GENERIC DERIVATION IS THE NEW REFLECTION

Alexander Ioffe
@deusaquilus

github.com/deusaquilus/derivation_examples

SPECS

- MacBook Pro
- 2.4 GHz 8-Core
- Intel Core i9
- 64 GB 2667 MHz DDR4

Benchmarks via ScalaMeter.

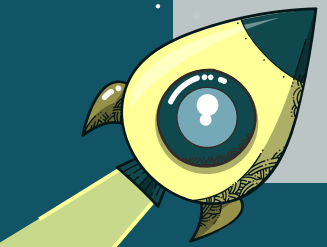
- 500-1000 Warmup Runs
- 400,000 Benchmark Runs
- ~15% Std-Dev on all metrics

01 Java Reflection

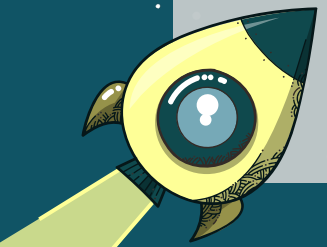
The Golden Age that never was...

GOOD OLD MUTABILITY

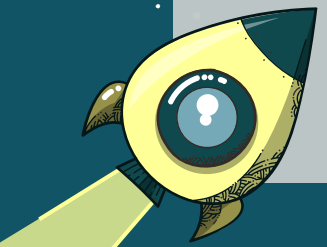
```
val p = Person("Joe", "Bloggs", 123)

val map = mutable.Map[String, Any]()
map.put("firstName",  p.firstName)
map.put("lastName",  p.lastName)
map.put("age",  p.age)

// 96ns (hey, that's pretty fast!)
```



GOOD OLD MUTABILITY

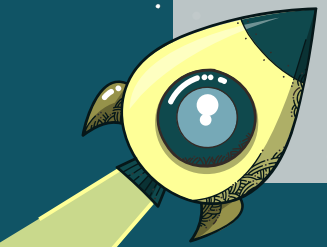


~~GOOD OLD MUTABILITY~~



~~GOOD OLD MUTABILITY~~

```scala
val list = List(
  "firstName" -> p.firstName,
  "lastName" -> p.lastName,
  "age" -> p.age)

list.foldLeft(Map[String, Any]())(
  (map, pair) => map + pair
)

// 161ns (a price we can pay?...)
```



~~GOOD OLD~~ REFLECTION

```scala
def reflectCaseClassFields[T <: Product](p: T) = {
  val fieldNames = p.getClass.getDeclaredFields.map(_.getName)
  p.getClass.getDeclaredMethods.filter { m =>
    fieldNames.contains(m.getName)
  }.toList
}
```

```scala
val fields = reflectCaseClassFields(p)
fields.foldLeft(Map[String, Any]()) {
  (map, field) =>
    map + ((field.getName, field.invoke(p)))
}
```

} 1161 ns!

} 181ns

Cache it? Amortize it?

~~GOOD OLD~~ REFLECTION

```scala
def reflectCaseClassFields[T <: Product](p: T) = {
  val fieldNames = p.getClass.getDeclaredFields.map(_.getName)
  p.getClass.getDeclaredMethods.filter { m =>
    fieldNames.contains(m.getName)
  }.toList
}
```

```scala
val fields = reflectCaseClassFields(p)
fields.foldLeft(Map[String, Any]()) {
  (map, field) =>
    map + ((field.getName, field.invoke(p)))
}
```

} 1161 ns!

} 181ns

MACRO IT!

I AM MACRO!

```scala
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }
```

```scala
def populateMapImpl[T: Type](
 from: Expr[T], map: Expr[mutable.Map[String, Any]]
)(using qctx: Quotes): Expr[Unit] = {
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val value = from.asTerm.select(f).asExpr
      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })
}
```

```scala
val p = Person("Joe", "Bloggs", 123)
val myMap = mutable.Map[String, Any]()
populateMap(p, myMap)
```

I AM MACRO!

Reflect the Person case class

```
Value = "Joe"
Key = "firstName"
'{ myMap.put("firstName", "Joe") }
```

```
{ /* block */
  '{ myMap.put("firstName", "Joe") }
  '{ myMap.put("lastName", "Bloggs") }
  '{ myMap.put("age", 123) }
  ()
}
```

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }


  def populateMapImpl[T: Type](
   from: Expr[T], map: Expr[mutable.Map[String, Any]]
  )(using qctx: Quotes): Expr[Unit] = {
    import qctx.reflect._
    val cls = TypeRepr.of[T].classSymbol.get
    val selects =
      cls.caseFields.map { f =>
        val value = from.asTerm.select(f).asExpr
        val key = Expr(f.name)
        '{ $map.put($key, $value) }
      }
    Expr.block(selects, '{ () })
  }


      val p = Person("Joe", "Bloggs", 123)
      val myMap = mutable.Map[String, Any]()
      populateMap(p, myMap)
```

I AM MACRO!

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }
```

```
def populateMapImpl[T: Type](
 from: Expr[T], map: Expr[mutable.Map[String, Any]]
)(using qctx: Quotes): Expr[Unit] = {
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val value = from.asTerm.select(f).asExpr
      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })
}
```

```
{
 myMap.put("firstName", "Joe")
 myMap.put("lastName", "Bloggs")
 myMap.put("age", 123)
 ()
}
```

```
val p = Person("Joe", "Bloggs", 123)
val myMap = mutable.Map[String, Any]()
populateMap(p, myMap)
```

I AM MACRO!

```
{
  myMap.put("firstName", "Joe")
  myMap.put("lastName", "Bloggs")
  myMap.put("age", 123)
  ()
}
```

```
val p = Person("Joe", "Bloggs", 123)
val myMap = mutable.Map[String, Any]()

{
  myMap.put("firstName", "Joe")
  myMap.put("lastName", "Bloggs")
  myMap.put("age", 123)
  ()
}
```

```
// 91ns!!!!
```

I AM MACRO!

```
val myMap = mutable.Map()
{
  myMap.put("firstName", "Joe")
  myMap.put("lastName", "Bloggs")
  myMap.put("age", 123)
  ()
}
returns:
  map.view: MapView[String, Any]
```

```
inline def populateMapBlock[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }

  def populateMapImpl[T: Type](
    from: Expr[T],
    map: Expr[mutable.Map[String, Any]]
  )(using qctx: Quotes): Expr[Unit] =
    import qctx.reflect._
    val cls = TypeRepr.of[T].classSymbol.get
    val selects =
      cls.caseFields.map { f =>
        val value = from.asTerm.select(f).asExpr
        val key = Expr(f.name)
        '{ $map.put($key, $value) }
      }
    Expr.block(selects, '{ () })
```

```
inline def populateMap[T](from: T): MapView[String, Any] =
  val myMap = mutable.Map[String, Any]()
  populateMapBlock(from, myMap)
  myMap.view
```

```
val p = Person("Joe", "Bloggs", 123)
val map = populateMap(p)
```

// 103ns!!!!

I AM MACRO!

```
val myMap = mutable.Map()
{
  myMap.put("firstName", "Joe")
  myMap.put("lastName", "Bloggs")
  myMap.put("age", 123)
  ( )
}
returns:
  map.view: MapView[String, Any]
```

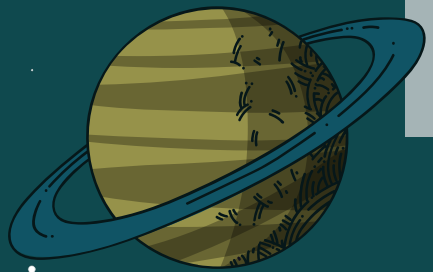
```
val p = Person("Joe", "Bloggs", 123)
val map = populateMap(p)
```

// 103ns!!!!

WHAT ABOUT HIERARCHIES?

```
case class Name(first: String, last: String)
case class Person(name: Name, age: Int)

val p = Person(Name("Joe", "Bloggs"), 123)
toMap(p)
```

```
Map(
  "name" -> Map(
    "first" -> "Joe",
    "last" -> "Bloggs"
  ),
  "age" -> 123
)
```



WHOOPS... THATS HARD!

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }

def populateMapImpl[T: Type](from: Expr[T], map: Expr[mutable.Map[String, Any]])(using qctx: Quotes): Expr[Unit] =
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val field = from.asTerm.select(f)
      val value =
        if (field.tpe <:< TypeRepr.of[Product])
          field.tpe.asType match
            case '[tpe] =>
              '{ recurseGetMap[tpe](${from.asTerm.select(f).asExprOf[tpe]}) }
        else
          from.asTerm.select(f).asExpr

      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })


inline def recurseGetMap[T](from: T): mutable.Map[String, Any] =
  val map = mutable.Map[String, Any]()
  populateMap(from, map)
  map
```

WHOOPS... THATS HARD!

Is the field a product?

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }

def populateMapImpl[T: Type](from: Expr[T], map: Expr[mutable.Map[String, Any]])(using qctx: Quotes): Expr[Unit] =
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val field = from.asTerm.select(f)
      val value =
        if (field.tpe <:< TypeRepr.of[Product])
          field.tpe.asType match
            case '[tpe] =>
              '{ recurseGetMap[tpe](${from.asTerm.select(f).asExprOf[tpe]}) }
        else
          from.asTerm.select(f).asExpr

      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })


inline def recurseGetMap[T](from: T): mutable.Map[String, Any] =
  val map = mutable.Map[String, Any]()
  populateMap(from, map)
  map
```

WHOOPS... THATS HARD!

Various voodoo needed to actually pull an Expr-Level type from a Term-Level type so that when we recurse we actually get fields...

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }

def populateMapImpl[T: Type](from: Expr[T], map: Expr[mutable.Map[String, Any]])(using qctx: Quotes): Expr[Unit] =
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val field = from.asTerm.select(f)
      val value =
        if (field.tpe <:< TypeRepr.of[Product])
          field.tpe.asType match
            case '[tpe] =>
              '{ recurseGetMap[tpe](${from.asTerm.select(f).asExprOf[tpe]}) }
        else
          from.asTerm.select(f).asExpr

      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })


inline def recurseGetMap[T](from: T): mutable.Map[String, Any] =
  val map = mutable.Map[String, Any]()
  populateMap(from, map)
  map
```

WHOOPS... THATS HARD!

If the field is indeed a product,
Then splice in a quote...

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }

def populateMapImpl[T: Type](from: Expr[T], map: Expr[mutable.Map[String, Any]])(using qctx: Quotes): Expr[Unit] =
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val field = from.asTerm.select(f)
      val value =
        if (field.tpe <:< TypeRepr.of[Product])
          field.tpe.asType match
            case '[tpe] =>
              '{ recurseGetMap[tpe](${from.asTerm.select(f).asExprOf[tpe]}) }
        else
          from.asTerm.select(f).asExpr

      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })


inline def recurseGetMap[T](from: T): mutable.Map[String, Any] =
  val map = mutable.Map[String, Any]()
  populateMap(from, map)
  map
```

WHOOPS... THATS HARD!

Which will then recurse the inline calling the initial method again on a new map it has created!

```
inline def populateMap[T](from: T, map: mutable.Map[String, Any]): Unit =
  ${ populateMapImpl('from, 'map) }

def populateMapImpl[T: Type](from: Expr[T], map: Expr[mutable.Map[String, Any]])(using qctx: Quotes): Expr[Unit] =
  import qctx.reflect._
  val cls = TypeRepr.of[T].classSymbol.get
  val selects =
    cls.caseFields.map { f =>
      val field = from.asTerm.select(f)
      val value =
        if (field.tpe <:< TypeRepr.of[Product])
          field.tpe.asType match
            case '[tpe] =>
              '{ recurseGetMap[tpe](${from.asTerm.select(f).asExprOf[tpe]}) }
        else
          from.asTerm.select(f).asExpr

      val key = Expr(f.name)
      '{ $map.put($key, $value) }
    }
  Expr.block(selects, '{ () })


inline def recurseGetMap[T](from: T): mutable.Map[String, Any] =
  val map = mutable.Map[String, Any]()
  populateMap(from, map)
  map
```

WHOOPS... THATS HARD!

```scala
case class Name(first: String, last: String)
case class Person(name: Name, age: Int)
val p = Person(Name("Joe", "Bloggs"), 123)
```

Macro

```scala
val map = mutable.Map[String, Any]()
populateMap(p, map)
```

// 157ns

Manual

```scala
val map = mutable.Map[String, Any]()
val nameMap = mutable.Map[String, Any]()
nameMap.put("first", p.name.first)
nameMap.put("last", p.name.last)
map.put("name", nameMap)
map.put("age", p.age)
```

// 156ns

Problem 01

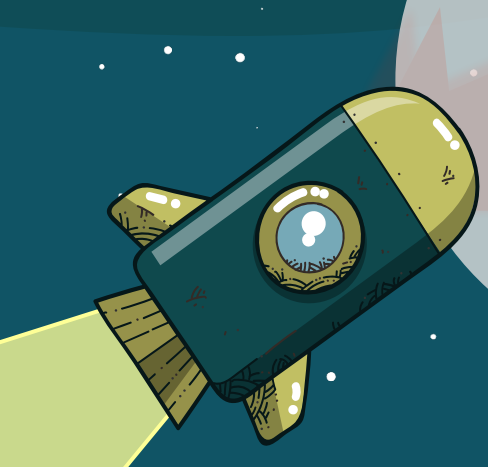
NEED EXTENSIBLE TYPES

```
case class Name(first: String, last: String)
case class ConvertableAge(value: Int) {
  def numDecades = value/10
}
case class Person(name: Name, age: ConvertableAge)


val p = Person(Name("Joe", "Bloggs"), ConvertableAge(123))
val map = mutable.Map[String, Any]()
populateMap(p, map)


Map(
  "name" -> Map("last" -> "Bloggs", "first" -> "Joe"),
  "age" -> Map("value" -> 123)
)
```

Problem 01

NEED EXTENSIBLE TYPES

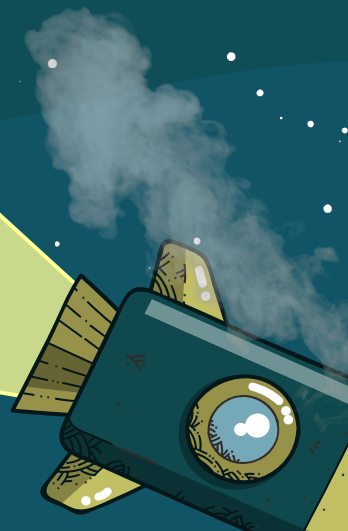
```
case class Name(first: String, last: String)
case class ConvertableAge(value: Int) {
  def numDecades = value/10
}
case class Person(name: Name, age: ConvertableAge)


val p = Person(Name("Joe", "Bloggs"), ConvertableAge(123))
val map = mutable.Map[String, Any]()
populateMap(p, map)


Map(
  "name" -> Map("last" -> "Bloggs", "first" -> "Joe"),
  "age" -> Map("value" -> 123) // WHOOPS! Should be just 123
)
```



Problem 02

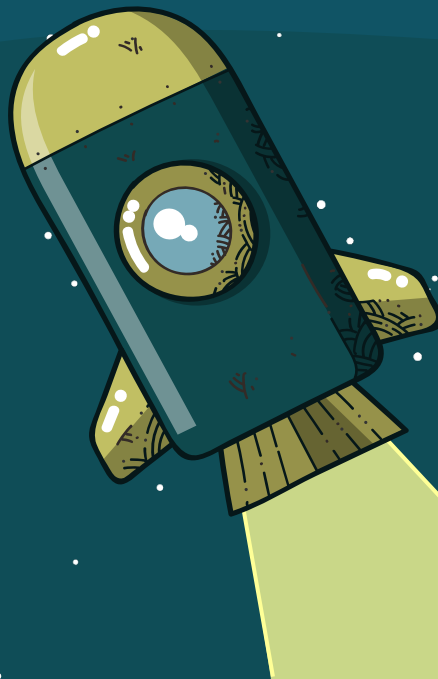
NEED POLYMORPHISM

```
trait Name
case class SimpleName(first: String, last: String) extends Name
case class Title(first: String, middle: String, last: String) extends Name
case class Person(name: Name, age: Int)

val p = Person(Name("Joe", "Bloggs"), ConvertableAge(123))
val map = mutable.Map[String, Any]()
populateMap(p, map)


Map(
  "name" -> Map(???) // What fields go here???,
  "age" -> Map("value" -> 123)
)
```



02 RETHINKING IT

If developer sanity was a real concern,
how would we do it?



Reflection Reimagined

FAST

EXTENSIBLE

POLYMORPHIC

Typeclass Derivation

FAST

Uses Compile-Time Macros

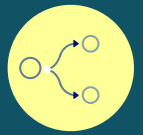
EXTENSIBLE

Composed with Typeclasses

POLYMORPHIC

Products / Co-Products

FIRST SOME PREREQUISITES



Tuple Composition

Type Matching

Products / Coproducts

N-Ary Tuple Composition:

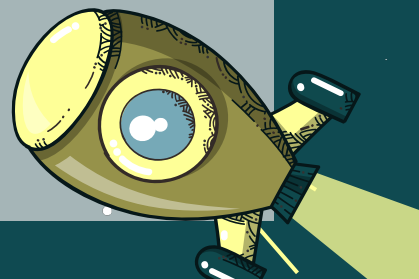
```
("foo", "bar") => ("foo" *: ("bar" *: (EmptyTuple)))
```

The *: Operator Composes Tuples:

```
("foo" *: ("bar", "baz")) == ("foo", "bar", "baz")
```

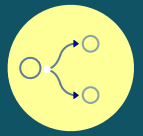
It Also Decomposes them:

```
("foo", "bar", "baz") match
  case (foo *: barbaz) =>
    foo == "foo"
    barbaz == ("bar", "baz")
```



TupleComposition.scala

FIRST SOME PREREQUISITES



Tuple Composition

Type Matching

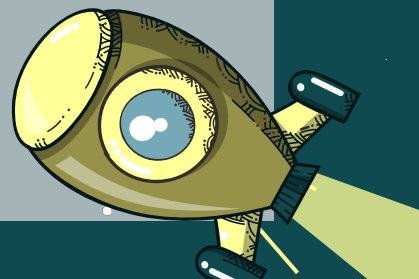
Products / Coproducts

It can also match by type:

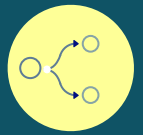
```
("foo", 123) match
  case tup: (String *: (Int *: EmptyTuple)) =>
    println("Matched The Tuple: " + tup)
```

It can also match by type:

```
("foo", 123) match
  case tup: (String, Int) =>
    println("Matched The Tuple: " + tup)
```



FIRST SOME PREREQUISITES



Tuple Composition

Type Matching

Products / Coproducts

It can also match by type:

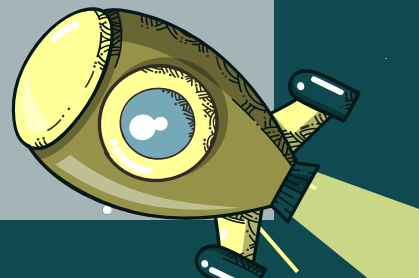
```
("foo", 123) match
  case tup: (String *: (Int *: EmptyTuple)) =>
    println("Matched The Tuple: " + tup)
```

Which we don't need to actually extract...

```
("foo", 123) match
  case _: (String *: (Int *: EmptyTuple)) =>
    println("Matched The Tuple")
```

The types can be wildcards (they are always lower-case):

```
("foo", 123) match
  case _: (String *: stuff) =>
    doStuffWith[stuff]
```



FIRST SOME PREREQUISITES

Tuple Composition

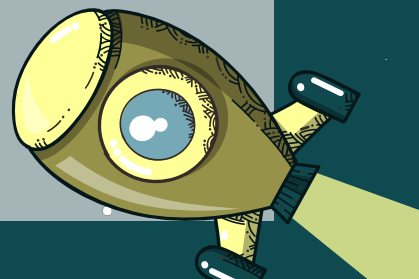
Type Matching

Products / Coproducts

Inline methods can use 'erasedValue' to match a type of a type-parameter:

```
inline def matchType[T]: String =
  inline erasedValue[T] match
    case _: String => "It's a String"
    case _: Int    => "It's an Int"
```

'erasedValue' can only be from an inline-method i.e. macro, i.e. compile-time construct since type-information is erased at runtime...



FIRST SOME PREREQUISITES

Tuple Composition

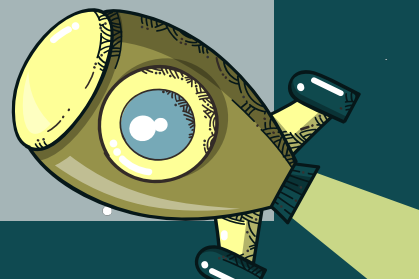
Type Matching

Products / Coproducts

Inline methods can use 'erasedValue' to match a type of a type-parameter:

```
inline def matchType[T]: String =
  inline erasedValue[T] match
    case _: String => "It's a String"
    case _: Int    => "It's an Int"
```

```
> println(matchType[Int])
"It's an Int"
```



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

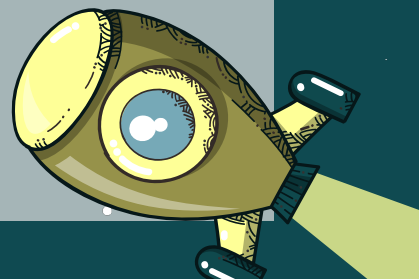
Products / Coproducts

Inline methods can use 'erasedValue' to match a type of a type-parameter:

```
inline def matchType[T]: String =
  inline erasedValue[T] match
    case _: String => "It's a String"
    case _: Int    => "It's an Int"
```

```
> println( {{case _: String => "It's a String"}} )
"It's an Int"
```

Spliced During Compile-Time



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

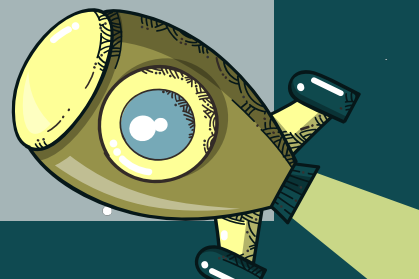
Products / Coproducts

Inline methods can use 'erasedValue' to match a type of a type-parameter:

```
inline def matchType[T]: String =
  inline erasedValue[T] match
    case _: String => "It's a String"
    case _: Int    => "It's an Int"
```

```
> println( {{"It's a String"}} )
"It's an Int"
```

Spliced During Compile-Time



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Inline methods can use 'erasedValue' to match a type of a type-parameter:

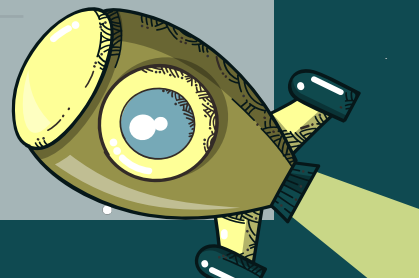
```
inline def matchType[T]: String =
  inline erasedValue[T] match
    case _: String => "It's a String"
    case _: Int    => "It's an Int"
```

The tuple *: operator can also be used to match a tuple, even with wildcards

```
inline def matchTup[T] =
  inline erasedValue[T] match
    case _: (String *: suffix) =>
```

Type-Wildcards can be used as extractors

```
inline def matchTup[T] =
  inline erasedValue[T] match
    case _: (String *: suffix) => doStuffWith[suffix]
```



FIRST SOME PREREQUISITES

Tuple Composition

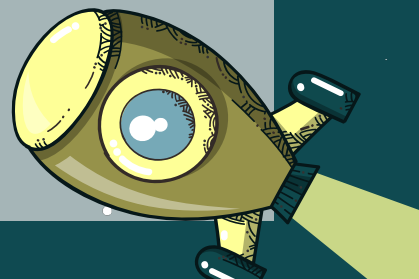
Type Matching

Products / Coproducts

Typically, Type-Wildcards are use to Summon Typeclass Instances

```
trait Fooable[T]:
  def fooIt: String
given Fooable[String] with
  def fooIt = "String-Fooable"
given Fooable[Int] with
  def fooIt = "Int-Fooable"

inline def matchSummonType[T] =
  inline erasedValue[T] match
    case _: (prefix *: suffix) =>
      summonInline[Fooable[prefix]].fooIt
```

```
> println(matchSummonType[(Int, String)].fooIt)
Int-Fooable
```



FIRST SOME PREREQUISITES


Tuple Composition

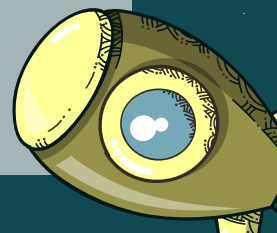

Type Matching


Products / Coproducts

With Tuples, this can be done recursively

```
trait Fooable[T]:
  def fooIt: String
given Fooable[String] with
  def fooIt = "String-Fooable"
given Fooable[Int] with
  def fooIt = "Int-Fooable"

inline def summonRecurse[T]: List[String] =
  inline erasedValue[T] match
    case _: (prefix *: suffix) =>
      summonInline[Fooable[prefix]].fooIt +: summonRecurse[suffix]
    case _: EmptyTuple =>
      Nil
```

```
> println(summonRecurse[(String, Int, String)])
List(String-Fooable, Int-Fooable, String-Fooable)
```



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

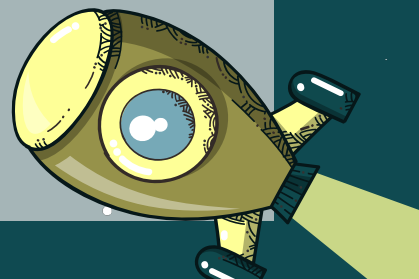
Just FYI

Type-holes can be matched too

```
inline def matchTup[T] =
  inline erasedValue[T] match
    case _: (Option[optType] *: suffix) =>
```

Types can be used to do filters

```
type IsProduct[T <: Product]
inline def matchTup[T] =
  inline erasedValue[T] match
    case _: (IsProduct[productType] *: suffix) =>
```



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Product:

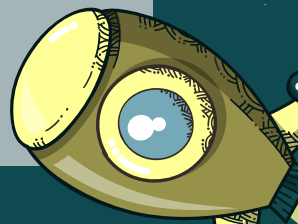
Examples:

Scala Case Class:

```
case class Plate(
  topSlot: Grain,
  leftSlot: Vegetable,
  rightSlot: Protein
)
```

Scala Tuple:

```
(Grain, Vegetable, Protein)
```



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Product:

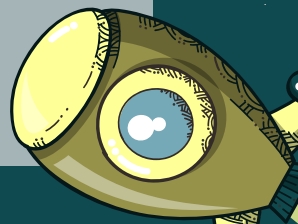
Examples:

Scala Case Class:

```
case class Person(
  firstName: String,
  lastName: String,
  age: Int
)
```

Scala Tuple:

```
(String, String, Int)
```



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

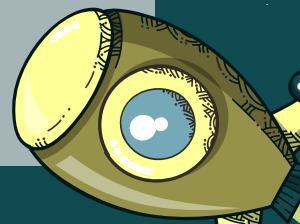
Products / Coproducts

Product:

Examples:

Scala 3 Case:

```
enum Utensil {
  case Plate(
    topSlot: Grain,
    leftSlot: Vegetable,
    rightSlot: Protein
  )
}
```



FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Co-product / Sum:

Choose...

Examples:

Scala 2 Sealed Trait:

```
sealed trait Serving
object Jam extends Serving
case class Nuts(quantity: Int) extends Serving
case class Strawberries(quantity: Int) extends Serving
```

FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Co-product / Sum:

Examples:

Scala 2 Sealed Trait:

```
sealed trait Serving
object Jam extends Serving
case class Nuts(quantity: Int) extends Serving
case class Strawberries(quantity: Int) extends Serving
```

Scala 3 Enum:

```
enum Serving:
  case Jam extends Serving
  case Nuts(quantity: Int) extends Serving
  case Strawberries(quantity: Int) extends Serving
```

FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Combine Products & Co-Products => Describe (Almost) Any Business Domain

```
enum Plate:
  case TwoSlot(leftSlot: Food, rightSlot: Food)
  case ThreeSlot(topSlot: Food, leftSlot: Food, rightSlot: Food)

enum Food:
  case Chicken
  case Peas(quantity: Int)
  case Rice(riceType: RiceType)

enum RiceType:
  case Jasmine
  case Brown
  case White
```

FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Mirrors are Types:

```
> summon[Mirror.Of[Person]]
Mirror.ProductOf[Person] {
  MirroredElemTypes = (String, String, Int);
  MirroredElemLabels = ("firstName", "lastName", "age")
}

> summon[Mirror.Of[Food]]
Mirror.SumOf[Person] {
  MirroredElemTypes = (Chicken, Peas, Rice);
  MirroredElemLabels = ("Chicken", "Peas", "Rice")
}
```

Product

```
case class Person(
  firstName: String,
  lastName: String,
  age: Int
)
```

Co-Product / Sum

```
enum Food:
  case Chicken
  case Peas(quantity: Int)
  case Rice(riceType: RiceType)
```

FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Mirrors are Types:

```
> summon[Mirror.Of[Person]]
> summon[Mirror.Of[Food]]
```

```
inline def summonRecurse[T]: List[String] =
  inline erasedValue[T] match
    case _: (prefix *: suffix) =>
      summonInline[Fooable[prefix]].fooIt +: summonRecurse[suffix]
    case _: EmptyTuple =>
      Nil

summonInline[Mirror.Of[Person]] match
  case m: Mirror.ProductOf[Person] =>
    summonRecurse[m.MirroredElemLabels]
```

Product

```
case class Person(
  firstName: String,
  lastName: String,
  age: Int
)
```

Co-Product / Sum

```
enum Food:
  case Chicken
  case Peas(quantity: Int)
  case Rice(riceType: RiceType)
```

FIRST SOME PREREQUISITES

Tuple Composition

Type Matching

Products / Coproducts

Mirrors are Types:

```
> summon[Mirror.Of[Person]]
> summon[Mirror.Of[Food]]
```

```
inline def summonRecurse[T]: List[String] =
  inline erasedValue[T] match
    case _: (prefix *: suffix) =>
      summonInline[Fooable[prefix]].fooIt +: summonRecurse[suffix]
    case _: EmptyTuple =>
      Nil

summonInline[Mirror.Of[Food]] match
  case m: Mirror.SumOf[Food] =>
    summonRecurse[m.MirroredElemLabels]
```

Product

```
case class Person(
  firstName: String,
  lastName: String,
  age: Int
)
```

Co-Product / Sum

```
enum Food:
  case Chicken
  case Peas(quantity: Int)
  case Rice(riceType: RiceType)
```

Generic Derivation - Take 1

This case class derives a generic Typeclass 'WriteToMap'

```
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

---

given WriteToMap[Person] = WriteToMap.derived
val map = mutable.Map[String, Any]()
summon[WriteToMap[Person]].writeToMap(map)("", p)
> println(map)

---

object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

This case class derives a generic Typeclass 'WriteToMap'

```
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

---

given WriteToMap[Person] = WriteToMap.derived
val map = mutable.Map[String, Any]()
summon[WriteToMap[Person]].writeToMap(map)("", p)
> println(map)

---

object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[("firstName", "lastName", "age")]
          val writers = recurseTypes[(String, String, Int)]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

This case class derives a generic Typeclass 'WriteToMap'

```
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

---

given WriteToMap[Person] = WriteToMap.derived
val map = mutable.Map[String, Any]()
summon[WriteToMap[Person]].writeToMap(map)("", p)
> println(map)

---

object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

What kind of mirror is it?

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}
```

Generic Derivation - Take 1

If it is a product mirror...

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

type Names =
("firstName", "lastName", "age")

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

(name := **"firstName"**
 *: ("lastName",
  *: ("age", EmptyTuple)))

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

"firstName" +:
recurse(
("lastName", *: ("age", EmptyTuple)))
)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

type Names =
("lastName", "age")

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

(name := **"lastName"**,
 *: ("age", EmptyTuple))

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

"firstName" +: "lastName"
recurse(
("age", EmptyTuple))
)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

type Names =
("age")

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

(name := **"age"**, EmptyTuple)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

```
"firstName" +: "lastName" +: "age"
  recurse(
    (EmptyTuple)
  )
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

type Names =
(EmptyTuple)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

(EmptyTuple)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:
("firstName", "lastName", "age")

"firstName" +: "lastName" +: "age" + Nil

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

Get the fields:

List("firstName", "lastName", "age")

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseNames[Names <: Tuple]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      constValue[name].toString +: recurseNames[names]
    case _: EmptyTuple =>
      Nil
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

type Types =
  (String, String, Int)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

(tpe := **String**
 *: (String,
  *: (Int, EmptyTuple)))

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

writeStr +:
recurse(
(String, *: (Int, EmptyTuple)))
)

Summon This:

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

type Types =
  (String, Int)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

(tpe := **String**,
 *: (Int, EmptyTuple))

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

writeStr +: writeStr
recurse(
("age", EmptyTuple))
)

Summon This:

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

type Types =
  (Int)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

writeStr +: writeStr +: writeInt
recurse(
(EmptyTuple)
)

Summon This:

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

type Types =
  (EmptyTuple)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

(EmptyTuple)

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:
(writeStr, writeStr, writeInt)

writeStr +: writeStr +: writeInt + Nil

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

From Elem Types:
(String, String, Int)
Get the writers:

List(**writeStr, writeStr, writeInt**)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

("firstName", "lastName", "age") zip
(writeStr, writeStr, writeInt)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

("firstName", "lastName", "age") zip
(writeStr, writeStr, writeInt) zip
(1,2,3)

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
(("firstName", writeStr), 1),
(("lastName", writeStr), 2),
(("age", writeInt), 3)
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
(("firstName", writeStr), 1) =>
  map.put("firstName", writeStr(values(1)))
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
((("firstName", writeStr), 1) =>
  map.put("firstName", writeStr("Joe"))
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
((("firstName", writeStr), 1) =>
  map.put("lastName", writeStr(values(2)))
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
((("firstName", writeStr), 1) =>
  map.put("lastName", writeStr("Bloggs"))
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
(("firstName", writeStr), 1) =>
  map.put("age", writeStr(values(3)))
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1

```
((("firstName", writeStr), 1) =>
  map.put("age", writeInt(123))
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}

inline def recurseTypes[Types <: Tuple]: List[WriteToMap[Any]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      summonWriter[tpe].asInstanceOf[WriteToMap[Any]] +: recurseTypes[types]
    case _: EmptyTuple =>
      Nil

given writeInt: WriteToMap[Int] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Int): Unit =
    map.put(key, value)

given writeStr: WriteToMap[String] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: String): Unit =
    map.put(key, value)
```

Generic Derivation - Take 1


```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        val names = recurseNames[proMir.MirroredElemLabels]
        val writers = recurseTypes[proMir.MirroredElemTypes]
        names.zip(writers).zipWithIndex.map {
          case ((name, writer), i) =>
            writer.writeToMap(map)(name, value.productIdx(i))
        }
}
```

Generic Derivation - Take 1


```
extension [T](value: T)(using wtm: WriteToMap[T])
  def writeToMap: MapView[String, Any] =
    val map = mutable.Map[String, Any]()
    wtm.writeToMap(map)("", value)
    map.view


case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap  // 214ns 😔


object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

Allocate a list for each component...

```
writeToMap[Person](p)(
  new Person().derived$WriteToMap)
{
  ...
  val names: List[String] = ({
    val elem$1: String = "firstName"
    ({
      val `elem$1₂`: String = "lastName"
      ({
        val `elem$1₃`: String = "age"
        (Nil: List[String]).+(`elem$1₃`)
      }: List[String]).+:(`elem$1₂`)
    }: List[String]).+:(elem$1)
  }: List[String])
  ...
}
```

```scala
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap // 214ns 😔


object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

Allocate a list for each component again...

```
writeToMap[Person](p)(
  new Person().derived$WriteToMap)
{
  ...
  val writers: List[WriteToMap[Any]] = ({
    val elem$2: WriteToMap[Any] = ({
      val t: writeString.type = writeString

      (t: writeString)
    }: WriteToMap[String]).asInstanceOf[WriteToMap[Any]]
    ({
      val `elem$2₂`: WriteToMap[Any] = ({
        val `t₂`: writeString.type = writeString

        (`t₂`: writeString)
      }: WriteToMap[String]).asInstanceOf[WriteToMap[Any]]
      ({
        val `elem$2₃`: WriteToMap[Any] = ({
          val `t₃`: writeInt.type = writeInt

          (`t₃`: writeInt)
        }: WriteToMap[Int]).asInstanceOf[WriteToMap[Any]]
        (Nil: List[WriteToMap[Any]]).+:[WriteToMap[Any]](`elem$2₃`)
      }: List[WriteToMap[Any]]).+:[WriteToMap[Any]](`elem$2₂`)
    }: List[WriteToMap[Any]]).+:[WriteToMap[Any]](elem$2)
  }: List[WriteToMap[Any]])
  ...
}
```

```scala
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap // 214ns 😔
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

and.. Allocate another list...

```scala
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap  // 214ns 😔
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

and.. Allocate another list...

```scala
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap  // 214ns 😔
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Generic Derivation - Take 1

```scala
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap  // 214ns 😔
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

and.. Allocate another list...!
(actually we should use a foreach here,
I tried it but it didn't affect performance)

Generic Derivation - Take 1


```scala
case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap  // Allocated: 1 Map + 8 Lists = 214ns 😔
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          val names = recurseNames[proMir.MirroredElemLabels]
          val writers = recurseTypes[proMir.MirroredElemTypes]
          names.zip(writers).zipWithIndex.map {
            case ((name, writer), i) =>
              writer.writeToMap(map)(name, value.productIdx(i))
          }
  }
}
```

Can We Do Better?

Can We Do Both in One?


```
inline def recurseNames[Names]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      ...
```

```
inline def recurseTypes[Types]: List[WriteToMap]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      ...
```

```
inline erasedValue[(Names, Types)] match
  case (_: (name *: names), _: (tpe *: types)) =>
```

Can We Do Both in One?


```scala
inline def recurseNames[Names]: List[String] =
  inline erasedValue[Names] match
    case _: (name *: names) =>
      ...

inline def recurseTypes[Types]: List[WriteToMap]] =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      ...

inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

Can We Do Both in One?


```scala
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

What kind of mirror is it?

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

If it is a product mirror...

Can We Do Both in One?

```scala
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

type Names =
("firstName", "lastName", "age")
type Types =
(String, String, Int)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

(name := "firstName" *:
  "lastName", "age")

(tpe := String *: String, Int)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

key := "firstName"

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

value := "Joe"

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

summon: writeStr

writeStr
.writeToMap(map)("firstName", "Joe")

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

```
recurse[
  ("lastName", "age"),
  (String, Int)
]
```

```scala
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

type Names =
("lastName", "age")
type Types =
(String, Int)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

(name := "lastName" *: "age")

(tpe := String *: Int)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

key := "lastName"

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

value := "Bloggs"

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

summon: writeStr

writeStr
  .writeToMap(map)("lastName", "Bloggs")

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

```
recurse[
  ("age"),
  (Int)
]
```

```scala
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

type Names =
("age")
type Types =
(Int)

```scala
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

(name := "age")

(tpe := Int)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

key := "age"

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

value := 123

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

summon: writeInt

writeStr
  .writeToMap(map)("age", 123)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

```
recurse[
  (EmptyTuple),
  (EmptyTuple)
]
```

```
Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them
```

```scala
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

type Names =
  (EmptyTuple)
type Types =
  (EmptyTuple)

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

Can We Do Both in One?

Nothing more to do...

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

Done!

Get the fields:
("firstName", "lastName", "age")
and the types:
(String, String, Int)
and use them

```
inline def recurse[Names, Types](element: Product, map: mutable.Map[String, Any])(index: Int): Unit =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      summonWriter[tpe].writeToMap(map)(key, value)
      recurse[names, types](element, map)(index + 1)
    case (_: EmptyTuple, _) =>
      // Ignore
```

```
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
```

Can We Do Both in One?

```scala
extension [T](value: T)(using wtm: WriteToMap[T])
  def writeToMap: MapView[String, Any] =
    val map = mutable.Map[String, Any]()
    wtm.writeToMap(map)("", value)
    map.view


case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)

val map = p.writeToMap  // 103ns!  (Same ns as Pure Macro, 96ns was Manual)
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
}
```

Can We Do Both in One?

```
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```
({
  val `key₂`: String = "firstName".toString()
  val `value₂`: String = element$proxy1.productElement(0).asInstanceOf[String]
  ({
    val t: writeString.type = writeString
    (t: writeString)
  }: WriteToMap[String]).writeToMap(map)(`key₂`, `value₂`)
  ({
    val `key₃`: String = "lastName".toString()
    val `value₃`: String = element$proxy1.productElement(1).asInstanceOf[String]
    ({
      val `t₂`: writeString.type = writeString
      (`t₂`: writeString)
    }: WriteToMap[String]).writeToMap(map)(`key₃`, `value₃`)
    ({
      val `key₄`: String = "age".toString()
      val `value₄`: Int = element$proxy1.productElement(2).asInstanceOf[Int]
      ({
        val `t₃`: writeInt.type = writeInt
        (`t₃`: writeInt)
      }: WriteToMap[Int]).writeToMap(map)(`key₄`, `value₄`)
      ((): Unit)
    }: Unit)
  }: Unit)
}: Unit)
```

Can We Do Both in One?

```
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```
({
  val `key₂`: String = "firstName"
  val `value₂`: String = "Joe"
  ({
    val t: writeString        = writeString

  }).writeToMap(map)(`key₂`, `value₂`)
  ({
    val `key₃`: String = "lastName"
    val `value₃`: String = "Bloggs"
    ({
      val `t₂`: writeString        = writeString

    }).writeToMap(map)(`key₃`, `value₃`)
    ({
      val `key₄`: String = "age"
      val `value₄`: Int = 123
      ({
        val `t₃`: writeInt        = writeInt

      }).writeToMap(map)(`key₄`, `value₄`)
      (():  Unit)
    }: Unit)
  }: Unit)
}: Unit)
```

Can We Do Both in One?

```
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```
({
  val `key₂`: String = "firstName"
  val `value₂`: String = "Joe"
  ({
    writeString

  }).writeToMap(map)("firstName", "Joe"₂)
  ({


    ({
      writeString₂

    }).writeToMap(map)("lastName"₃, "Bloggs"₃)
    ({


      ({
        writeInt₃

      }).writeToMap(map)(`key₄`, `value₄`)
      ((): Unit)
    }: Unit)
  }: Unit)
}: Unit)
```

Can We Do Both in One?

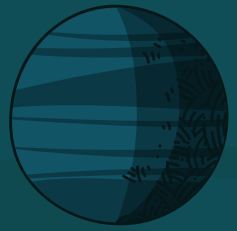
```
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```
{ writeString.writeToMap(map)("firstName", "Joe"₂)
  { writeString.writeToMap(map)("lastName"₃, "Bloggs"₃)
    { writeInt.writeToMap(map)(`key₄`, `value₄`)
    ((): Unit)
  }: Unit)
}: Unit)
}: Unit)
```

Can We Do Both in One?

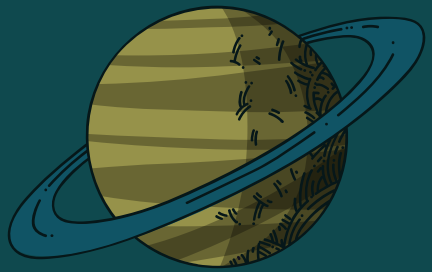
```
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```
{ map.put("firstName", "Joe"₂)
  { map.put("lastName"₃, "Bloggs"₃)
    { map.put(`key₄`, `value₄`)
    ((): Unit)
  }: Unit)
}: Unit)
}: Unit)
```



“Doing literally the Same Thing,
will frequently yield
literally the Same Results.”

—SOMEONE IN HISTORY



Wait... how did you get this?

```scala
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```scala
({
  val `key₂`: String = "firstName".toString()
  val `value₂`: String = element$proxy1.productElement(0).asInstanceOf[String]
  ({
    val t: writeString.type = writeString
    (t: writeString)
  }: WriteToMap[String]).writeToMap(map)(`key₂`, `value₂`)
  ({
    val `key₃`: String = "lastName".toString()
    val `value₃`: String = element$proxy1.productElement(1).asInstanceOf[String]
    ({
      val `t₂`: writeString.type = writeString
      (`t₂`: writeString)
    }: WriteToMap[String]).writeToMap(map)(`key₃`, `value₃`)
    ({
      val `key₄`: String = "age".toString()
      val `value₄`: Int = element$proxy1.productElement(2).asInstanceOf[Int]
      ({
        val `t₃`: writeInt.type = writeInt
        (`t₃`: writeInt)
      }: WriteToMap[Int]).writeToMap(map)(`key₄`, `value₄`)
      (((): Unit)
    }: Unit)
  }: Unit)
}: Unit)
```

PrintMac.scala

Wait... how did you get this?

```scala
val map = p.writeToMap
// 103ns! (Same ns as Pure Macro, 96ns was Manual)
```

```scala
((
  val `key₁`: String = "firstName".toString()
  val `value₁`: String = element$proxy1.productElement(0).asInstanceOf[String]
  ((
    val t: writeString.type = writeString
    (t: writeString)
  ): WriteToMap[String]).writeToMap(map)(`key₁`, `value₁`)
  ((
    val `key₁`: String = "lastName".toString()
    val `value₁`: String = element$proxy1.productElement(1).asInstanceOf[String]
    ((
      val `t₁`: writeString.type = writeString
      (`t₁`: writeString)
    ): WriteToMap[String]).writeToMap(map)(`key₁`, `value₁`)
    ((
      val `key₁`: String = "age".toString()
      val `value₁`: Int = element$proxy1.productElement(2).asInstanceOf[Int]
      ((
        val `t₁`: writeInt.type = writeInt
        (`t₁`: writeInt)
      ): WriteToMap[Int]).writeToMap(map)(`key₁`, `value₁`)
      (((): Unit)
    ): Unit)
  ): Unit)
): Unit)
```

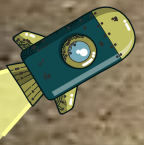
```scala
object PrintMacPass {
  inline def apply[T](inline any: T): T = ${ printMacImpl('any) }
  def printMacImpl[T: Type](any: Expr[T])(using qctx: Quotes): Expr[T] = {
    import qctx.reflect._
    println(Printer.TreeShortCode.show(any.asTerm))
    any
  }
}
```

```scala
object WriteToMap {
  inline def derived[T](using mir: Mirror.Of[T]) = PrintMacPass(new WriteToMap[T] {
    def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
})
```

Print's on the next recompile

PrintMac.scala

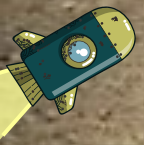
AN UNRELATED NOTE



```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      case _ =>
        throw new IllegalArgumentException(s"No mirror found for ${value}")
}
```

```scala
inline def derived[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

AN UNRELATED NOTE



```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      case _ =>
        throw new IllegalArgumentException(s"No mirror found for ${value}")
}
```

```scala
inline def derived[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

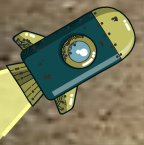
AN UNRELATED NOTE

```scala
inline def derived[T](using mir: Mirror.Of[T]) = new WriteToMap[T] {
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
    inline mir match
      case proMir: Mirror.ProductOf[T] =>
        recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      case _ =>
        throw new IllegalArgumentException(s"No mirror found for ${value}")
}
```

```scala
inline def derived[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

AN UNRELATED NOTE

DERIVE FOR ALL [T]



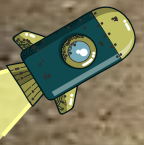
```scala
inline given writeArbitraryToMap[T]: WriteToMap[T] = WriteToMap.derived

case class Person(firstName: String, lastName: String, age: Int) // derives WriteToMap
val p = Person("Joe", "Bloggs", 123)
PrintMac(p.writeToMap)
val map = p.writeToMap // 105ns!
```

You don't need to use it anymore!

```scala
inline def derived[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

AN UNRELATED NOTE

DERIVE FOR ALL [T]



```
inline given writeArbitraryToMap[T]: WriteToMap[T] = WriteToMap.derived

case class Person(firstName: String, lastName: String, age: Int) derives WriteToMap
val p = Person("Joe", "Bloggs", 123)
PrintMac(p.writeToMap)
val map = p.writeToMap // 105ns!
```

... but you still can!

```
inline def derived[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

AN UNRELATED NOTE

```scala
inline given writeArbitraryToMap[T]: WriteToMap[T] = WriteToMap.derived

case class Person(firstName: String, lastName: String, age: Int) // derives WriteToMap
val p = Person("Joe", "Bloggs", 123)
PrintMac(p.writeToMap)
val map = p.writeToMap // 105ns!
```

summonFrom also does it

```scala
inline def derived[T] =
  summonFrom {
    case mir: Mirror.Of[T] =>
      inline mir match
        case proMir: Mirror.ProductOf[T] =>
          new WriteToMap[T] {
            def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
              recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
          }
        case _ =>
          throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
  }
```

AN UNRELATED NOTE

DERIVE FOR ALL [T]

```
inline given writeArbitraryToMap[T]: WriteToMap[T] = WriteToMap.derived

case class Person(firstName: String, lastName: String, age: Int) // derives WriteToMap
val p = Person("Joe", "Bloggs", 123)
PrintMac(p.writeToMap)
val map = p.writeToMap // 105ns!
```

---

can also use 'given'

```
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

AN UNRELATED NOTE

DERIVE FOR ALL [T]

```
inline given writeArbitraryToMap[T]: WriteToMap[T] = WriteToMap.derived

case class Person(firstName: String, lastName: String, age: Int) // derives WriteToMap
val p = Person("Joe", "Bloggs", 123)
PrintMac(p.writeToMap)
val map = p.writeToMap // 105ns!
```

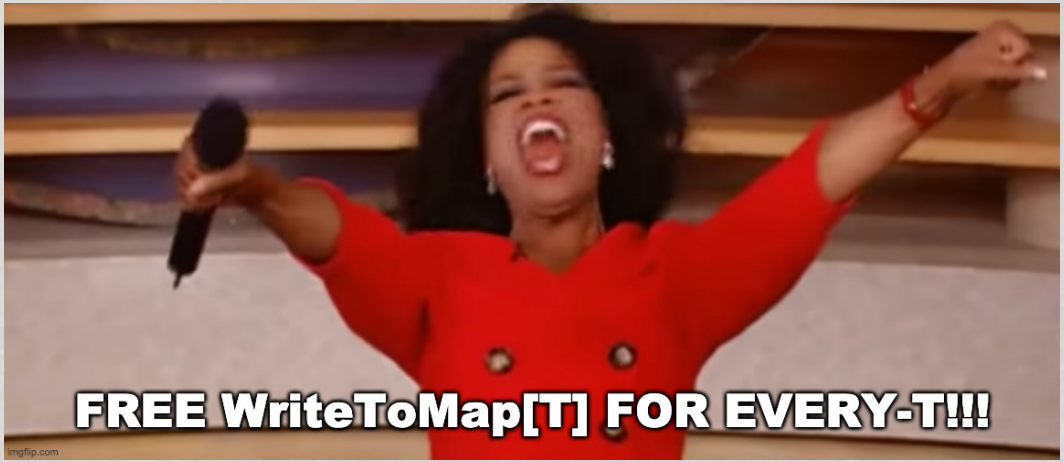
---

this also works... for now

```
implicit inline def derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T] {
        def writeToMap(map: mutable.Map[String, Any])(key: String, value: T): Unit =
          recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product], map)(0)
      }
    case _ =>
      throw new IllegalArgumentException(s"No mirror found for ${summonInline[Type[T]]}")
```

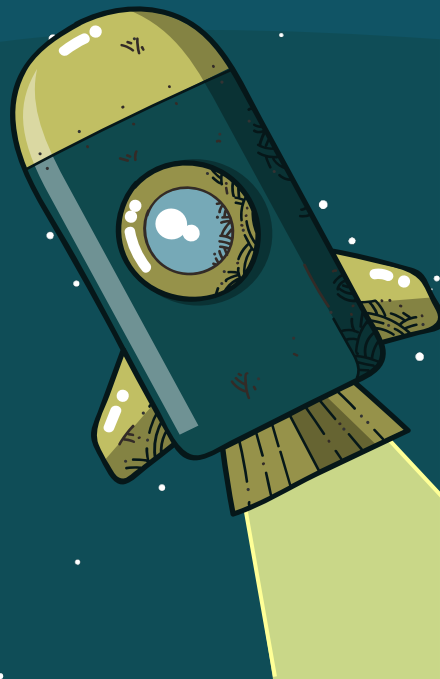
AN UNRELATED NOTE

```
inline given writeArbitraryToMap[T]: WriteToMap[T] = WriteToMap.derived
```



03 ... BUT DOES IT ACTUALLY WORK

when you introduce the corporate nightmare
of data complexity!



EXTENSIBILITY

```scala
case class Person(firstName: String, lastName: String, age: Long) derives WriteToMap
val q = Person("Q", "Q", 4000000000L)
```

```scala
given writeLong: WriteToMap[Long] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, value: Long): Unit =
    map.put(key, value)
```

```scala
val map = q.writeToMap  // 104ns!
```

EXTENSIBILITY

```scala
case class ConvertableAge(value: Int):
  def numDecades = value/10

case class Person(firstName: String, lastName: String, age: ConvertableAge) derives WriteToMap
val q = Person("Joe", "Bloggs", ConvertableAge(123))
```

```scala
given writeConvertableAge: WriteToMap[ConvertableAge] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, ca: ConvertableAge): Unit =
    map.put(key, ca.value)
```

```scala
val map = q.writeToMap
```

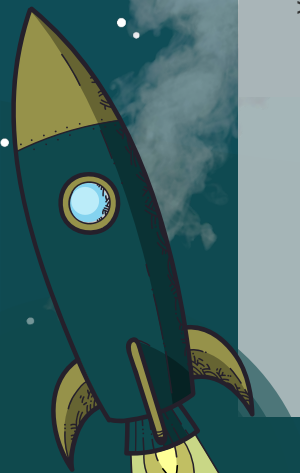
EXTENSIBILITY

How do you do this?

```
case class Person(firstName: String, lastName: String, nicknames: List[String])

case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, nicknames: List[Address])
```

```
val p = Person("Yosef", "Bloggs", List("Joseph", "Joe"))
> p.writeToMap
```

```
Map(
  "lastName" -> "Bloggs",
  "firstName" -> "Yosef",
  "nicknames" -> List("Joseph", "Joe")
)
```



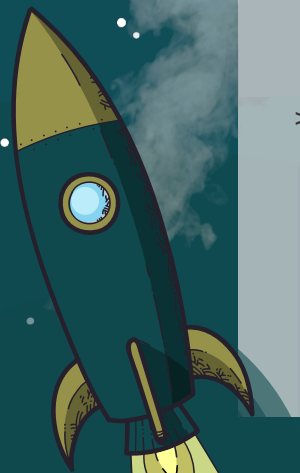
EXTENSIBILITY

How do you do this?

```
case class Person(firstName: String, lastName: String, nicknames: List[String])

case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, nicknames: List[Address])
```

```
val p =
  Person("Yosef", "Bloggs", List(
    Address("123 Place", 11122),
    Address("456 Ave", 11122))
  )
> p.writeToMap
```

```
Map(
  "lastName" -> "Bloggs",
  "firstName" -> "Yosef",
  "nicknames" -> List(
    Map("zip" -> 11122, "street" -> "123 Place"),
    Map("zip" -> 11122, "street" -> "456 Ave")
  )
)
```

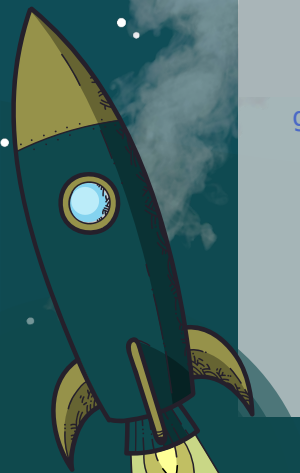


EXTENSIBILITY

How do you do this?

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])

case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, nicknames: List[Address])
```

```scala
given writeList[T](using wtm: WriteToMap[T]): WriteToMap[List[T]] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, values: List[T]): Unit =
    val valueKeys =
      values.map { v =>
       ???
      )
    map.put(key, valueKeys)
```

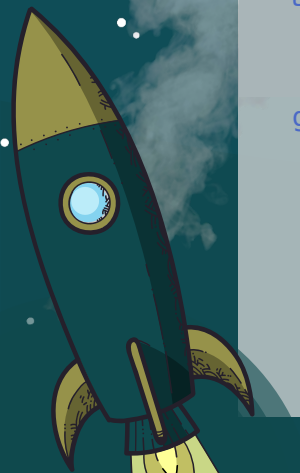


EXTENSIBILITY

Maybe like this?

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])

case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, nicknames: List[Address])
```

```scala
trait WriteToMap[T]:
  def writeToMap(mapOrReturn: mutable.Map[String, Any], justReturn: Boolean)(key: String, value: T): Any


given writeList[T](using wtm: WriteToMap[T]): WriteToMap[List[T]] with
  def writeToMap(map: mutable.Map[String, Any])(key: String, values: List[T]): Unit =
    val valueKeys =
      values.map { v =>
       ???
      )
    map.put(key, valueKeys)
```

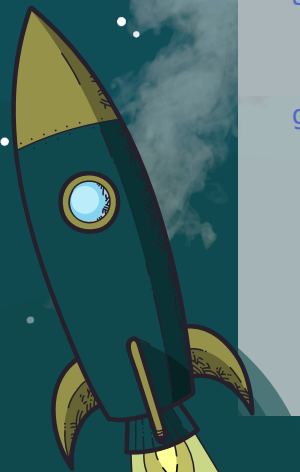


EXTENSIBILITY

Maybe like this?

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])

case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, addresses: List[Address])
```

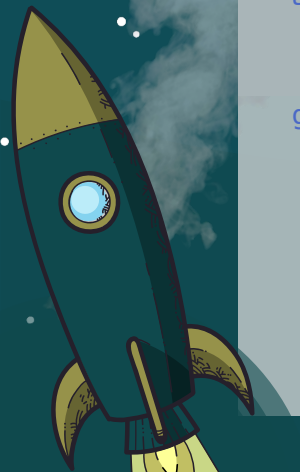
```scala
trait WriteToMap[T]:
  def writeToMap(mapOrReturn: mutable.Map[String, Any], justReturn: Boolean)(key: String, value: T): Any


given writeList[T](using wtm: WriteToMap[T]): WriteToMap[List[T]] with
  def writeToMap(map: mutable.Map[String, Any], justReturn: Boolean)(key: String, values: List[T]): Any =
    val valueKeys =
      values.map(v =>
        wtm.writeToMap(null, true)("k", v)
      )
    if (justReturn) valueKeys
    else map.put(key, valueKeys); map
```



EXTENSIBILITY

```
val p = Person("Yosef", "Bloggs", List("Joseph", "Joe"))
p.writeToMap   // 175 ns!

val map = mutable.Map[String, Any]()
map.put("firstName", "Yosef")
map.put("lastName", "Bloggs")
map.put("addresses", List("Joseph", "Joe"))
map   // 178 ns!
```

```
trait WriteToMap[T]:
  def writeToMap(mapOrReturn: mutable.Map[String, Any], justReturn: Boolean)(key: String, value: T): Any


given writeList[T](using wtm: WriteToMap[T]): WriteToMap[List[T]] with
  def writeToMap(map: mutable.Map[String, Any], justReturn: Boolean)(key: String, values: List[T]): Any =
    val valueKeys =
      values.map(v =>
        wtm.writeToMap(null, true)("k", v)
      )
    if (justReturn) valueKeys
    else map.put(key, valueKeys); map
```



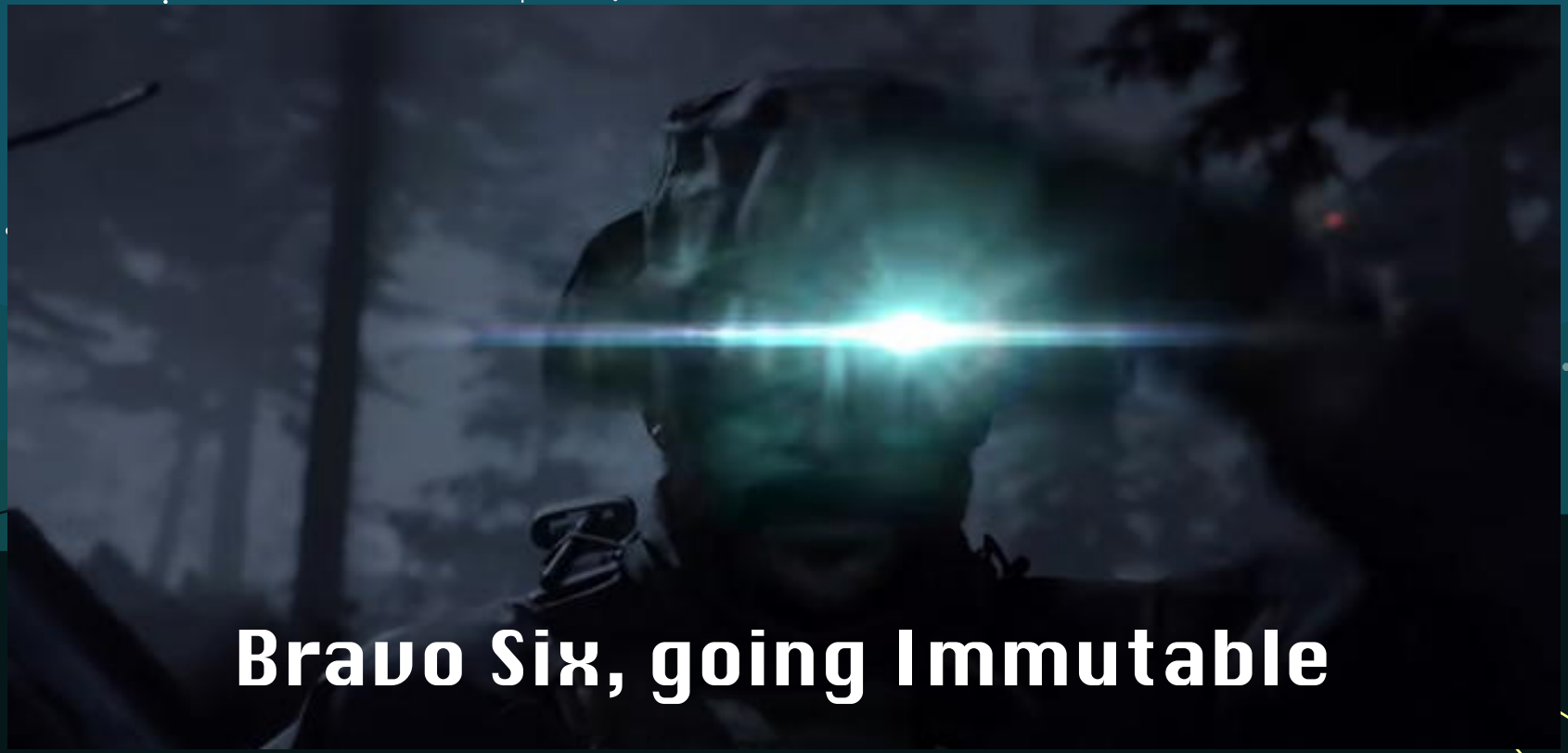
EXTENSIBILITY

```
case class Person(firstName: String, lastName: String, nicknames: List[Address])
val p =
  Person("Yosef", "Bloggs", List(
    Address("123 Place", 11122),    // 347 ns
    Address("456 Ave", 11122))
  )
```

```
val map = mutable.Map[String, Any]()
map.put("firstName", "Yosef"); map.put("lastName", "Bloggs")
val map1 = mutable.Map[String, Any]()
map1.put("street", "123 Place"); map1.put("zip", 11122)
val map2 = mutable.Map[String, Any]()
map1.put("street", "456 Ave"); map1.put("zip", 11122)
map.put("addresses", List(map1, map2))
map
      // 375 ns
```

```
trait WriteToMap[T]:
  def writeToMap(mapOrReturn: mutable.Map[String, Any], justReturn: Boolean)(key: String, value: T): Any
```

EXTENSIBILITY



EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)
      val writtenValue = summonWriter[tpe].writeToMap(value).value
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]



inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)
      val writtenValue = summonWriter[tpe].writeToMap(value).value
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]



inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)                    // next := Map[String, Any](...)
      val writtenValue = summonWriter[tpe].writeToMap(value).value
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]



inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1).              // next := Map[String, Any](...)
      val writtenValue = summonWriter[tpe].writeToMap(value).value       // writtenValue := writeStr("Joe")
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]



inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1).              // next := Map[String, Any](...)
      val writtenValue = summonWriter[tpe].writeToMap(value).value       // writtenValue := writeStr("Joe")
      recurse[names, types](element)(index + 1) + (key -> writtenValue)  // "firstName" -> writtenValue
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]



inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
given writeInt: WriteToMap[Int] with
  def writeToMap(value: Int) = WriteOutput.Leaf(value)

given writeString: WriteToMap[String] with
  def writeToMap(value: String) = WriteOutput.Leaf(value)

given writeList[T](using WriteToMap[T]): WriteToMap[List[T]] with
  def writeToMap(value: List[T]) = WriteOutput.Leaf(value)
```

```scala
object WriteToMapOps {
  extension [T <: Product](value: T)(using wtm: WriteToMap[T])
    def writeToMap: Map[String, Any] =
      wtm.writeToMap(value) match
        case Node(map) => map
}
```

EXTENSIBILITY

```scala
case class Person(firstName: String, lastName: String, age: Int)
val p = Person("Joe", "Bloggs", 123)
p.writeToMap  // 86ns Immutable vs 96ns with Mutable!!
```

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])
val p = Person("Yosef", "Bloggs", List("Joseph", "Joe"))
p.writeToMap
// 94ns Immutable vs 175ns with Mutable!!
```

```scala
case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, addresses: List[Address])
val p =
  Person("Yosef", "Bloggs", List(
    Address("123 Place", 11122),
    Address("456 Ave", 11122))
  )
p.writeToMap // 89ns Immutable vs 347ns with Mutable!!
```

EXTENSIBILITY

HOW IS THIS EVEN POSSIBLE!?

```
// 86ns Immutable vs 96ns with Mutable!!

// 94ns Immutable vs 175ns with Mutable!!

// 89ns Immutable vs 347ns with Mutable!!
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)
      val writtenValue = summonWriter[tpe].writeToMap(value).value
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]
```

FROM THIS

```scala
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  def value: Any
  case Leaf(value: Any) extends WriteOutput
  case Node(map: Map[String, Any]) extends WriteOutput

trait WriteToMap[T]:
  def writeToMap(value: T): WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)
      val writtenValue =
        summonWriter[tpe].writeToMap(value) match
          case Node(value) => value
          case Leaf(value) => value
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]

inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

... TO THIS

EXTENSIBILITY

```scala
case class Person(firstName: String, lastName: String, age: Int)
val p = Person("Joe", "Bloggs", 123)
p.writeToMap // 86 -> 73ns Immutable vs 96ns with Mutable!!
```

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])
val p = Person("Yosef", "Bloggs", List("Joseph", "Joe"))
p.writeToMap // 94 -> 74ns Immutable vs 175ns with Mutable!!
```

```scala
case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, addresses: List[Address])
val p =
  Person("Yosef", "Bloggs", List(
    Address("123 Place", 11122),
    Address("456 Ave", 11122))
  )
p.writeToMap // 89 -> 77ns Immutable vs 347ns with Mutable!!
```

EXTENSIBILITY

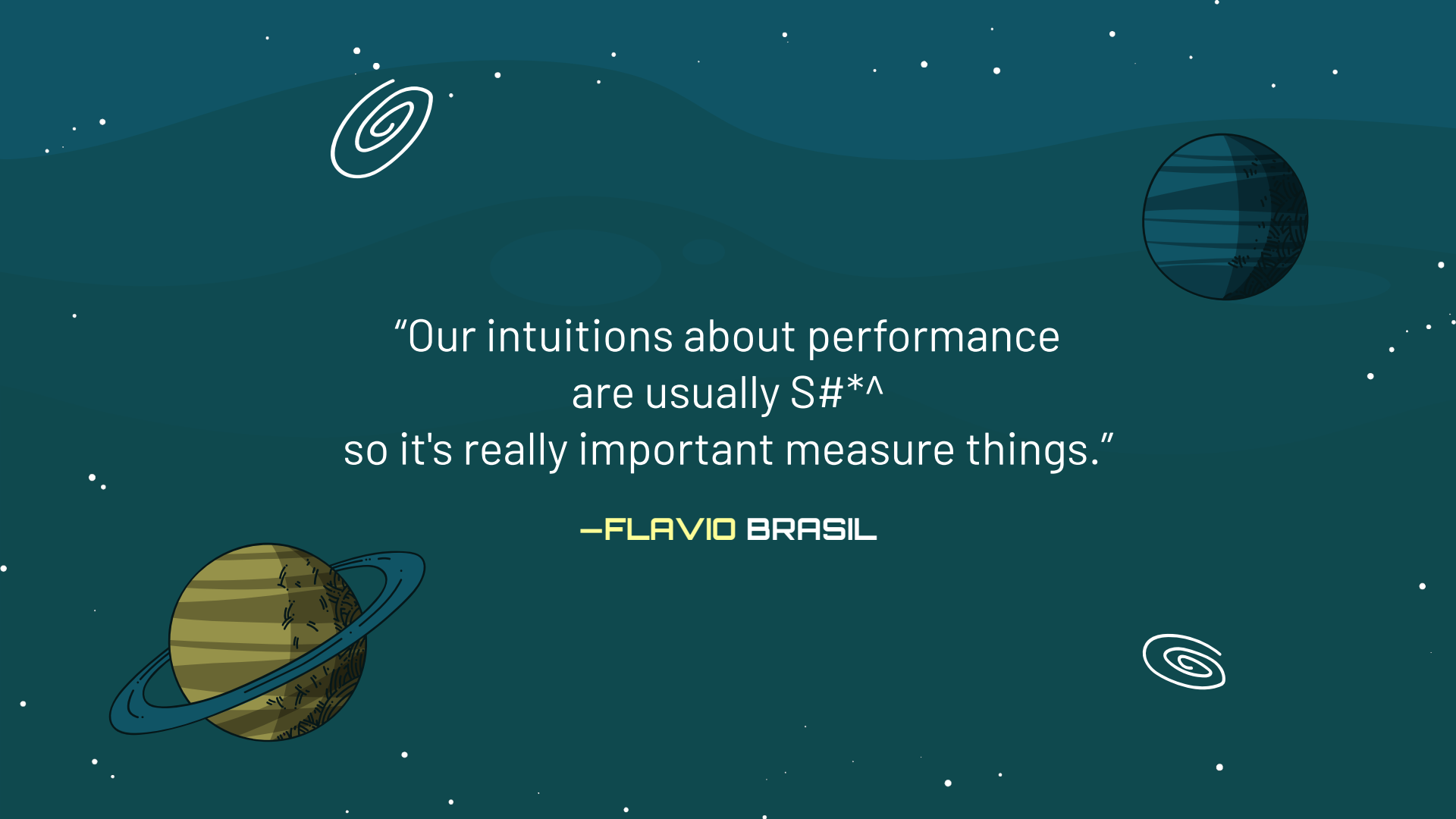

```
summonWriter[tpe].writeToMap(value).value
```

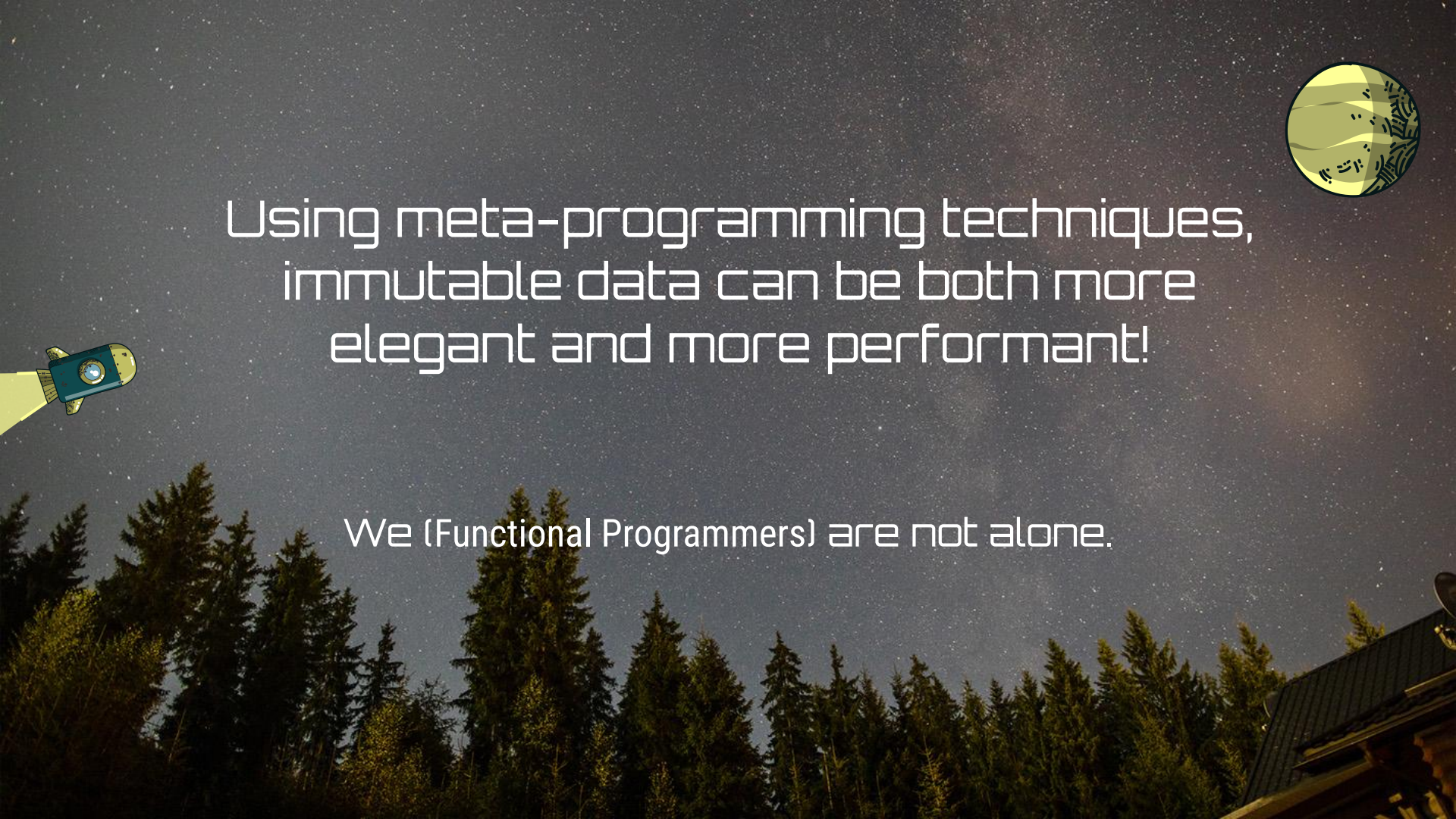
94ns

74ns

MORE EFFICIENT???

```
val writtenValue =
  summonWriter[tpe].writeToMap(value) match
    case Node(value) => value
    case Leaf(value) => value
```


“Our intuitions about performance
are usually S#*^
so it's really important measure things.”
—FLAVIO BRASIL


Using meta-programming techniques, immutable data can be both more elegant and more performant!
We (Functional Programmers) are not alone.

EXTENSIBILITY

```scala
enum WriteOutput:
  case Leaf(value: Any) extends WriteOutput
  case Node(keyValues: Map[String, Any]) extends WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)
      val writtenValue = summonWriter[tpe].writeToMap(value).value
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]

inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  case Leaf(value: Any) extends WriteOutput
  case Node(keyValues: Map[String, WriteOutput]) extends WriteOutput
  case Arr(list: List[WriteOutput]) extends WriteOutput
```

```scala
inline def recurse[Names <: Tuple, Types <: Tuple](element: Product)(index: Int): Map[String, Any] =
  inline erasedValue[(Names, Types)] match
    case (_: (name *: names), _: (tpe *: types)) =>
      val key = constValue[name].toString
      val value = element.productElement(index).asInstanceOf[tpe]
      val next = recurse[names, types](element)(index + 1)
      val writtenValue = summonWriter[tpe].writeToMap(value)
      recurse[names, types](element)(index + 1) + (key -> writtenValue)
    case (_: EmptyTuple, _) =>
      Map.empty[String, Any]
```

```scala
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  case Leaf(value: Any) extends WriteOutput
  case Node(keyValues: Map[String, WriteOutput]) extends WriteOutput
  case Arr(list: List[WriteOutput]) extends WriteOutput
```

```scala
given writeInt: WriteToMap[Int] with
  def writeToMap(value: Int) = WriteOutput.Leaf(value)

given writeString: WriteToMap[String] with
  def writeToMap(value: String) = WriteOutput.Leaf(value)

given writeList[T](using wtm: WriteToMap[T]): WriteToMap[List[T]] with
  def writeToMap(value: List[T]) = WriteOutput.Arr(value.map(wtm.writeToMap(_)))
```

EXTENSIBILITY

```scala
enum WriteOutput:
  case Leaf(value: Any) extends WriteOutput
  case Node(keyValues: Map[String, WriteOutput]) extends WriteOutput
  case Arr(list: List[WriteOutput]) extends WriteOutput
```

```scala
def encode(wo: WriteOutput): String =
  wo match
    case Leaf(value) =>
      s""""${value}""""
    case Node(keyValues) =>
      keyValues
        .map((k, v) => (k, encode(v)))
        .map((k, v) => s"${k}:${v}")
        .mkString("{", ",", "}")
    case Arr(list) =>
      list
        .map(encode)
        .mkString("[",",","]")
```

```scala
extension [T](value: T)(using wtm: WriteToMap[T])
  def toJson: String =
    encode(wtm.writeToMap(value))
```

```scala
case class Person(firstName: String, lastName: String, age: Int)
val p = Person("Joe", "Bloggs", 123)
p.toJson
```

```
{age:"123",lastName:"Bloggs",firstName:"Joe"}
```

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])
val p = Person("Yosef", "Bloggs", List("Joseph", "Joe"))
p.toJson
```

```
{nicknames:["Joseph","Joe"],lastName:"Bloggs",firstName:"Yosef"}
```

```scala
case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, addresses: List[Address])
val p =
  Person("Yosef", "Bloggs", List(
    Address("123 Place", 11122),
    Address("456 Ave", 11122))
  )
p.toJson
```

```
{addresses:[{zip:"11122",street:"123 Place"},{zip:"11122",street:"456 Ave"}],
 lastName:"Bloggs",firstName:"Yosef"}
```

EXTENSIBILITY

```scala
enum WriteOutput:
  case Leaf(value: Any) extends WriteOutput
  case Node(keyValues: Map[String, WriteOutput]) extends WriteOutput
  case Arr(list: List[WriteOutput]) extends WriteOutput
```

```scala
def encode(wo: WriteOutput): String =
  wo match
    case Leaf(value) =>
      s""""${value}""""
    case Node(keyValues) =>
      keyValues
        .map((k, v) => (k, encode(v)))
        .map((k, v) => s"${k}:${v}")
        .mkString("{", ",", "}")
    case Arr(list) =>
      list
        .map(encode)
        .mkString("[",",","]")
```

```scala
extension [T](value: T)(using wtm: WriteToMap[T])
  def toJson: String =
    encode(wtm.writeToMap(value))
```

```scala
case class Person(firstName: String, lastName: String, age: Int)
val p = Person("Joe", "Bloggs", 123)
p.toJson
```

```
{age:"123",lastName:"Bloggs",firstName:"Joe"}
```

```scala
case class Person(firstName: String, lastName: String, nicknames: List[String])
val p = Person("Yosef", "Bloggs", List("Joseph", "Joe"))
p.toJson
```

```
{nicknames:["Joseph","Joe"],lastName:"Bloggs",firstName:"Yosef"}
```

```scala
case class Address(street: String, zip: Int)
case class Person(firstName: String, lastName: String, addresses: List[Address])
val p =
  Person("Yosef", "Bloggs", List(
    Address("123 Place", 11122),
    Address("456 Ave", 11122))
  )
p.toJson
```

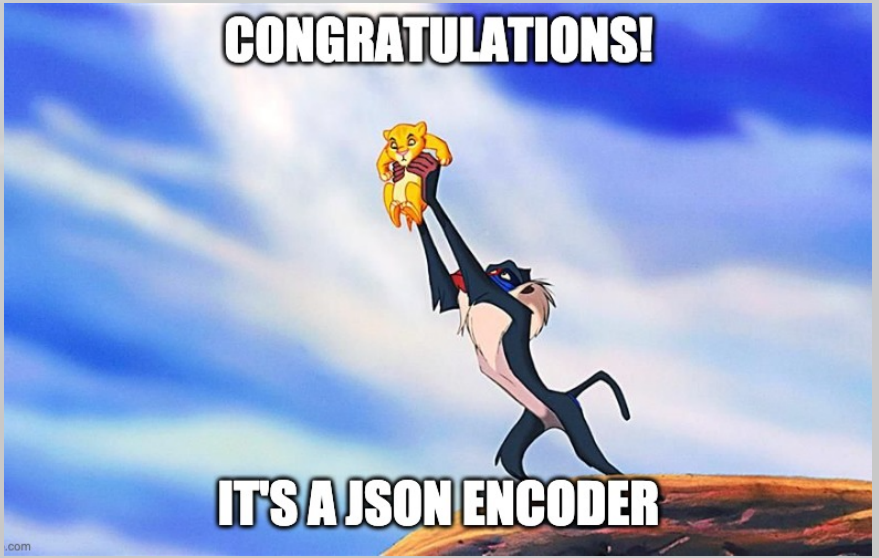
```
{addresses:[{zip:"11122",street:"123 Place"},{zip:"11122",street:"456 Ave"}],
 lastName:"Bloggs",firstName:"Yosef"}
```

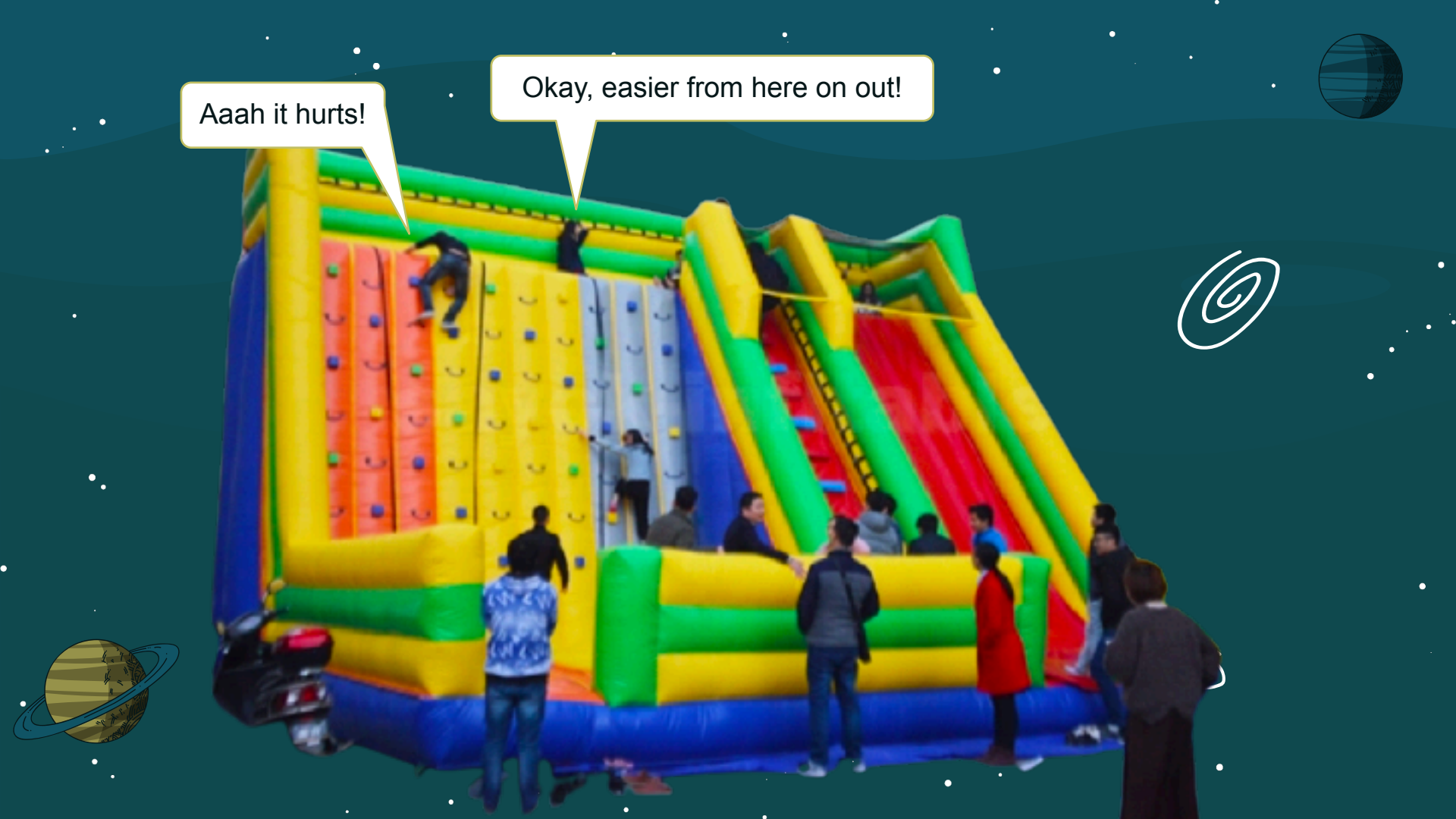
EXTENSIBILITY

```scala
enum WriteOutput:
  case Leaf(value: Any) extends WriteOutput
  case Node(keyValues: Map[String, WriteOutput]) extends WriteOutput
  case Arr(list: List[WriteOutput]) extends WriteOutput
```

```scala
def encode(wo: WriteOutput): String =
  wo match
    case Leaf(value) =>
      s"""${value}"""
    case Node(keyValues) =>
      keyValues
        .map((k, v) => (k, encode(v)))
        .map((k, v) => s"${k}:${v}")
        .mkString("{", ",", "}")
    case Arr(list) =>
      list
        .map(encode)
        .mkString("[",",","]")
```

```scala
extension [T](value: T)(using wtm: WriteToMap[T])
  def toJson: String =
    encode(wtm.writeToMap(value))
```




Aaah it hurts!
Okay, easier from here on out!

COPRODUCT POLYMORPHISM

```scala
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
```

COPRODUCT POLYMORPHISM

```scala
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput = recurseSum[sumMir.MirroredElemTypes, T](value)
```

```scala
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      if (element.isInstanceOf[tpe])
        summonWriter[tpe].writeToMap(element.asInstanceOf[tpe])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```
enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name

case class Person(name: Name, age: Int)
val p = Person(Name.Simple("Joe", "Bloggs"), 123)
p.writeToMap
```

```
Map("age" -> 123, "name" -> Map("last" -> "Bloggs", "first" -> "Joe"))
```

```
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      if (element.isInstanceOf[tpe])
        summonWriter[tpe].writeToMap(element.asInstanceOf[tpe])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs"), 123
```

```scala
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput = recurseSum[sumMir.MirroredElemTypes, T](value)
```

```scala
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      if (element.isInstanceOf[tpe])
        summonWriter[tpe].writeToMap(element.asInstanceOf[tpe])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")
```

```scala
inline given derived[T]: WriteToMap[T] =
  inline summonInline[Mirror.ProductOf[T]] match
    case proMir: Mirror.ProductOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          Node(recurse[proMir.MirroredElemLabels, proMir.MirroredElemTypes](value.asInstanceOf[Product])(0))
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput = recurseSum[sumMir.MirroredElemTypes, T](value)
```

```scala
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      if (element.isInstanceOf[tpe])
        summonWriter[tpe].writeToMap(element.asInstanceOf[tpe])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")

enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput = recurseSum[(Name.Simple, NameTitle), T](value)
```

```scala
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      if (element.isInstanceOf[tpe])
        summonWriter[tpe].writeToMap(element.asInstanceOf[tpe])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")
```

```scala
case sumMir: Mirror.SumOf[T] =>
  new WriteToMap[T]:
    def writeToMap(value: T): WriteOutput = recurseSum[(Name.Simple, NameTitle), T](value)
```

```scala
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (tpe *: types) =>
      if (element.isInstanceOf[tpe])
        summonWriter[tpe].writeToMap(element.asInstanceOf[tpe])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")

enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput = recurseSum[(Name.Simple, NameTitle), T](value)
```

```scala
inline def recurseSum[Types <: Tuple, T](element: T): WriteOutput =
  inline erasedValue[Types] match
    case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
      if (element.isInstanceOf[Name.Simple])
        summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
      else
        recurseSum[types, T](element)
    case _: EmptyTuple =>
      throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")

enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else
              recurseSum[types, T](element)
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")
```

```scala
enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else
              recurseSum[types, T](element)
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")
```

```scala
enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else
              recurseSum[(Name.Title, EmptyTuple), T](element)
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")

enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else
              case _: (Name.Title *: EmptyTuple)) =>
                if (element.isInstanceOf[Name.Title])
                  summonWriter[Name.Title].writeToMap(element.asInstanceOf[Name.Title])
                else
                  recurseSum[types, T](element)
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")

enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else
              case _: (Name.Title *: EmptyTuple)) =>
                if (element.isInstanceOf[Name.Title])
                  summonWriter[Name.Title].writeToMap(element.asInstanceOf[Name.Title])
                else
                  recurseSum[EmptyTuple, T](element)
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")

enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
    case sumMir: Mirror.SumOf[T] =>
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =
          case _: (Name.Simple *:(Name.Title, EmptyTuple)) =>
            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else
              case _: (Name.Title *: EmptyTuple)) =>
                if (element.isInstanceOf[Name.Title])
                  summonWriter[Name.Title].writeToMap(element.asInstanceOf[Name.Title])
                else
                  case _: EmptyTuple =>
                    throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```
val n: Name = Name.Simple("Joe", "Bloggs")
```

```
enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```
      new WriteToMap[T]:
        def writeToMap(value: T): WriteOutput =

            if (element.isInstanceOf[Name.Simple])
              summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
            else

                if (element.isInstanceOf[Name.Title])
                  summonWriter[Name.Title].writeToMap(element.asInstanceOf[Name.Title])
                else

                    throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")
```

```scala
enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
given WriteToMap[Name] = WriteToMap.derived
```

```scala
       new WriteToMap[T]:
         def writeToMap(value: T): WriteOutput =

              if (element.isInstanceOf[Name.Simple])
                summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
              else

                   if (element.isInstanceOf[Name.Title])
                     summonWriter[Name.Title].writeToMap(element.asInstanceOf[Name.Title])
                   else

                        throw new IllegalArgumentException(s"Invalid coproduct type")
```

COPRODUCT POLYMORPHISM

```scala
val n: Name = Name.Simple("Joe", "Bloggs")
```

```scala
enum Name:
  case Simple(first: String, last: String) extends Name
  case Title(first: String, middle: String, last: String) extends Name
```

```scala
given WriteToMap[Name] =
     new WriteToMap[T]:
       def writeToMap(value: T): WriteOutput =
          if (element.isInstanceOf[Name.Simple])
            summonWriter[Name.Simple].writeToMap(element.asInstanceOf[Name.Simple])
          else
             if (element.isInstanceOf[Name.Title])
               summonWriter[Name.Title].writeToMap(element.asInstanceOf[Name.Title])
             else
                throw new IllegalArgumentException(s"Invalid coproduct type")
```

```
> n.writeToMap
Map("last" -> "Bloggs", "first" -> "Joe")
```

04

Come let us quill together

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

quilling Together

```scala
class RunQuery(uri: String) {
  inline def apply[T]: String = ...
    // Generate the Query
    // Run it on the DB and tell me if it works
    // Return the Generated Query
    // All while you're compiling...
}
```

```scala
class Database(uri: String) {
  def runQuery(str: String): QueryStatus
}

enum QueryStatus:
  case Success
  case Failure
```

quilling Together

```scala
class RunQuery(uri: String) {
  inline def apply[T]: String = ${ RunQueryMacro[T]('uri) }
}



object RunQueryMacro {
  def apply[T: Type](uri: Expr[String])(using qctx: Quotes): Expr[String] =

    // Generate the Query
    // Run it on the DB and tell me if it works
    // Return the Generated Query
    // All while you're compiling...
}
```

quilling Together

```scala
class RunQuery(uri: String) {
  inline def apply[T]: String = ${ RunQueryMacro[T]('uri) }
}



object RunQueryMacro {
  def apply[T: Type](uri: Expr[String])(using qctx: Quotes): Expr[String] =

    // Generate the Query
    // Run it on the DB and tell me if it works
    // Return the Generated Query
    // All while you're compiling...
}
```

quilling Together

```scala
class RunQuery(uri: String) {
  inline def apply[T]: String = ${ RunQueryMacro[T]('uri) }
}


object RunQueryMacro {
  def apply[T: Type](uri: Expr[String])(using qctx: Quotes): Expr[String] =

    // Generate the Query
    // Run it on the DB and tell me if it works
    // Return the Generated Query
    // All while you're compiling...
}
```

quilling Together

```scala
class RunQuery(uri: String) {
  inline def apply[T]: String = ${ RunQueryMacro[T]('uri) }
}


object RunQueryMacro {
  def apply[T: Type](db: Expr[String])(using qctx: Quotes): Expr[String] = {

      // Generate the Query
      val q: String = ???

      // Run it on the DB and tell me if it works
      db match
        case Expr(str: String) =>
          new Database(str).runQuery(q) match
            case QueryStatus.Success => println("Query Succeeded")
            case QueryStatus.Failure => println("Query Failed")


      // Return the Generated Query
      Expr(q)

      // (Check!) All while you're compiling...
  }
}
```

quilling Together

```scala
class RunQuery(uri: String) {
  inline def apply[T]: String = ${ RunQueryMacro[T]('uri) }
}


object RunQueryMacro {
  def apply[T: Type](db: Expr[String])(using qctx: Quotes): Expr[String] = {

      // Generate the Query
      val q: String = generateQuery[T]

      // Run it on the DB and tell me if it works
      db match
        case Expr(str: String) =>
          new Database(str).runQuery(q) match
            case QueryStatus.Success => println("Query Succeeded")
            case QueryStatus.Failure => println("Query Failed")


      // Return the Generated Query
      Expr(q)

      // (Check!) All while you're compiling...
  }
}
```

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

```
def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  // Get all the possible types of T
  // Get all the possible fields in them
  val allFields: List[String] = ???
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}
```

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

```
def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  // Get all the possible types of T (Person, Robot, Yeti)
  // Get all the possible fields in them (name, lastName, cpu, furColor)
  val allFields: List[String] = ???
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}
```

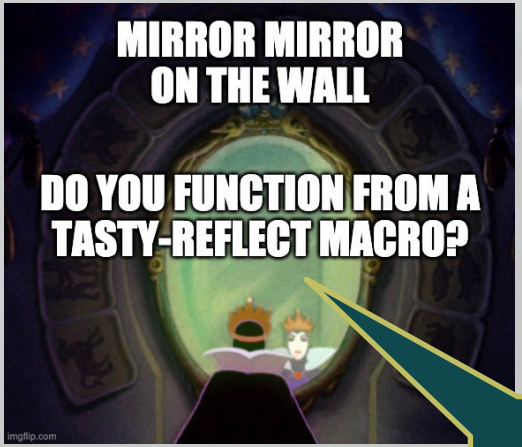
quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```



Mirror.Sum

(Person, Robot, Yeti)
(name, lastName, cpu, furColor)

Mirror.Product

I do if you quote me right...

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

```
def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  // Get all the possible types of T (Person, Robot, Yeti)
  // Get all the possible fields in them (name, lastName, cpu, furColor)
  val allFields: List[String] = ???
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}
```

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

```
def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  val allFields: List[String] = GatherFields.base[T]
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}
```

```
def nameOf[T: Type](using qctx: Quotes): String =
  import qctx.reflect._
  TypeRepr.of[T].typeSymbol.name
```

By The Way...

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

```
def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  val allFields: List[String] = GatherFields.base[T]
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}

object GatherFields {
  def base[T: Type](using qctx: Quotes): List[String] = ...
}
```

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] = ...

}
```

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])

}
```

Instead of a Mirror, summon an expression of a mirror i.e. Expr[Mirror.ProductOf[T]] instead of Mirror.ProductOf[T]

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])

}
```

Then match the mirror using the expression-matching syntax. types can be pulled out of this thing via Type.of[elementLables]

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])

}
```

Instead of doing that,
we pass the elementLabels
directly into a function that
will recursively process them...

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Instead of doing that,
we pass the elementLabels
directly into a function that
will recursively process them...

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Now we pull out elementLables by doing Type.of[Names] which is really: Type.of[elementLabels']

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredEl
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Recurse over the tuple ("name", "lastName") via the *: operator like we saw before:

First:
(name *: (lastName, EmptyTuple))

Then:
(lastName, EmptyTuple)

Then:
(EmptyTuple)

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[Person]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[("name", "lastName")]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredEl
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[("name", "lastName")](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '["name" *: ("lastName", EmptyTuple)] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Recurse over the tuple ("name", "lastName") via the *: operator like we saw before:

First:
(name *: (lastName, EmptyTuple))

Then:
(lastName, EmptyTuple)

Then:
(EmptyTuple)

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[Person]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[("name", "lastName")]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredEl
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[("name", "lastName")](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '["name" *: ("lastName", EmptyTuple)] => constValue["name"] +: collectLabels[("lastName", EmptyTup)]
        case '[EmptyTuple] => Nil

}
```

Recurse over the tuple ("name", "lastName") via the *: operator like we saw before:

First:
(name *: (lastName, EmptyTuple))

Then:
(lastName, EmptyTuple)

Then:
(EmptyTuple)

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[Person]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[("name", "lastName")]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredEl
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[("lastName")](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '["lastName" *: EmptyTuple] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Recurse over the tuple ("name", "lastName") via the *: operator like we saw before:

First:
(name *: (lastName, EmptyTuple))

Then:
(lastName, EmptyTuple)

Then:
(EmptyTuple)

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[Person]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[("name", "lastName")]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredEl
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[("lastName")](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '["lastName" *: EmptyTuple] => constValue["lastName"] +: collectLabels[EmptyTuple]
        case '[EmptyTuple] => Nil

}
```

Recurse over the tuple ("name", "lastName") via the *: operator like we saw before:

First:
(name *: (lastName, EmptyTuple))

Then:
(lastName, EmptyTuple)

Then:
(EmptyTuple)

quilling Together

```
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[Person]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              collectLabels[(("name", "lastName"))]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[EmptyTuple](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Now we pull out elementLables by doing Type.of[Names] which is really: Type.of[elementLabels']

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil
```

```scala
def constValue[T: Type](using qctx: Quotes): String =
  import qctx.reflect._
  TypeRepr.of[T] match
    case ConstantType(StringConstant(value)) => value.toString
```

By The Way...

```scala
}
```

quilling Together

```scala
object GatherFields {

    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil

}
```

Go through all the coproduct-types that T can be i.e. (Person, Robot, Yeti)

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil



    def gatherTypeFields[Types: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Types] match
        case '[tpe *: types] => base[tpe] ++ gatherTypeFields[types]
        case '[EmptyTuple] => Nil
}
```

Go through all the coproduct-types that T can be i.e. (Person, Robot, Yeti)

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type Mirro
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredEl
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil



    def gatherTypeFields[Types: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Types] match
        case '[tpe *: types] => base[tpe] ++ gatherTypeFields[types]
        case '[EmptyTuple] => Nil
}
```

Recurse over the tuple
(Person, Robot, Yeti) via the *:
operator like we saw before:

First:
(Person *:
 (Robot *: (Yeti, EmptyTuple))

Then:
(Robot *: (Yeti, EmptyTuple))

Then:
(Yeti, EmptyTuple)

Then:
(EmptyTuple)

quilling Together

```
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Expr.summon[Mirror.Of[T]] match
        case Some(mir) =>
          mir match
            case '{ $m: Mirror.ProductOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              collectLabels[elementLabels]
            case '{ $m: Mirror.SumOf[T] { type MirroredElemLabels = elementLabels; type MirroredElemTypes = elementTypes }} =>
              gatherTypeFields[elementTypes]
        case _ =>
          report.throwError("Whoops, can't summon a mirror for: " + Type.of[T])


    def collectLabels[Names: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Names] match
        case '[name *: names] => constValue[name] +: collectLabels[names]
        case '[EmptyTuple] => Nil



    def gatherTypeFields[Types: Type](using qctx: Quotes): List[String] =
      import qctx.reflect._
      Type.of[Types] match
        case '[tpe *: types] => base[tpe] ++ gatherTypeFields[types]
        case '[EmptyTuple] => Nil
}
```

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] = ...

}
```

quilling Together

```scala
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] = ...
}

def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  val allFields: List[String] = GatherFields.base[T]
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}

object RunQueryMacro {
  def apply[T: Type](db: Expr[String])(using qctx: Quotes): Expr[String] = {

      // Generate the Query
      val q: String = generateQuery[T]

      // Run it on the DB and tell me if it works
      db match
        case Expr(str: String) =>
          new Database(str).runQuery(q) match
            case QueryStatus.Success => println("Query Succeeded")
            case QueryStatus.Failure => println("Query Failed")

      // Return the Generated Query
      Expr(q)
  }
}
```

quilling Together

```
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] = ...
}

def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  val allFields: List[String] = GatherFields.base[T]
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}

object RunQueryMacro {
  def apply[T: Type](db: Expr[String])(using qctx: Quotes): Expr[String] = {


      val q: String = generateQuery[T]


      db match
        case Expr(str: String) =>
          new Database(str).runQuery(q) match
            case QueryStatus.Success => println("Query Succeeded")
            case QueryStatus.Failure => println("Query Failed")


      Expr(q)
  }
}
```

quilling Together

```
object GatherFields {
    def base[T: Type](using qctx: Quotes): List[String] = ...
}

def generateQuery[T: Type](using qctx: Quotes): String = {
  import qctx.reflect._
  val allFields: List[String] = GatherFields.base[T]
  s"SELECT ${allFields.mkString(", ")} FROM ${nameOf[T]}"
}

object RunQueryMacro {
  def apply[T: Type](db: Expr[String])(using qctx: Quotes): Expr[String] = {
      val q: String = generateQuery[T]
      db match
        case Expr(str: String) =>
          new Database(str).runQuery(q) match
            case QueryStatus.Success => println("Query Succeeded")
            case QueryStatus.Failure => println("Query Failed")

      Expr(q)
  }
}
object RunQuery:
  inline def apply[T](inline uri: String): String = ${ RunQueryMacro[T]('uri) }

> Query Is: SELECT name, lastName, name, cpu, name, furColor FROM Customer
> Query Succeeded
```

quilling Together

Customer Table

type	name	lastName	cpu	furColor
Person	Joe	Bloggs		
Robot	IG-88		i486	
Yeti	Rhaarg			white

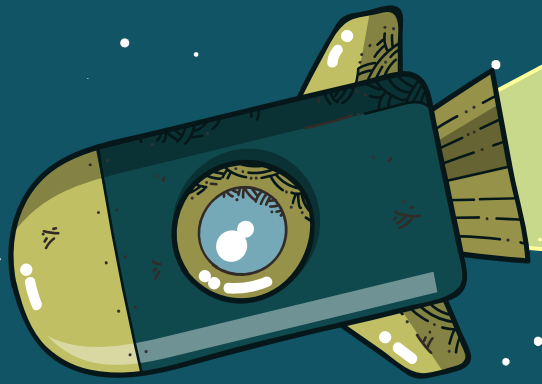
Data Model

```
enum Customer:
  case Person(name: String, lastName: String)
  case Robot(name: String, cpu: String)
  case Yeti(name: String, furColor: String)
```

```
object RunQuery:
  inline def apply[T](inline uri: String): String = ${ RunQueryMacro[T]('uri) }
```

```
RunQuery.apply[Customer]("uri/to/db")
```

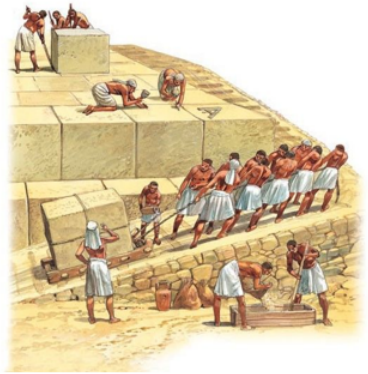
```
> Query Is: SELECT name, lastName, name, cpu, name, furColor FROM Customer
> Query Succeeded
```



05 CONCLUSION

In case you're still here...

Compare with similar items



	Generic Derivation	Handwritten Code	Java Reflection
	Add to Cart	Add to Cart	Add to Cart #1 Best Seller
Customer Rating	★★★★½ (586)	★☆☆☆☆ (48245671)	★★★★☆ (7492)
Price (nanoseconds)	$78ns	$78ns	$1342ns
Sold By	Scala 3	A Disgruntled Dev. Near You!	Oracle Inc.
Technology	Compile-Time	Build-time, Freemarker, etc…	Runtime

@This

https://github.com/deusaquilus/derivation_examples

https://getquill.io/

Dotty 

https://github.com/deusaquilus/dotty_test

Scala 3 

https://dotty.epfl.ch/docs/reference/contextual/derivation.html