class Sum(name: String,
left: Shape, right: Shape)
extends Shape
```

```
sealed trait Coproduct

sealed trait :+:[+H,
                 +T <: Coproduct]
  extends Coproduct
sealed trait CNil
  extends Coproduct
```

```
sealed trait Animal


case class Dog(name: String)


case class Parrot(
    name: String
  , age: Int)
```

```
type AnimalShape =
  Dog :+: Parrot :+: CNil
```

https://github.com/milessabin/shapeless

Product (HList)

```
case class Product(left: Shape,
right: Shape) extends Shape

case class UnitC() extends Shape
case class Value[A](recordName:
String, a: A) extends Shape
```

```
sealed trait HList
final case class ::[+H, +T <: HList(
  head : H, tail : T) extends HList {
  override def toString =
   head+" :: "+tail.toString
}
sealed trait HNil extends HList {
  def ::[H](h : H) = shapeless.::(h, this)
  override def toString = "HNil"
}
```

```
sealed trait Animal


case class Dog(name: String)


case class Parrot(
    name: String
  , age: Int)
```

```
type DogCons = String :: HNil
type ParrotCons =
  String :+: Int :+: HNil
```

https://github.com/milessabin/shapeless

Generics

- Derive the to/from via a macro, no boilerplate required.

- Instead of a Shape, we get a Representation parametrised by the types of on the Coproduct and Products.

Labels	⛔
Total Decode Function	✅

Aside: Singletons

- `sealed trait TheStringFoo`
- `case class TheStringFooValue extends TheStringFoo`
- `func stringFoo: TheStringFoo = TheStringFooValue`
- A macro (in shapeless)

```
sealed trait Animal


case class Dog(name: String)


case class Parrot(
    name: String
  , age: Int)
```

```
type AnimalShape =
  Dog :+: Parrot :+: CNil
```

```
type AnimalShape =
      ("Dog", Dog)
  :+: ("Parrot", Parrot)
  :+: CNil
```

```
type DogCons = String :: HNil
type ParrotCons =
  String :: Int :: HNil
```

```
type DogCons =
  ("name", String) :: HNil

type ParrotCons =
      ("name", String)
  :: ("age", Int)
  :: HNil
```

Labelled Generic

```
implicit def EncodeJsonTypeClass: LabelledTypeClass[EncodeJson]
  = new LabelledTypeClass[EncodeJson] {
    def emptyCoproduct =
      EncodeJson(_ => jEmptyObject)

    def coproduct[L, R <: Coproduct](name: String, CL: => EncodeJson[L]
                                     ,CR: => EncodeJson[R]): EncodeJson[L :+: R] =
      EncodeJson(a => a match {
        case Inl(x) => Json((name -> CL.encode(x)))
        case Inr(t) => CR.encode(t)
      })

  def emptyProduct =
    EncodeJson(_ => jEmptyObject)

  def product[A, T <: HList](name: String, A: EncodeJson[A], T: EncodeJson[T]) =
    EncodeJson(a => (name -> A.encode(a.head)) ->: T.encode(a.tail))

  def project[F, G](instance: => EncodeJson[G], to : F => G, from : G => F) =
    instance.contramap(to)
}
```

Labels	✅
Total Decode Function	✅

https://github.com/argonaut-io/argonaut/pull/108

Labelled Generic Encoder 

```
case class Dog(name: String)

implicitly[EncodeJson[Dog]]
```

```
Dog("Kabosu").asJson

{
 "name" : "Kabosu"
}
```

Labelled Generic Encoder 🐴

```
sealed trait Animal
case class Dog(name: String)
case class Parrot(
    name: String
  , age: Int)

implicitly[EncodeJson[Animal]]

Dog("Kabosu").asJson

{
  "Dog" :
    {
      "name" : "Kabosu"
    }
}
```


such wow
much finish