Introduction to Type Driven Development in Scala

Marcus A. Henry, Jr.
@dreadedsoftware

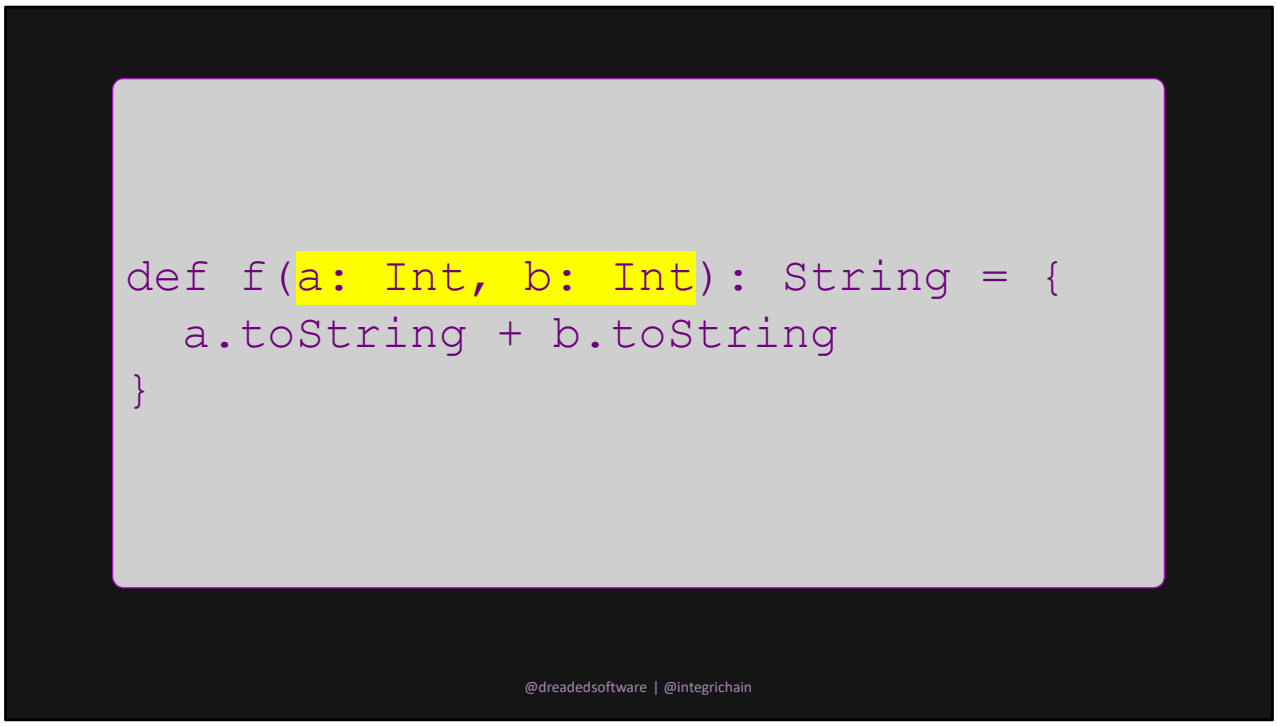
Software Engineer @integrichain

```
def f(a: Int, b: Int): String = {
  a.toString + b.toString
}
```


This is a function

```
def f(a: Int, b: Int): String = {
  a.toString + b.toString
}
```


keyword


def f(a: Int, b: Int): String = {
a.toString + b.toString
}
@dreadedsoftware | @integrichain

input

```
def f(a: Int, b: Int): String = {
  a.toString + b.toString
}
```

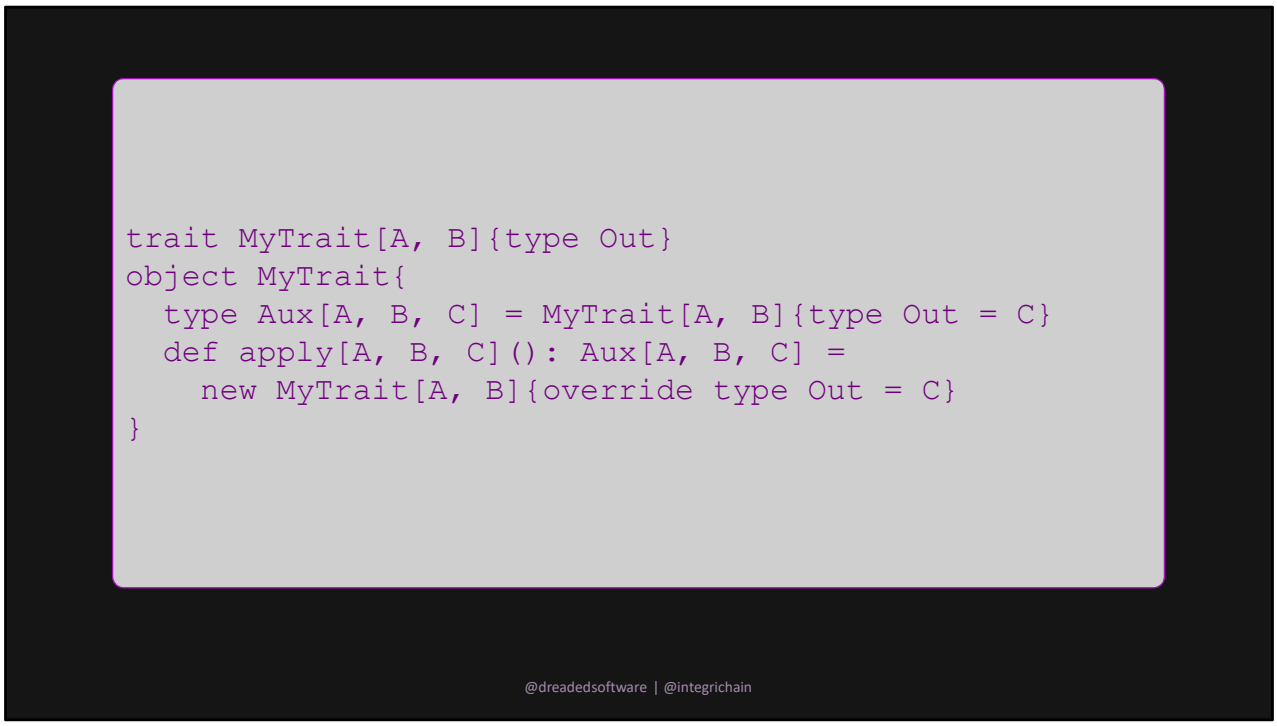

output

```
def f(a: Int, b: Int): String = {
  a.toString + b.toString
}
```

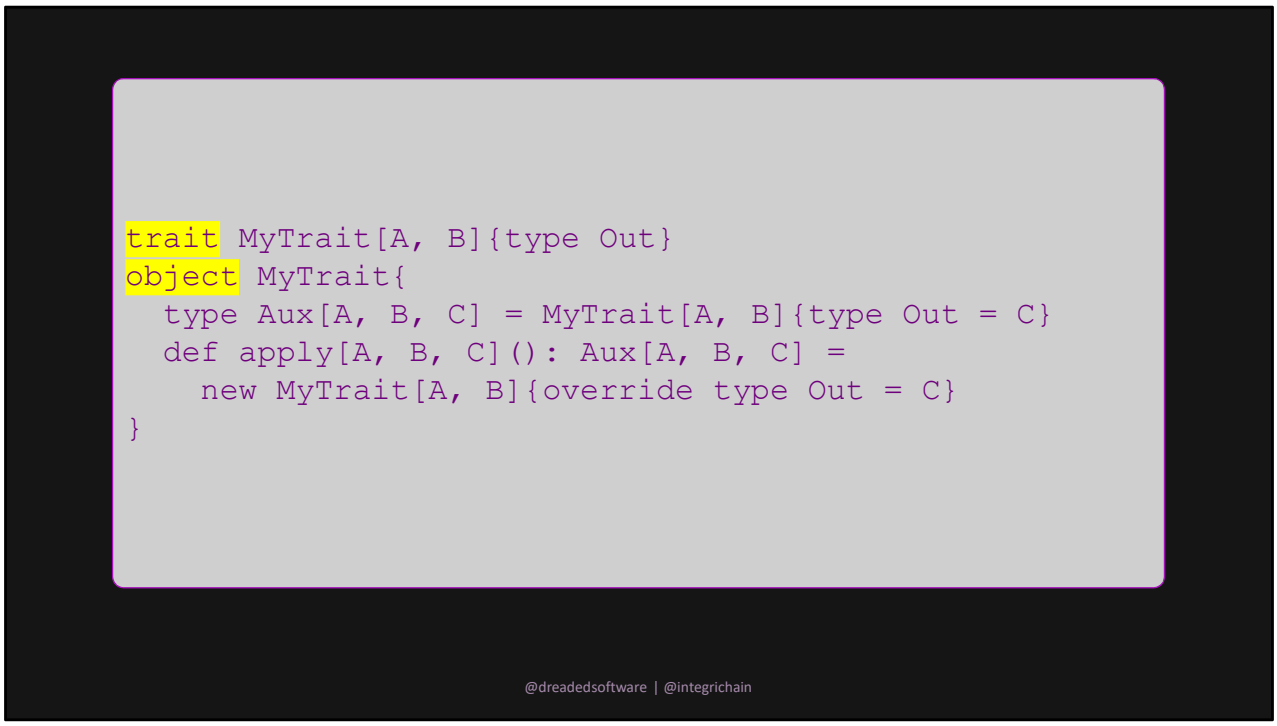

body

```
trait MyTrait[A, B]{type Out}
object MyTrait{
  def apply[A, B, C](): MyTrait[A, B]{type Out = C} =
    new MyTrait[A, B]{override type Out = C}
}
```

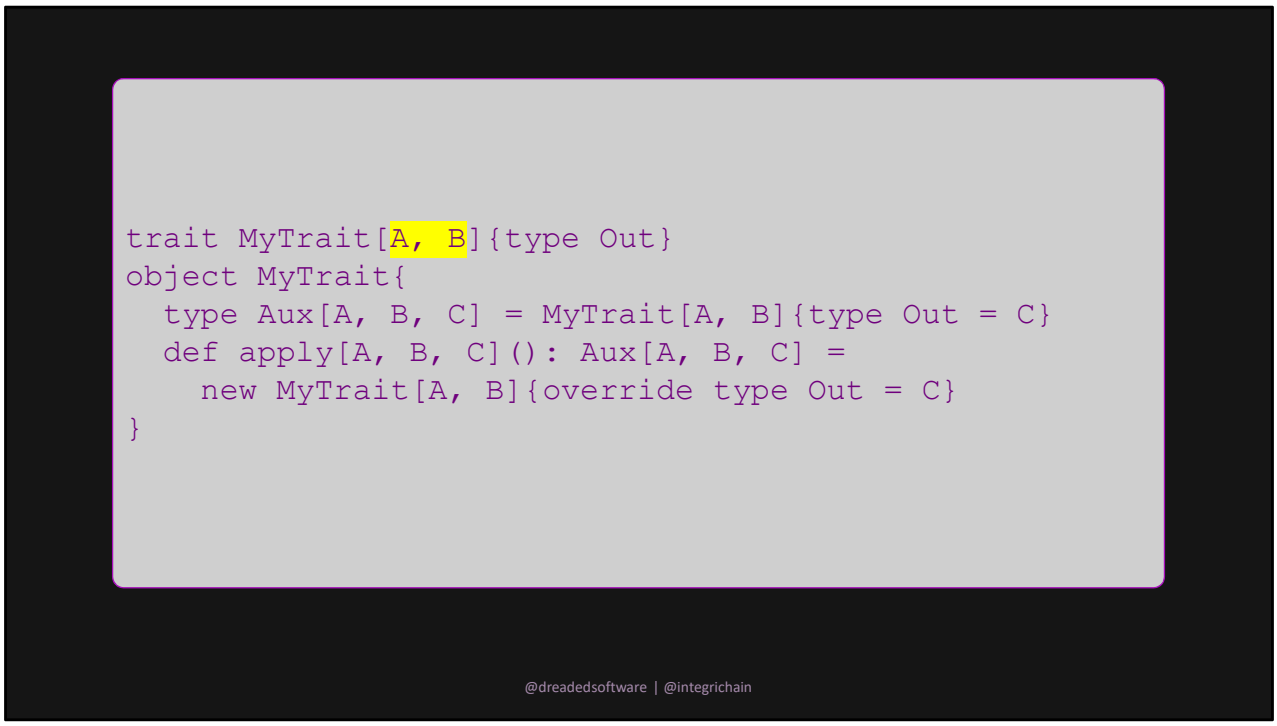

This is a type level function



To simplify we use the Aux pattern


trait MyTrait[A, B]{type Out}
object MyTrait{
type Aux[A, B, C] = MyTrait[A, B]{type Out = C}
def apply[A, B, C](): Aux[A, B, C] =
new MyTrait[A, B]{override type Out = C}
}
@dreadedsoftware | @integrichain

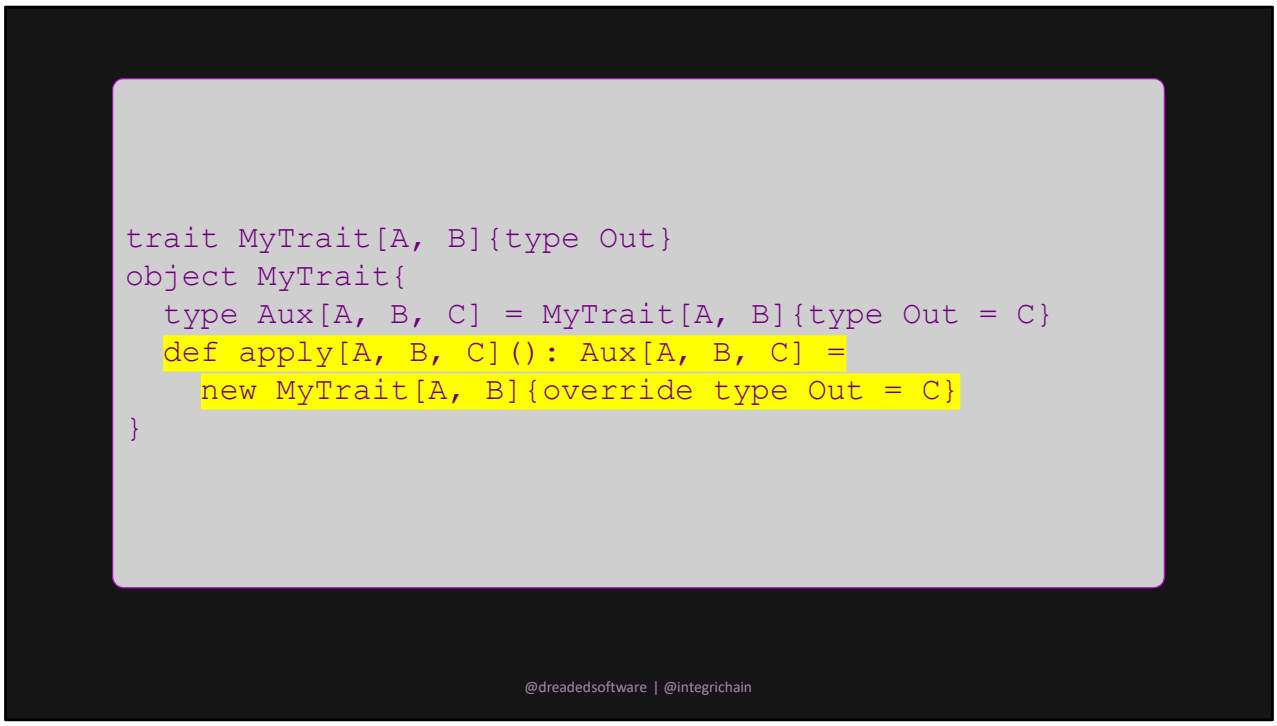
keyword


trait MyTrait[A, B]{type Out}
object MyTrait{
type Aux[A, B, C] = MyTrait[A, B]{type Out = C}
def apply[A, B, C](): Aux[A, B, C] =
new MyTrait[A, B]{override type Out = C}
}
@dreadedsoftware | @integrichain

input

```
trait MyTrait[A, B]{type Out}
object MyTrait{
  type Aux[A, B, C] = MyTrait[A, B]{type Out = C}
  def apply[A, B, C](): Aux[A, B, C] =
    new MyTrait[A, B]{override type Out = C}
}
```

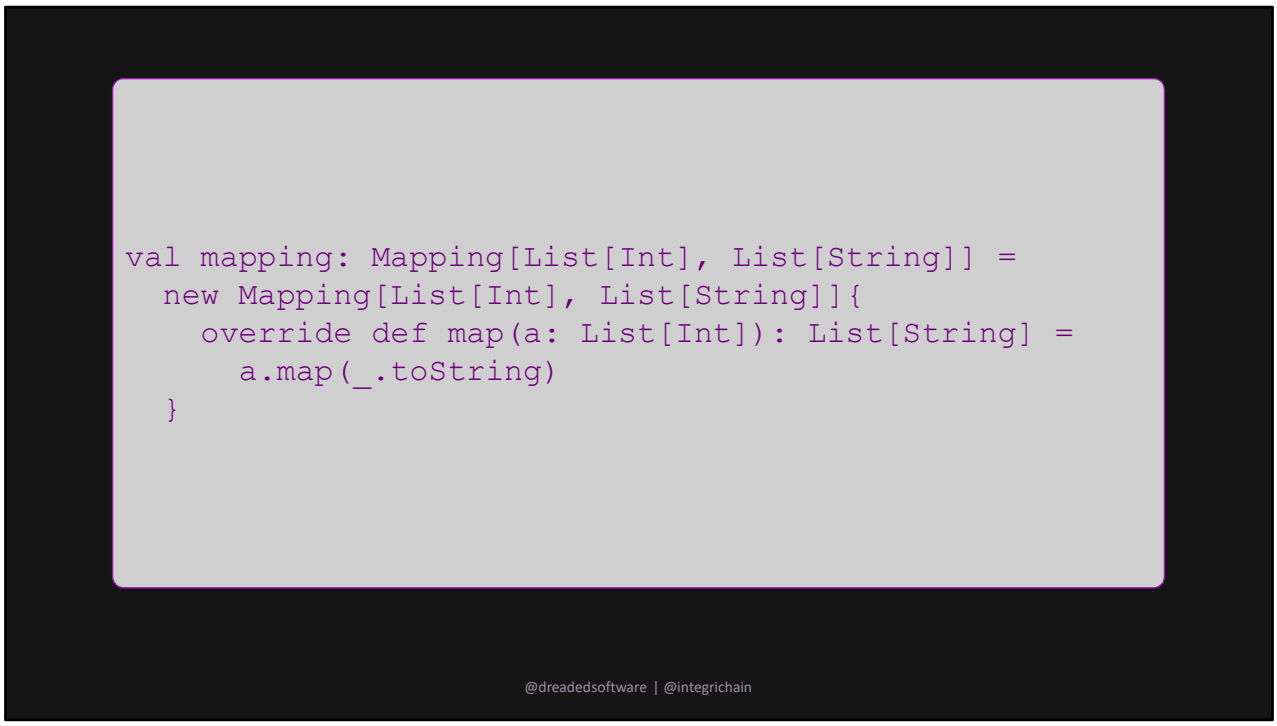

output


trait MyTrait[A, B]{type Out}
object MyTrait{
type Aux[A, B, C] = MyTrait[A, B]{type Out = C}
def apply[A, B, C](): Aux[A, B, C] =
new MyTrait[A, B]{override type Out = C}
}
@dreadedsoftware | @integrichain

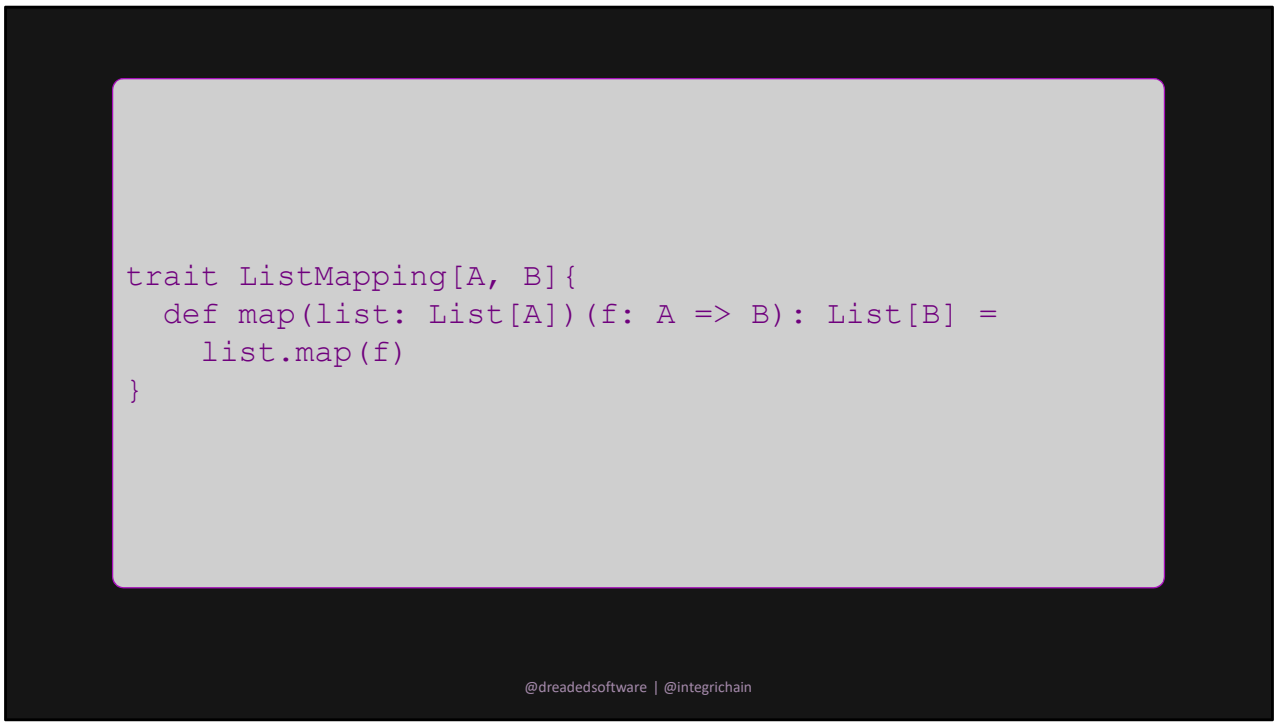
body

```
trait Mapping[A, B]{
  def map(a: A): B
}
```


Take this mapper type thing



And an instance for List[Int] => List[String]
Super restrictive, new instance for each type of List
We want to map any List of any type

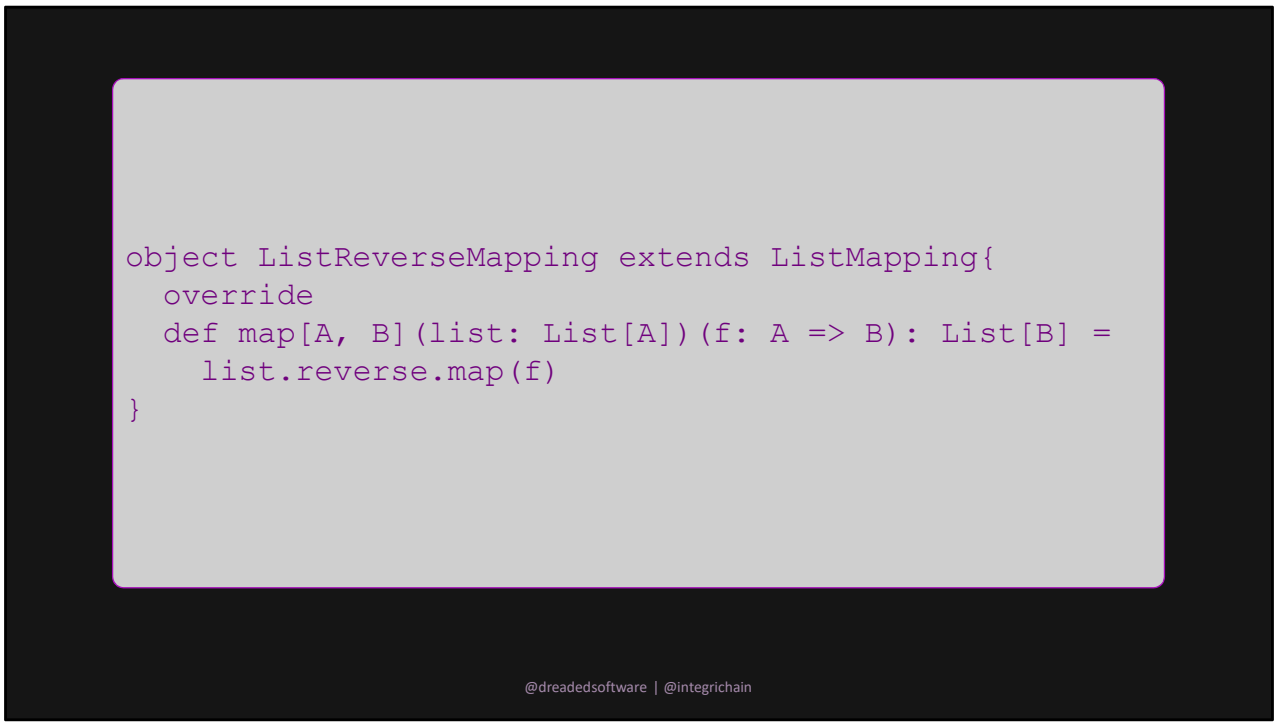


Given a List[A] and a function A => B we can get a List[B]

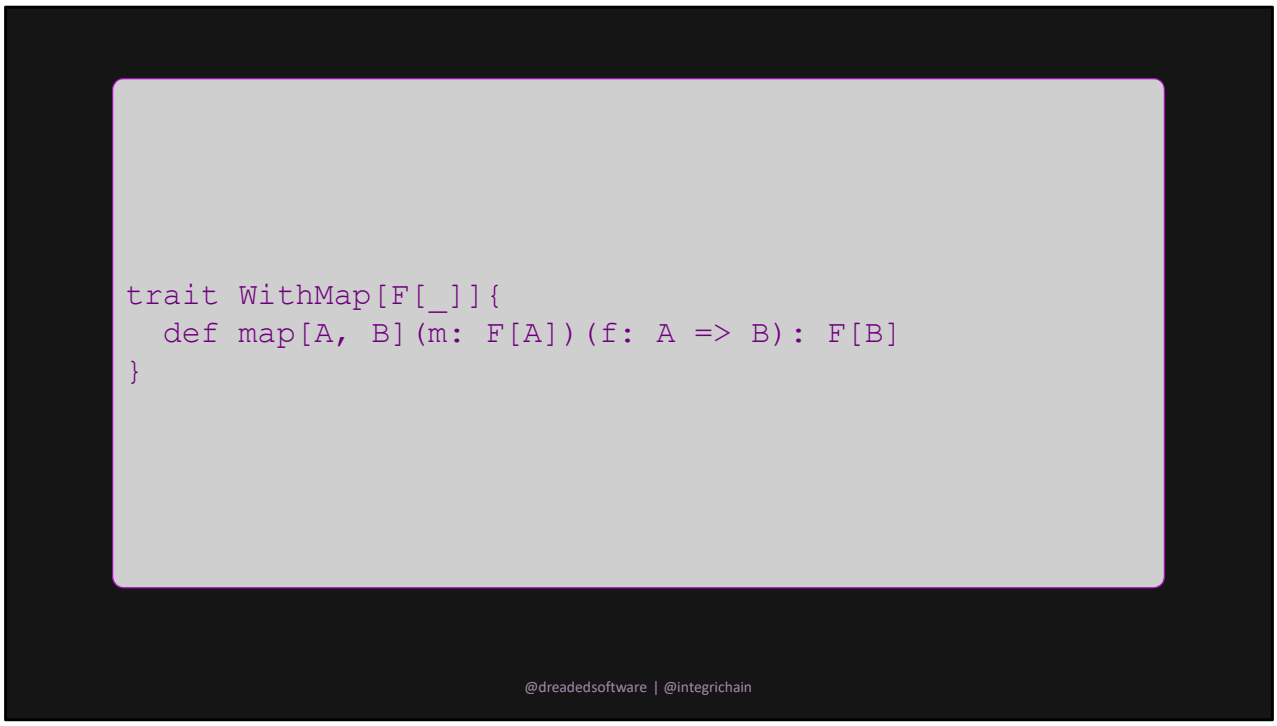
```
trait ListMapping{
  def map[A, B](list: List[A])(f: A => B): List[B] =
    list.map(f)
}
```

@dreadedsoftware | @integrichain

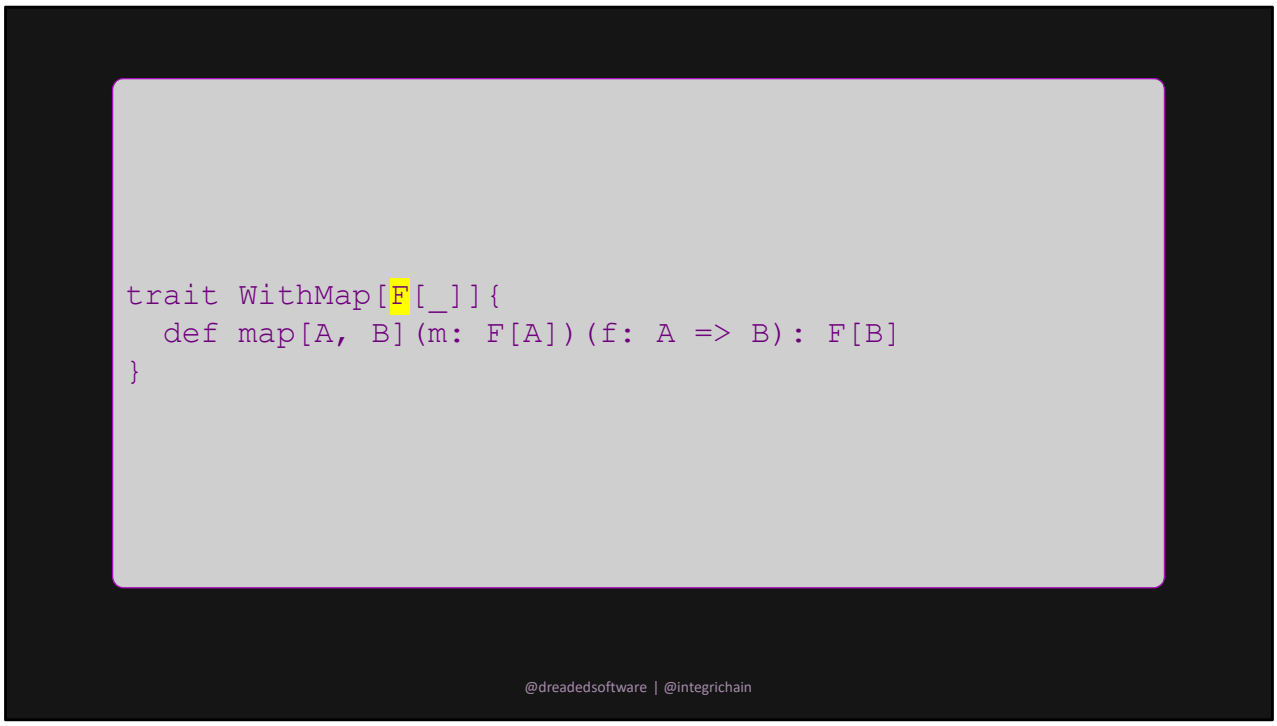
By dropping the type parameters to the function, we get more freedom
But this is crazy! We could just call map on List, why use a typeclass at all



More than one way to map
This approach gives us the freedom to define new functionality for old structures



Why stop at List
Scala allows higher kinded types
abstracting over types which abstract over types (type constructors)



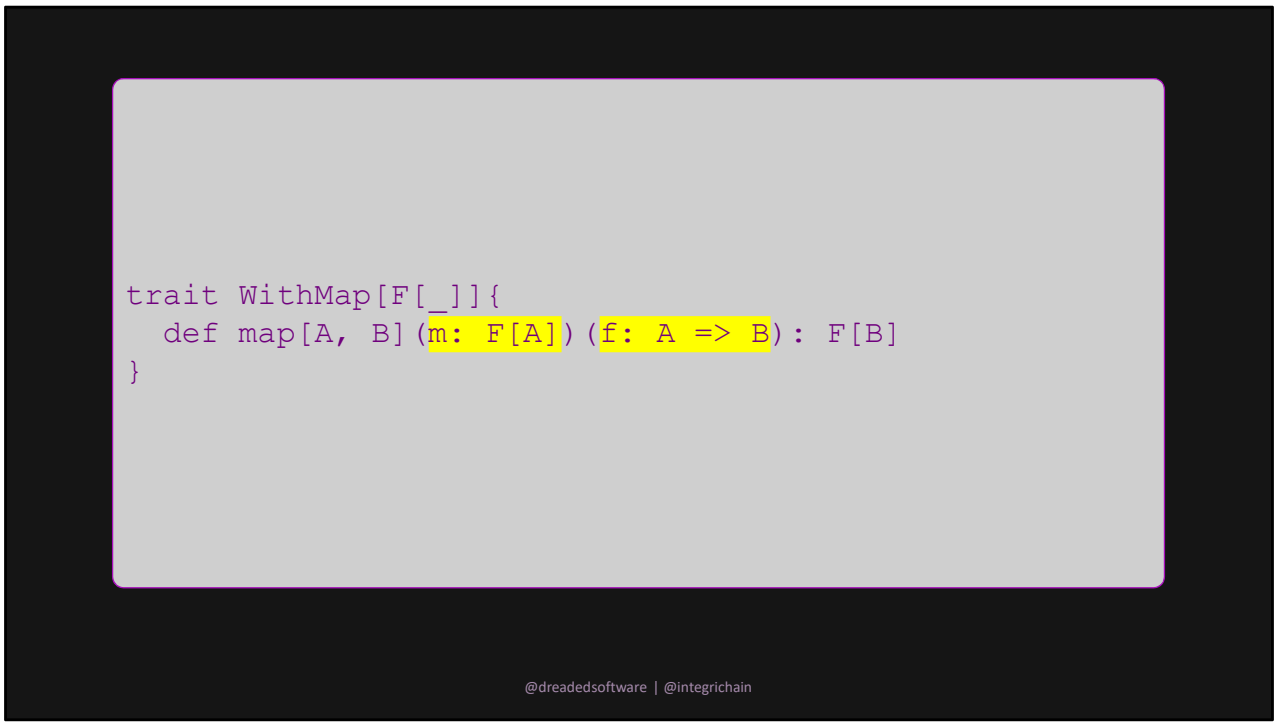
Given a type, F

```
trait WithMap[F[_]]{
  def map[A, B](m: F[A])(f: A => B): F[B]
}
```


Which takes another type, _

```
trait WithMap[F[_]]{
  def map[A, B](m: F[A])(f: A => B): F[B]
}
```


We can decide how to change the inner type from A to B



Given an F[A] and a function A => B we can get F[B]
Changes values (A, B) without changing context (F)

```
implicit val listWithMap = new WithMap[List]{
  override def map[A, B](m: List[A])(f: A => B): List[B] =
    m.map(f)
}
implicit val optionWithMap = new WithMap[Option]{
  override def map[A, B](m: Option[A])(f: A => B): Option[B] =
    m.map(f)
}
implicit val streamWithMap = new WithMap[Stream]{
  override def map[A, B](m: Stream[A])(f: A => B): Stream[B] =
    m.map(f)
}
val reverseListWithMap = new WithMap[List]{
  override def map[A, B](m: List[A])(f: A => B): List[B] =
    m.reverse.map(f)
}
```

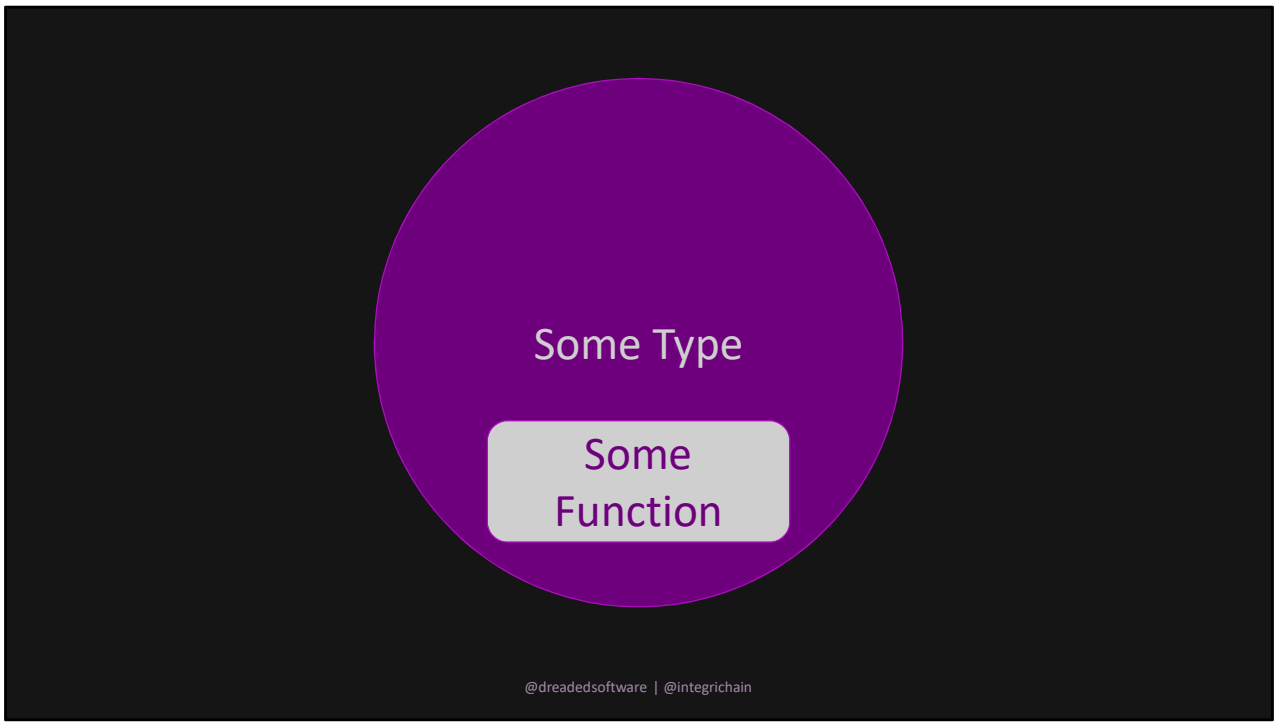

And we can do this with any Parameterized type of arity 1

```
def prettyString[
F[_]: WithMap, A](m: F[A])(f: A => String): String = {
  implicitly[WithMap[F]].map(m)(f).toString
}

val list = List(rnd, rnd, rnd, rnd, rnd)
val stream = Stream(rnd, rnd, rnd, rnd, rnd)
val option = Option(rnd)
val f1: Int => String = {i: Int =>
  "As String: " + i.toString
}

prettyString(list)(f1)
prettyString(option)(f1)
prettyString(stream)(f1)
```

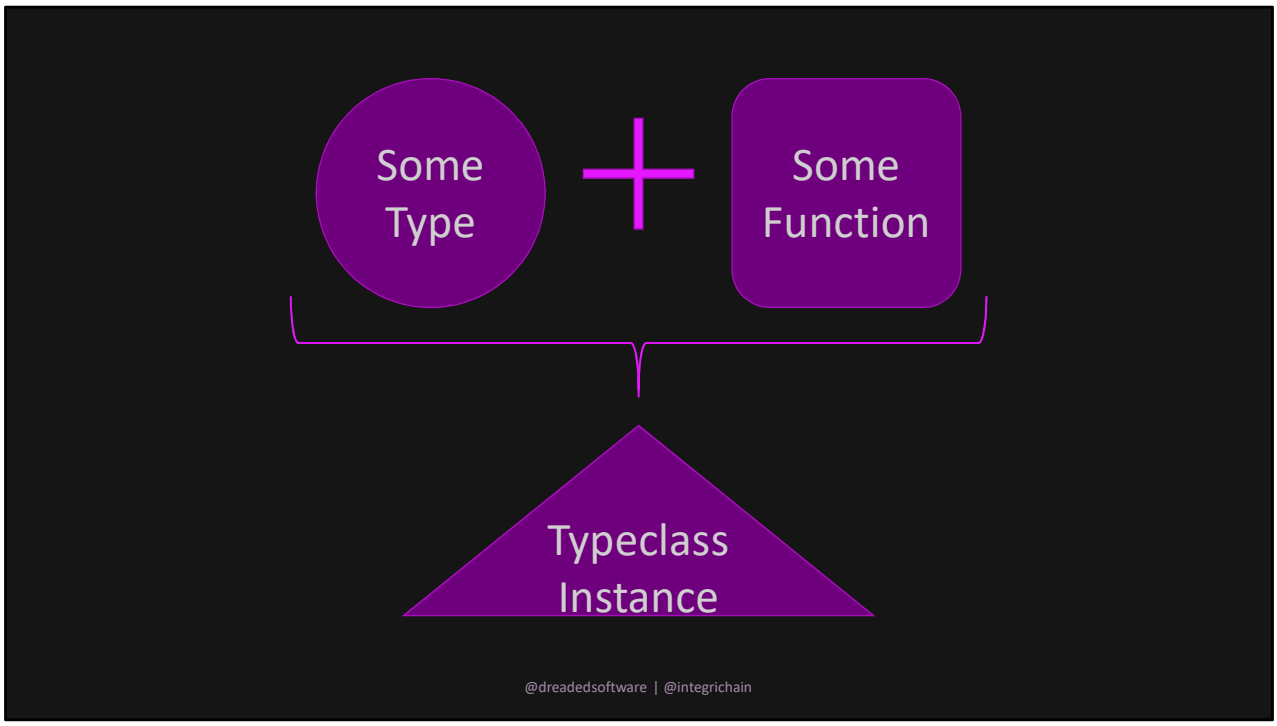

We can define super polymorphic functions, here is a pretty string printer

```
def processData[F[_]: WithMap, A, B, C, D](m1: F[A])(
  f1: A => B)(
  f2: B => C)(
  f3: C => D): F[D] = {
  val F = implicitly[WithMap[F]]
  val m2 = F.map(m1)(f1)
  val m3 = F.map(m2)(f2)
  F.map(m3)(f3)
}

val f2: String => Array[Byte] = _.getBytes
val f3: Array[Byte] => Long = _.map(_.toLong).sum

processData(list)(f1)(f2)(f3)
processData(option)(f1)(f2)(f3)
processData(stream)(f1)(f2)(f3)
```


And a data processor



Too restrictive



Much less restrictive; more composable.

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/hlists.scala

```
sealed trait HList extends Product with Serializable

final case class ::[+H, +T <: HList](head : H, tail : T) extends HList {
  override def toString = head match {
    case _: ::[_, _] => "("+head+") :: "+tail.toString
    case _ => head+" :: "+tail.toString
  }
}

sealed trait HNil extends HList {
  def ::[H](h : H) = shapeless.::(h, this)
  override def toString = "HNil"
}

case object HNil extends HNil
```


This is HList

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/hlists.scala

```
sealed trait HList extends Product with Serializable

final case class ::[+H, +T <: HList](head : H, tail : T) extends HList {
  override def toString = head match {
    case _: ::[_, _] => "("+head+") :: "+tail.toString
    case _ => head+" :: "+tail.toString
  }
}

sealed trait HNil extends HList {
  def ::[H](h : H) = shapeless.::(h, this)
  override def toString = "HNil"
}

case object HNil extends HNil
```


Much like a regular List, it has a head element and a tail element
The tail is recursive
This however, it at the type level rather than at the value level

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


A function on HList

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


keyword

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


input

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


output

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


Body; read `implicit def` as a type level `case` statement

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


The first case

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


Given a type constructor

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


Yield the empty HList

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


The second case

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


Given a head type, a tail which is an Hlist, a type constructor and an Out which is an HList

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


Given evidence that there is an Aux instance for out Tail, F and OutM

https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala#L62

```
trait Mapped[L <: HList, F[_]] extends Serializable {
  type Out <: HList
}
object Mapped {
  …
  type Aux[L <: HList, F[_], Out0 <: HList] =
    Mapped[L, F] { type Out = Out0 }

  implicit def hnilMapped[F[_]]: Aux[HNil, F, HNil] =
    new Mapped[HNil, F] { type Out = HNil }
  …
  implicit def hlistMapped1[
  H, T <: HList, F[_], OutM <: HList](implicit
  mt: Mapped.Aux[T, F, OutM]): Aux[H :: T, F, F[H] :: OutM] =
    new Mapped[H :: T, F] { type Out = F[H] :: OutM }
}
```


We can have an HList where our type constructor is applied to H and our Tail follows.

```
def zipper[A, B, C](
  a: List[A], b: List[B], c: List[C]
): List[(A, (B, C))] = a.zip(b.zip(c))
```


This zips three lists together
We know we can make this more general with a type class, let's start there.

```
trait Zip[F[_]]{
  def apply[A, B](a: F[A], b: F[B]): F[(A, B)]
}
```


Given two instances of some type constructor, we can produce an instance of that type constructor with a tuple as its parameter.

```
def zipper[F[_]: Zip, A, B, C](
  a: F[A], b: F[B], c: F[C]): F[(A, (B, C))] = {
  val F = implicitly[Zip[F]]
  F(a, F(b, c))
}
```


Now, we work with any type constructor. This is still super restrictive.
It only works with 3 values so maintenance is terrible if you want to use 2 or 4 or 70 values.
Let's try to employ our new found type level powers here

```
def zipper[F[_]: Zip, H, T](
  h: F[H], t: F[T]): F[(H, T)] = {
  val F = implicitly[Zip[F]]
  F(h, t)
}
```


The first step is simplifying the function to single instances of its constituent parts

```
implicit val ZipList = new Zip[List]{
  override def apply[A, B](
    a: List[A], b: List[B]): List[(A, B)] = a.zip(b)
}
val with2 = zipper(list1, list2)
val with3 = zipper(list1,
            zipper(list2, list3))
val with6 = zipper(list1,
            zipper(list2,
            zipper(list3,
            zipper(list4,
            zipper(list5, list6)))))
```


We can now call it recursively to produce the desired result.
Recall the shapeless code we read. Recursive type valued functions can be automated with implicits

```
implicit def zipper[F[_]: Zip, H, T](implicit
  h: F[H], t: F[T]): F[(H, T)] = {
  val F = implicitly[Zip[F]]
  F(h, t)
}
```


It is the same damn code with implicit placed in two locations

```
implicit def zipper[F[_]: Zip, H, T](implicit
  h: F[H], t: F[T]): F[(H, T)] = {
  val F = implicitly[Zip[F]]
  F(h, t)
}
```


Implicit the inputs so we don’t have to specify them by hand

```
implicit def zipper[F[_]: Zip, H, T](implicit
  h: F[H], t: F[T]): F[(H, T)] = {
  val F = implicitly[Zip[F]]
  F(h, t)
}
```


Implicit the calling of the function (it can implicitly call itself too)

```
type F[A] = List[A]
type Result =
  F[
    (Int, (Long, (String, (Double, (Float, Array[Byte])
  )))))]

implicit val list1: List[Int] = ???
implicit val list2: List[Long] = ???
implicit val list3: List[String] = ???
implicit val list4: List[Double] = ???
implicit val list5: List[Float] = ???
implicit val list6: List[Array[Byte]] = ???

implicitly[Result]
```


The way we tell the compiler what it needs is using implicits and types
We can use type aliases to make things simpler in the business logic
We use implicit vals to give the compiler the values it needs
We use the `implicitly` function to tell the compiler what it needs to execute
Also, note the order in which we define our implicits doesn't matter
the compiler assembled everything in the proper order for us
No more errors from swapping two arguments

```
trait ZipG[F[_], G[_, _]]{
  def apply[A, B](a: F[A], b: F[B]): F[G[A, B]]
}
implicit def zipper[F[_], G[_, _], H, T](implicit
  F: ZipG[F, G], h: F[H], t: F[T]): F[G[H, T]] = {
  F(h, t)
}
implicit val zipListTuple2 = new ZipG[List, Tuple2]{
  override def apply[A, B](
    a: List[A], b: List[B]): List[(A, B)] = a.zip(b)
}
```


No need to stop at tuples!
We can abstract this to any type which takes two type parameters

```
implicit val zipListTuple2 = new ZipG[List, Tuple2]{
  override def apply[A, B](
    a: List[A], b: List[B]): List[(A, B)] = a.zip(b)
}
implicit val zipListEither = new ZipG[List, Either]{
  override def apply[A, B](
    a: List[A], b: List[B]): List[Either[A, B]] =
    for{a <- a; b <- b}yield{
      if(a.toString.size < b.toString.size) Left(a)
      else Right(b)
    }
}
```


And some instances

```
type F[A] = List[A]
type Result =
  F[
    (Int, (Long, (String, (Double, (Float, Array[Byte])
  )))))]

implicit val list1: List[Int] = ???
implicit val list2: List[Long] = ???
implicit val list3: List[String] = ???
implicit val list4: List[Double] = ???
implicit val list5: List[Float] = ???
implicit val list6: List[Array[Byte]] = ???

implicitly[Result]
```


We can use it like before with Tuples

```
type F[A] = List[A]
type Result =
  F[
    Either[Int, Either[Long, Either[String,
    Either[Double, Either[Float, Array[Byte]]
  ]]]]

implicit val list1: List[Int] = ???
implicit val list2: List[Long] = ???
implicit val list3: List[String] = ???
implicit val list4: List[Double] = ???
implicit val list5: List[Float] = ???
implicit val list6: List[Array[Byte]] = ???

implicitly[Result]
```


Switching to Either is simple as well

```
def stringify1[A, B, C](
  fa: A => String, fb: B => String, fc: C => String,
  in: List[(A, (B, C))]): String = {
  in.map{case (a, (b, c)) =>
    fa(a) + ", " + fb(b) + ", " + fc(c)
  }.mkString("(", "; ", ")")
}
```


Destructuring
Start with a naïve example

```
import cats.Functor
val functorList = new Functor[List]{
  override def map[A, B](fa: List[A])(f: A => B): F[B] =
    fa.map(f)
}
def stringify2[F[_]: Functor, A, B, C](
  fa: A => String, fb: B => String, fc: C => String,
  in: F[(A, (B, C))]): String = {
  val F = implicitly[Functor[F]]
  F.map(in){case (a, (b, c)) =>
    fa(a) + ", " + fb(b) + ", " + fc(c)
  }
  ???
}
```


First step is map
Recall Functor, cats library provides one
We also need a mkString for our F, cats provides this as well

```
import cats.Show
def stringify3[F[_]: Functor, A, B, C](
  fa: A => String, fb: B => String, fc: C => String,
  in: F[(A, (B, C))])(implicit
  FS: Show[F[String]]): String = {
  val F = implicitly[Functor[F]]
  val result = F.map(in){case (a, (b, c)) =>
    fa(a) + ", " + fb(b) + ", " + fc(c)
  }
  FS.show(result)
}
```


Now we have a way to describe stringifying small things and can use that to stringify our big thing.

```
def stringify4[F[_]: Functor, A: Show, B: Show, C: Show](
  in: F[(A, (B, C))])(implicit
  FS: Show[F[String]]): String = {
  val F = implicitly[Functor[F]]
  val fa = implicitly[Show[A]].show _
  val fb = implicitly[Show[B]].show _
  val fc = implicitly[Show[C]].show _
  val result = F.map(in){case (a, (b, c)) =>
    fa(a) + ", " + fb(b) + ", " + fc(c)
  }
  FS.show(result)
}
```


Get rid of these Function1 instances since we know Show provides this for us

```
def stringify5[F[_]: Functor, A: Show, B: Show](
  in: F[(A, B)])(implicit
  FS: Show[F[String]]): String = {
  val F = implicitly[Functor[F]]
  val fa = implicitly[Show[A]].show _
  val fb = implicitly[Show[B]].show _
  val result = F.map(in){case (a, b) =>
    fa(a) + ", " + fb(b)
  }
  FS.show(result)
}
```


Build a recursive version like before…
This is not what we want
We need to recurse INSIDE the functor
We need a recursive Show instance
Just like Shapeless builds recursive instances

```
implicit def makeShow[A: Show, B: Show]: Show[(A, B)] = {
  val fa = implicitly[Show[A]].show _
  val fb = implicitly[Show[B]].show _
  new Show[(A, B)]{
    override def show(t: (A, B)): String = {
      val (a, b) = t
      "(" + fa(a) + ", " + fb(b) + ")"
    }
  }
}
```


Given any two Show instances, we can make a Show instance for the Tuple

```
def stringify[F[_]: Functor, A: Show](
  in: F[A])(implicit
  FS: Show[F[String]]): String = {
  val F = implicitly[Functor[F]]
  val fa = implicitly[Show[A]].show _
  val result = F.map(in)(fa)
  FS.show(result)
}
```


And stringify becomes this
Note the simplicity of the code here.
No more mention of tuple.
This is just the Show instance for Functor (but Show is not higher kinded)
This exact code will also work with head recursive nested tuple2 instances
In fact any nested combination of Tuple2 instances will work
Generalizing gave us MUCH more flexibility than we had before

```
implicit val ShowListString = new Show[List[String]]{
  def show(in: List[String]): String =
    in.mkString("(", "; ", ")")
}
implicit val showInt = new Show[Int]{
  override def show(in: Int): String = in.toString}
implicit val showLong = new Show[Long]{
  override def show(in: Long): String = in.toString}
implicit val showString = new Show[String]{
  override def show(in: String): String = in}
implicit val showDouble = new Show[Double]{
  override def show(in: Double): String = f"$in%.2f"}
implicit val showFloat = new Show[Float]{
  override def show(in: Float): String = f"$in%.2f"}
implicit val showArrayByte = new Show[Array[Byte]]{
  override def show(in: Array[Byte]): String = new String(in)}
```


So we define all of our individual Show instances

```
//Write the thing with our writer
val result = implicitly[Result1]

//Read the thing back with our reader
println(stringify(result))
```

@dreadedsoftware | @integrichain

And run everything like so
We can rearrange the argument types without big refactors
All our types are well defined and everything is purely functional
This type of code is a great way to get less knowledgable people spun up quickly
-- With little boilerplate one can produce a reader and writer in our library

Questions?

@dreadedsoftware
@integrichain