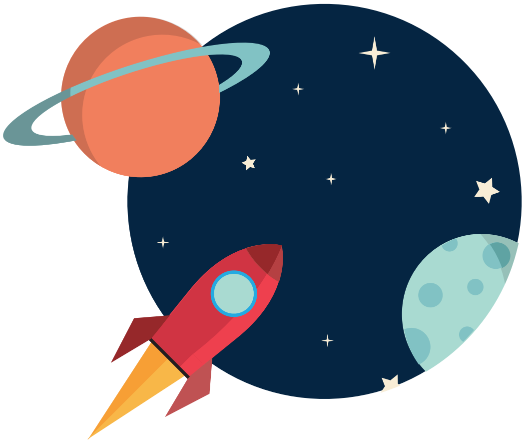
The Type Astronaut's Guide to Shapeless



August 2016

Dave Gurnell

underscore

The Type Astronaut's Guide to Shapeless

August 2016

Published by Underscore Consulting LLP, Brighton, UK.

Disclaimer: *Every precaution was taken in the preparation of this book. However,* ***the author and Underscore Consulting LLP assume no responsibility for errors or omissions, or for damages*** *that may result from the use of information (including program listings) contained herein.*

Contents

Chapter 1

Introduction

This book is a guide to using shapeless, a library for *generic programming* in Scala. Shapeless is large library, so rather than cover everything it has to offer, we will concentrate on a few compelling use cases and use them to build a picture of the tools and patterns available.

Before we start, let's talk about what generic programming is and why shapeless is so exciting to Scala developers.

1.1 What is generic programming?

As Scala developers we are used to types. Types are helpful because they are specific: they show us how different pieces of code fit together, help us prevent bugs, and guide us toward solutions when we code.

Sometimes, however, types are *too* specific. There are situations where we want to exploit similarities between types to avoid repetition and boilerplate. For example, consider the following definitions:

```
case class Employee(name: String, number: Int, manager: Boolean)

case class IceCream(name: String, numCherries: Int, inCone: Boolean)
```

These two case classes represent different kinds of data but they have clear similarities: they both contain three fields of the same types. Suppose we want to implement a generic operation such as serializing to a CSV file. Despite the similarity between the two types, we have to write two separate serialization methods:

```
def employeeCsv(e: Employee): List[String] =
  List(e.name, e.number.toString, e.manager.toString)

def iceCreamCsv(c: IceCream): List[String] =
  List(c.name, c.numCherries.toString, c.inCone.toString)
```

Generic programming is about overcoming differences like these. Shapeless makes it convenient to convert specific types into generic ones that we can manipulate with common code.

For example, we can use the code below to convert employees and ice creams to values of the same type. Don't worry if you don't follow this example yet: we'll get to grips with the various concepts later on:

```
import shapeless._

val genericEmployee = Generic[Employee].to(Employee("Dave", 123, false))
// genericEmployee: shapeless.::[String,shapeless.::[Int,shapeless.::[
     Boolean,shapeless.HNil]]] = Dave :: 123 :: false :: HNil

val genericIceCream = Generic[IceCream].to(IceCream("Sundae", 1, false))
// genericIceCream: shapeless.::[String,shapeless.::[Int,shapeless.::[
     Boolean,shapeless.HNil]]] = Sundae :: 1 :: false :: HNil
```

Now that both sets of data are the same type, we can serialize them with the same function:

```
def genericCsv(gen: String :: Int :: Boolean :: HNil): List[String] =
  List(gen(0), gen(1).toString, gen(2).toString)
// genericCsv: (gen: shapeless.::[String,shapeless.::[Int,shapeless.::[
     Boolean,shapeless.HNil]]])List[String]

genericCsv(genericEmployee)
```

```
// res2: List[String] = List(Dave, 123, false)

genericCsv(genericIceCream)
// res3: List[String] = List(Sundae, 1, false)
```

This example is basic but it hints at the essence of generic programming. We reformulate problems so we can solve them use generic building blocks, and write small kernels of code that work with a wide variety of types. Generic programming with shapeless allows us to eliminate huge amounts of boilerplate, making Scala applications easier to read, write, and maintain.

Does that sound compelling? Thought so. Let's jump in!

1.2 About this book

This book is divided into chapters that introduce parts of the shapeless machinery.

In Chapter **??** we introduce *generic representations* and shapeless' `Generic` type class, which can produce a generic encoding for any case class or sealed trait. The main use case will be something basic: converting one type of data to another.

In Chapter 3 we use `Generic` to derive instances of a custom type class. We use CSV encoding as an example, but these techniques can be extended to many situations. We will introduce shapeless' `Lazy` type, which lets us handle recursive data like lists and trees.

In Chapter 4 we cover some more theory: dependent types, dependently typed functions, and type level programming. We will introduce the programming patterns we need to generalise from deriving type class instances to doing more advanced things in shapeless.

In Chapter 5 we introduce `LabelledGeneric`, a variant of `Generic` that exposes field and type names as part of its generic representations. We also introduce some new theory: literal types, singleton types, phantom types, and type tagging. In our examples we upgrade from CSV encoding to writing JSON encoders that preserve field and type names in their output.

In Chapter **??** we open the shapeless toolbox, introducing a variety of operations on generic representations including mapping and filtering. We also introduce *polymorphic functions* that can have different output types depending on their parameter types.

TODO: Function and tuple interop

TODO: Counting with types

TODO: Polymorphic functions

Chapter 2

Algebraic data types and generic representations

The main idea behind generic programming is to solve problems for a wide variety of types by writing a small amount of generic code. Shapeless provides two sets of tools to this end:

1. a set of generic data types that can be inspected, traversed, and manipulated at the type level;

2. automatic mapping between *algebraic data types (ADTs)* (encoded in Scala as case classes and sealed traits) and these generic representations.

In this chapter we will start by recapping on what algebraic data types are and why they might be familiar to Scala developers. Then we will look at the data types shapeless uses as generic representations and discuss how they map on to concrete ADTs. Finally, we will introduce a type class called `Generic` that provides automatic mapping back and forth between ADTs and generic representations. We will finish with some simple examples using `Generic` to convert values from one type to another.

2.1 Recap: algebraic data types

Algebraic data types (*ADTs*[1]) are a functional programming concept with a fancy name but a very simple meaning. They are simply an idiomatic way of representing data using "ands" and "ors". For example:

- a shape is a rectangle **or** a circle
- a rectangle has a width **and** a height
- a circle has a radius

In ADT terminology, "and" types such as rectangle and circle are called *products*, and "or" types such as shape are called *coproducts*. In Scala we typically represent products using case classes and coproducts using sealed traits:

```
sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends Shape
final case class Circle(radius: Double) extends Shape

val rect: Shape = Rectangle(3.0, 4.0)
val circ: Shape = Circle(1.0)
```

The beauty of ADTs is that they are completely type safe. The compiler has complete knowledge of the algebras we define, so it can support us in writing complete, correctly typed methods involving our types:

```
def area(shape: Shape): Double =
  shape match {
    case Rectangle(w, h) => w * h
    case Circle(r)       => math.Pi * r * r
  }

area(rect)
// res1: Double = 12.0
```

[1]Be careful not to confuse algebraic data types with "abstract data types", which are a different modelling tool with little bearing on the discussion here.

```
area(circ)
// res2: Double = 3.141592653589793
```

2.1.1 Alternative encodings

Sealed traits and case classes are undoubtedly the most convenient encoding of ADTs in Scala. However, they aren't the *only* encoding. For example, the Scala standard library provides generic products in the form of `Tuples` and a generic coproduct in the form of `Either`. We could have chosen these to encode our Shape:

```
type Rectangle2 = (Double, Double)
type Circle2    = Double
type Shape2     = Either[Rectangle2, Circle2]

val rect2: Shape2 = Left((3.0, 4.0))
val circ2: Shape2 = Right(1.0)
```

While this encoding is less readable than the case class encoding above, it does have some of the same desirable properties. We can still write completely type safe operations involving Shape2:

```
def area2(shape: Shape2): Double =
  shape match {
    case Left((w, h)) => w * h
    case Right(r)     => math.Pi * r * r
  }

area2(rect2)
// res4: Double = 12.0

area2(circ2)
// res5: Double = 3.141592653589793
```

Importantly, Shape2 is a more *generic* encoding than Shape[2]. Any code that operates on a pair of `Doubles` will be able to operate on a `Rectangle2` and

[2]We're using "generic" with an informal way here, not the formal meaning of "a type with a type parameter".

vice versa. As Scala developers we tend to see interoperability as a bad thing: what havoc will we accidentally wreak across our codebase with such freedom?! However, in some cases it is a desirable feature. For example, if we're serializing data to disk, we don't care about the difference between a pair of `Doubles` and a `Rectangle2`. Just write or read two numbers and we're done.

shapeless gives us the best of both worlds: we can use friendly sealed traits and case classes by default, and switch to generic representations when we want interoperability (more on this later). However, instead of using `Tuples` and `Either`, shapeless uses its own data types to represent generic products and coproducts. We'll introduce to these types in the next sections.

2.2 Generic product encodings

In the previous section we introduced tuples as a generic representation of products. Unfortunately, Scala's built-in tuples have a couple of disadvantages that make them unsuitable for shapeless' purposes:

1. each size of tuple has a different, unrelated type, making it difficult to write code that abstracts over sizes;
2. there are no types for 0- and 1-length tuples, which are important for representing products with 0 and 1 fields.

For these reasons, shapeless uses a different generic encoding for product types called *heterogeneous lists* or `HLists`[3].

An `HList` is either the empty list `HNil`, or a pair `::[H, T]` where `H` is an arbitrary type and `T` is another `HList`. Because every `::` has its own `H` and `T`, the type each element is encoded separately in the type of the overall list:

[3]`Product` is perhaps a better name for `HList`, but the standard library unfortunately already has a type `scala.Product`.

```scala
import shapeless.{HList, ::, HNil}

val product: String :: Int :: Boolean :: HNil =
  "Sunday" :: 1 :: false :: HNil
```

The types and values of the `HLists` above mirror one another. Both represent three members: a `String`, an `Int`, and a `Boolean`. We can retrieve the head and `tail` and the types of the elements are preserved:

```scala
val first: String =
  product.head
// first: String = Sunday

val second: Int =
  product.tail.head
// second: Int = 1

val rest: Boolean :: HNil =
  product.tail.tail
// rest: shapeless.::[Boolean,shapeless.HNil] = false :: HNil
```

The compiler knows the exact length of each `HList`, so it becomes a compilation error to take the head or `tail` of an empty list:

```scala
product.tail.tail.tail.head
// <console>:15: error: could not find implicit value for parameter c:
     shapeless.ops.hlist.IsHCons[shapeless.HNil]
//        product.tail.tail.tail.head
//                                 ^
```

We can manipulate and transform `HLists` in addition to being able to inspect and traverse them. For example, we can prepend an element with the `::` method. Again, notice how the type of the result reflects the number and types of its elements:

```scala
val newProduct: Long :: String :: Int :: Boolean :: HNil =
  42L :: product
// newProduct: shapeless.::[Long,shapeless.::[String,shapeless.::[Int,
     shapeless.::[Boolean,shapeless.HNil]]]] = 42 :: Sunday :: 1 ::
```

```
    false :: HNil
```

Shapeless also provides tools for performing more complex operations such as mapping, filtering, and concatenating lists. We'll discuss these in more detail in later chapters.

DJG: INSERT CHAPTER NUMBER ABOVE

2.2.1 Switching representations using *Generic*

Shapeless provides a type class called `Generic` that allows us to switch back and forth between a concrete ADT and its generic representation. There's some macro magic going on behind the scenes that allows us to summon instances of `Generic` without boilerplate:

```
import shapeless.Generic

case class IceCream(name: String, numCherries: Int, inCone: Boolean)

val iceCreamGen = Generic[IceCream]
// iceCreamGen: shapeless.Generic[IceCream]{type Repr = shapeless.::[
    String,shapeless.::[Int,shapeless.::[Boolean,shapeless.HNil]]]} =
    anon$macro$4$1@22069ec5
```

Note that the instance of `Generic` has a type member `Repr` containing the type of its generic representation. In this case `iceCreamGen.Repr` is `String :: Int :: Boolean :: HNil`. Instances of `Generic` have two methods: one for converting `to` `Repr` and one for converting `from` it:

```
val iceCream: IceCream =
  IceCream("Sundae", 1, false)
// iceCream: IceCream = IceCream(Sundae,1,false)

val repr: iceCreamGen.Repr =
  iceCreamGen.to(iceCream)
// repr: iceCreamGen.Repr = Sundae :: 1 :: false :: HNil

val iceCream2: IceCream =
```

```
  iceCreamGen.from(repr)
// iceCream2: IceCream = IceCream(Sundae,1,false)
```

If two ADTs have the same `Repr`, we can convert back and forth between them using their `Generic`s:

```
case class Employee(name: String, number: Int, manager: Boolean)

// Create an employee from an ice cream:
val strangeEmployee: Employee =
  Generic[Employee].from(Generic[IceCream].to(iceCream))
// strangeEmployee: Employee = Employee(Sundae,1,false)
```

2.3 Generic coproducts

Now we know how shapeless encodes product types. What about coproducts? We looked at `Either` earlier, but that suffers from a similar drawback to tuples: we have no way of representing a disjunction of fewer than two types. For this reason, shapeless provides a different encoding that is similar to `HList`.

The type of a `Coproduct` encodes all the possible types in the disjunction, but each concrete instantiation contains a value for just one of the possibilities.

```
import shapeless.{Coproduct, :+:, CNil}

case class Red()
case class Amber()
case class Green()

type Light = Red :+: Amber :+: Green :+: CNil
```

In general, coproducts take the form `A :+: B :+: C :+: CNil` meaning "A or B or C". `:+:` can be loosely interpreted as an `Either`, with subtypes `Inl` and `Inr` corresponding loosely to `Left` and `Right`. `CNil` is an empty type with no values, similar to `Nothing`, so we can never instantiate an empty `Coproduct`. Similarly, we can never create a `Coproduct` purely from instances of `Inr`. We always have to have exactly one `Inl` in a value:

```
import shapeless.{Inl, Inr}

val red: Light = Inl(Red())
val green: Light = Inr(Inr(Inl(Green())))
```

2.3.1 Switching encodings using *Generic*

`Coproduct` types are difficult to parse on first glance. However, it is relatively easy to see how they fit into the larger picture of generic encodings. In addition to understanding case classes and case objects, shapeless' `Generic` type class also understands sealed traits and abstract classes:

```
import shapeless.Generic

sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends Shape
final case class Circle(radius: Double) extends Shape

val gen = Generic[Shape]
// gen: shapeless.Generic[Shape]{type Repr = shapeless.:+:[Rectangle,
     shapeless.:+:[Circle,shapeless.CNil]]} = anon$macro$1$1@367e48c5
```

Note that the `Repr` of the `Generic` is a `Coproduct` of the subtypes of the sealed trait: `Rectangle :+: Circle :+: CNil`. We can use the `to` and `from` methods of the generic to map back and forth between `Shape` and `gen.Repr`:

```
gen.to(Rectangle(3.0, 4.0))
// res4: gen.Repr = Inl(Rectangle(3.0,4.0))

gen.to(Circle(1.0))
// res5: gen.Repr = Inr(Inl(Circle(1.0)))
```

2.4 Summary

In this chapter we discussed the generic representations shapeless provides for algebraic data types in Scala: `HLists` for product types and `Coproducts` for coproduct types. We also introduced the `Generic` type class to map back and forth between concrete ADTs and their generic representations. We haven't yet discussed why generic encodings are so attractive. The one use case we did cover—converting between ADTs—is fun but not tremendously useful.

The real power of `HLists` and `Coproducts` comes from their recursive structure. We can write code to traverse the structures and calculate values from their constituents. In the next chapter we will look at our first real use case: automatically deriving type class instances.

Chapter 3

Automatically deriving type class instances

In the last chapter we saw how the `Generic` type class allowed us to convert any instance of an ADT to a generic encoding made of `HLists` and `Coproducts`. In this chapter we will look at our first serious use case: automatic derivation of type class instances.

3.1 Recap: type classes

Before we get into the depths of instance derivation, let's quickly recap on the important aspects of type classes.

Type classes are a programming pattern borrowed from Haskell (the word "class" has nothing to do with classes in object oriented programming). We encode them in Scala using traits and implicit parameters. A *type class* is a generic trait representing some sort of general functionality that we would like to apply to a wide range of types:

```
// Turn a value of type A into a row of cells in a CSV file:
trait CsvEncoder[A] {
```

```
  def encode(value: A): List[String]
}
```

We implement our type class with *instances* for each type we care about:

```
// Helper method for creating CsvEncoder instances:
def createEncoder[A](func: A => List[String]): CsvEncoder[A] =
  new CsvEncoder[A] {
    def encode(value: A): List[String] =
      func(value)
  }

// Custom data type:
case class Employee(name: String, number: Int, manager: Boolean)

// CsvEncoder instance for the custom data type:
implicit val employeeEncoder: CsvEncoder[Employee] =
  createEncoder(e => List(
    e.name,
    e.number.toString,
    if(e.manager) "yes" else "no"
  ))
```

We mark each instance with the keyword `implicit`, and define a generic *entry point* method that accepts an implicit parameter of the corresponding type:

```
def writeCsv[A](values: List[A])(implicit enc: CsvEncoder[A]): String =
  values.map(value => enc.encode(value).mkString(",")).
    mkString("\n")
```

When we call the entry point, the compiler calculates the value of the type parameter and searches for an implicit `CsvWriter` of the corresponding type:

```
val employees: List[Employee] = List(
  Employee("Bill", 1, true),
  Employee("Peter", 2, false),
  Employee("Milton", 3, false)
)

// The compiler inserts a CsvEncoder[Employee] parameter:
```

```
writeCsv(employees)
// res7: String =
// Bill,1,yes
// Peter,2,no
// Milton,3,no
```

We can use `writeCsv` with any data type we like, provided we have a corresponding implicit `CsvEncoder` in scope:

```
case class IceCream(name: String, numCherries: Int, inCone: Boolean)

implicit val iceCreamEncoder: CsvEncoder[IceCream] =
  createEncoder(i => List(
    i.name,
    i.numCherries.toString,
    if(i.inCone) "yes" else "no"
  ))

val iceCreams: List[IceCream] = List(
  IceCream("Sundae", 1, false),
  IceCream("Cornetto", 0, true),
  IceCream("Banana Split", 0, false)
)

writeCsv(iceCreams)
// res10: String =
// Sundae,1,no
// Cornetto,0,yes
// Banana Split,0,no
```

3.1.1 Resolving instances

Type classes are very flexible but they require us to define instances for every type we care about. Fortunately, the Scala compiler has a few tricks up its sleeve to resolve instances for us given sets of user-defined rules. For example, we can write a rule that creates a `CsvEncoder` for `(A, B)` given `CsvEncoders` for `A` and `B`:

```scala
implicit def pairEncoder[A, B](
  implicit
  aEncoder: CsvEncoder[A],
  bEncoder: CsvEncoder[B]
): CsvEncoder[(A, B)] =
  createEncoder {
    case (a, b) =>
      aEncoder.encode(a) ++ bEncoder.encode(b)
  }
```

When all the parameters to an `implicit def` are themselves marked as `implicit`, the compiler can use it as a *resolution rule* to create instances from other instances. For example, if we call `writeCsv` and pass in a `List[(Employee, IceCream)]`, the compiler is able to combine `pairEncoder`, `employeeEncoder`, and `iceCreamEncoder` to produce the required `CsvEncoder[(Employee, IceCream)]`:

```scala
writeCsv(employees zip iceCreams)
// res11: String =
// Bill,1,yes,Sundae,1,no
// Peter,2,no,Cornetto,0,yes
// Milton,3,no,Banana Split,0,no
```

Given a set of rules encoded as `implicit vals` and `implicit defs`, the compiler is capable of *searching* for combinations to give it the required instances. This behaviour, known as "implicit resolution", is what makes the type class pattern so powerful.

Traditionally the only limitation to this has been ADTs. The compiler can't pull apart the types of case classes and sealed traits, so we have always had to define instances for ADTs by hand. Shapeless' generic representations change all of this, allowing us to derive instances for any ADT for free.

3.2 Deriving instances for products

In this section we're going to use shapeless to derive type class instances for products (i.e. case classes). We'll use two intuitions:

1. If we have type class instances for the head and tail of an `HList`, we can derive an instance for the whole `HList`.

2. If we have a case class `A`, a `Generic[A]`, and a type class instance for the generic's `Repr`, we can combine them to create an instance for `A`

Take `CsvEncoder` and `IceCream` as examples:

- `IceCream` has a generic `Repr` of type `String :: Int :: Boolean :: HNil`.

- The `Repr` is made up of a `String`, an `Int`, a `Boolean`, and an `HNil`. If we have `CsvEncoders` for these types, we can create an encoder for the whole thing.

- If we can derive a `CsvEncoder` for the `Repr`, we can create one for `IceCream`.

3.2.1 Instances for *HLists*

Let's start by writing `CsvEncoders` for `String`, `Int`, and `Boolean`. See Section 3.1 for the definition of `createEncoder`:

```
implicit val stringEncoder: CsvEncoder[String] =
  createEncoder(str => List(str))

implicit val intEncoder: CsvEncoder[Int] =
  createEncoder(num => List(num.toString))

implicit val booleanEncoder: CsvEncoder[Boolean] =
  createEncoder(bool => List(if(bool) "yes" else "no"))
```

We can combine these building blocks to create an encoder for our `HList`. We'll use two rules: one for an `HNil` and one for `::`:

```scala
import shapeless.{HList, ::, HNil}

implicit val hnilEncoder: CsvEncoder[HNil] =
  createEncoder(hnil => Nil)

implicit def hlistEncoder[H, T <: HList](
  implicit
  hEncoder: CsvEncoder[H],
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :: T] =
  createEncoder {
    case h :: t =>
      hEncoder.encode(h) ++ tEncoder.encode(t)
  }
```

Taken together, these five rules allow us to summon `CsvEncoders` for any `HList` involving `Strings`, `Ints`, and `Booleans`:

```scala
val reprEncoder: CsvEncoder[String :: Int :: Boolean :: HNil] =
  implicitly

reprEncoder.encode("abc" :: 123 :: true :: HNil)
// res8: List[String] = List(abc, 123, yes)
```

3.2.2 Instances for concrete products {??}

We can combine our derivation rules for `HLists` with an instance of `Generic` to produce a `CsvEncoder` for `IceCream`:

```scala
import shapeless.Generic

implicit val iceCreamEncoder: CsvEncoder[IceCream] = {
  val gen = Generic[IceCream]
  val enc = implicitly[CsvEncoder[gen.Repr]]
  createEncoder(iceCream => enc.encode(gen.to(iceCream)))
}

writeCsv(iceCreams)
// res10: String =
```

```
// Sundae,1,no
// Cornetto,0,yes
// Banana Split,0,no
```

This solution is specific to `IceCream`. Ideally we'd like to have a single rule that handles all case classes that have a `Generic` and a matching `CsvEncoder`. Let's work through the derivation step by step. Here's a first cut:

```scala
implicit def genericEncoder[A](
  implicit
  gen: Generic[A],
  enc: CsvEncoder[???]
): CsvEncoder[A] = createEncoder(a => enc.encode(gen.to(a)))
```

The first problem we have is selecting a type to put in place of the `???`. We want to write the `Repr` type associated with gen, but we can't do this:

```scala
implicit def genericEncoder[A](
  implicit
  gen: Generic[A],
  enc: CsvEncoder[gen.Repr]
): CsvEncoder[A] =
  createEncoder(a => enc.encode(gen.to(a)))
// <console>:26: error: illegal dependent method type: parameter may
     only be referenced in a subsequent parameter section
//           gen: Generic[A],
//           ^
```

The problem here is a scoping issue: we can't refer to a type member of one parameter from another parameter in the same block. We won't dwell on the details here, but the trick to solving this kind of problem is to introduce a new type parameter to our method and refer to it in each of the associated parameters:

```scala
implicit def genericEncoder[A, R](
  implicit
  gen: Generic[A] { type Repr = R },
  enc: CsvEncoder[R]
): CsvEncoder[A] =
```

```
  createEncoder(a => enc.encode(gen.to(a)))
```

We'll cover this coding style in more detail the next chapter. Suffice to say, this definition now compiles and works as expected and we can use it with any case class as expected. Intuitively, this definition says:

> *Given a type A and an `HList` type R, an implicit `Generic` to map A to R, and a `CsvEncoder` for R, create a `CsvEncoder` for A.*

We now have a complete system that handles any case class. The compiler expands a call like:

```
writeCsv(iceCreams)
```

to use our family of derivation rules:

```
writeCsv(iceCreams)(
  genericEncoder(
    Generic[IceCream],
    hlistEncoder(stringEncoder,
      hlistEncoder(intEncoder,
        hlistEncoder(booleanEncoder, hnilEncoder)))))
```

I'm sure you'll agree, it's nice not to have to write this code by hand!

Aux type aliases

Type refinements like `Generic[A] { type Repr = L }` are verbose and difficult to read, so shapeless provides a type alias `Generic.Aux` to rephrase the type member as a type parameter:

```
package shapeless

object Generic {
  type Aux[A, R] = Generic[A] { type Repr = R }
}
```

Using this alias we get a much more readable definition:

```scala
implicit def genericEncoder[A, R](
  implicit
  gen: Generic.Aux[A, R],
  env: CsvEncoder[R]
): CsvEncoder[A] =
  createEncoder(a => env.encode(gen.to(a)))
```

Note that the Aux type isn't changing any semantics here—it's just making things easier to read. This pattern is used frequently in the shapeless codebase.

3.2.3 So what are the downsides?

If all of the above seems pretty magical, allow us to provide one significant dose of reality. If things go wrong, the compiler isn't great at telling us why.

There are two main reasons the code above might fail to compile. The first is when we can't find an implicit `Generic` instance. For example, here we try to call `writeCsv` with a non-case class:

```scala
class Foo(bar: String, baz: Int)

writeCsv(List(new Foo("abc", 123)))
// <console>:30: error: could not find implicit value for parameter
     encoder: CsvEncoder[Foo]
//        writeCsv(List(new Foo("abc", 123)))
//                ^
```

In this case the error message is relatively easy to understand. If shapeless can't calculate a `Generic` it means that the type in question isn't an ADT—somewhere in the algebra there is a type that isn't a case class or a sealed abstract type.

The other potential source of failure is when the compiler can't calculate a `CsvEncoder` for our `HList`. This normally happens because we don't have an encoder for one of the fields in our ADT. For example, so far we haven't defined a `CsvEncoder` for `java.util.Date`, so the following code fails:

```
import java.util.Date

case class Booking(room: String, date: Date)

writeCsv(List(Booking("Lecture hall", new Date())))
// <console>:32: error: could not find implicit value for parameter
     encoder: CsvEncoder[Booking]
//        writeCsv(List(Booking("Lecture hall", new Date())))
//                ^
```

The message we get here isn't very helpful. All the compiler knows is it tried a lot of implicit resolution rules and couldn't make them work. It has no idea which combination came closest to the desired result, so it can't tell us where the source(s) of failure lie.

There's not much good news here. We have find the source of the error ourselves by a process of elimination. We'll discuss debugging techniques in more detail next chapter. For now, the main redeeming feature is that implicit resolution always fails at compile time. There's little chance that we will end up with code that fails during execution.

3.3 Deriving instances for coproducts

In the last section we created a set of rules to automatically derive a `CsvEncoder` for any product type. In this section we will apply the same patterns to coproducts. Let's return to our shape ADT as an example:

```
sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends Shape
final case class Circle(radius: Double) extends Shape
```

The generic representation for Shape is `Rectangle :+: Circle :+: CNil`. We can write generic `CsvEncoders` for `:+:` and `CNil` using the same principles we used for `HLists`. Our existing product encoders will take care of `Rectangle` and `Circle`:

```scala
import shapeless.{Coproduct, :+:, CNil, Inl, Inr}

implicit val cnilEncoder: CsvEncoder[CNil] =
  createEncoder(cnil => throw new Exception("Mass hysteria!"))

implicit def coproductEncoder[H, T <: Coproduct](
  implicit
  hEncoder: CsvEncoder[H],
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :+: T] = createEncoder {
  case Inl(h) => hEncoder.encode(h)
  case Inr(t) => tEncoder.encode(t)
}
```

There are two key points of note:

1. Alarmingly, the encoder for `CNil` throws an exception! Don't panic, though. Remember that we can't actually create values of type `CNil`. It's just there as a marker for the compiler. It's ok to fail abruptly here because we should never reach this point.

2. Because `Coproducts` are *disjunctions* of types, the encoder for `:+:` has to *choose* whether to encode a left or right value. We pattern match on the two subtypes of `:+:`—`Inl` for left and `Inr` for right.

With these definitions and our product encoders from Section 3.2, we should be able to serialize a list of shapes. Let's give it a try:

```scala
val shapes: List[Shape] = List(
  Rectangle(3.0, 4.0),
  Circle(1.0)
)

writeCsv(shapes)
// <console>:33: error: could not find implicit value for parameter
     encoder: CsvEncoder[Shape]
//        writeCsv(shapes)
//                ^
```

Oh no, it failed! The error message is unhelpful as we discussed earlier. The reason for the failure is we don't have a `CsvEncoder` instance for `Double`:

```scala
implicit val doubleEncoder: CsvEncoder[Double] =
  createEncoder(d => List(d.toString))
```

With this definition in place, everything works as expected:

```scala
writeCsv(shapes)
// res7: String =
// 3.0,4.0
// 1.0
```

Aligning CSV Output

Our CSV encoder isn't very practical in its current form. It allows fields from `Rectangle` and `Circle` to occupy the same columns in the output. To fix this problem we need to modify the definition of `CsvEncoder` to incorporate the width of the data type and space the output accordingly. We leave this as an exercise to the reader.

3.4 Deriving instances for recursive types

Let's try something more ambitious—a binary tree:

```scala
sealed trait Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[A](value: A) extends Tree[A]
```

Theoretically we should already have all of the definitions in place to summon a CSV writer for this definition. However, calls to `writeCsv` fail to compile:

```scala
implicitly[CsvEncoder[Tree[Int]]]
// <console>:24: error: could not find implicit value for parameter e:
     CsvEncoder[Tree[Int]]
//        implicitly[CsvEncoder[Tree[Int]]]
//                  ^
```

The problem is that our type is recursive. The compiler senses an infinite loop applying our derivation rules and gives up.

3.4.1 Implicit Divergence

Implicit resolution is a search process. The compiler uses heuristics to determine whether it is "converging" on a solution. If the heuristics don't yield favorable results for a particular branch of search, the compiler assumes the branch is not converging and moves onto another.

One heuristic is specifically designed to avoid infinite loops. If the compiler sees the same target type twice in a particular branch of search, it gives up and moves on. We can see this happening if we look at the expansion for `CsvEncode[Tree[Int]]` The implicit resolution process goes through the following types:

```
implicitly[CsvEncoder[Tree[Int]]]                      // 1
implicitly[CsvEncoder[Branch[Int] :+: Leaf[Int] :+: CNil]] // 2
implicitly[CsvEncoder[Branch[Int]]]                    // 3
implicitly[CsvEncoder[Tree[Int] :: Tree[Int] :: HNil]]     // 4
implicitly[CsvEncoder[Tree[Int]]]                      // 5 uh oh
```

We see `Tree[A]` twice in lines 1 and 5, so the compiler moves onto another branch of search. The result is failure to find a suitable implicit.

In fact, the situation is worse than this. If the compiler sees the same type constructor twice and the complexity of the type parameters is *increasing*, it assumes that branch of search is "diverging". This is a problem for shapeless because types like `::[H, T]` and `:+:[H, T]` come up in different generic representations and cause the compiler to give up prematurely. Consider the following types:

```
case class Bar(baz: Int, qux: String)
case class Foo(bar: Bar)
```

The expansion for `Foo` looks like this:

```
implicitly[CsvEncoder[Foo]]                 // 1
implicitly[CsvEncoder[Bar :: HNil]]         // 2
```

```
implicitly[CsvEncoder[Bar]]                     // 3
implicitly[CsvEncoder[Int :: String :: HNil]] // 4 uh oh
```

The compiler attempts to resolve a `CsvEncoder[::[H, T]]` twice in this branch of search, on lines 2 and 4. The type parameter for T is more complex on line 4 than on line 2, so the compiler assumes (incorrectly in this case) that the branch of search is diverging. It moves onto another branch and, again, the result is failure to find a suitable implicit.

3.4.2 *Lazy*

Implicit divergence would be a show-stopper for libraries like shapeless. Fortunately, shapeless provides a type called `Lazy` as a workaround. `Lazy` does two things:

1. it delays resolution of an implicit parameter until it is strictly needed, permitting the derivation of self-referential implicits.
2. it guards against some of the aforementioned over-defensive convergence heuristics.

We use `Lazy` by wrapping it around specific implicit parameters. As a rule of thumb, it is always a good idea to wrap the "head" parameter of any `HList` or `Coproduct` rule and the "repr" parameter of any `Generic` rule in `Lazy`:

```
implicit def hlistEncoder[H, T <: HList](
  implicit
  hEncoder: Lazy[CsvEncoder[H]],
  tEncoder: CsvEncoder[T]
): CsvEncoder[H :: T] = createEncoder {
  case h :: t =>
    hEncoder.value.encode(h) ++ tEncoder.encode(t)
}

implicit def coproductEncoder[H, T <: Coproduct](
  implicit
  hEncoder: Lazy[CsvEncoder[H]], // wrapped in Lazy
  tEncoder: CsvEncoder[T]
```

```
): CsvEncoder[H :+: T] = createEncoder {
  case Inl(h) => hEncoder.value.encode(h)
  case Inr(t) => tEncoder.encode(t)
}

implicit def genericEncoder[A, R](
  implicit
  gen: Generic.Aux[A, R],
  rEncoder: Lazy[CsvEncoder[R]] // wrapped in Lazy
): CsvEncoder[A] = createEncoder { value =>
  rEncoder.value.encode(gen.to(value))
}
```

This prevents the compiler giving up prematurely, and enables the solution to work on complex/recursive types like `Tree`:

```
implicitly[CsvEncoder[Tree[Int]]]
// res2: CsvEncoder[Tree[Int]] = $anon$1@127c199
```

3.5 Summary

In this chapter we discussed how to use `Generic`, `HLists`, and `Coproducts` to automatically derive type class instances. We also covered the `Lazy` type as a means of handling complex/recursive types.

Taking all of this into account, we can write a common skeleton for deriving type class instances as follows:

```
import shapeless._

// Step 1. Define the type class

trait MyTC[A]

// Step 2. Define basic instances

implicit def intInstance: MyTC[Int] = ???
implicit def stringInstance: MyTC[String] = ???
implicit def booleanInstance: MyTC[Boolean] = ???
```

```scala
// Step 3. Define instances for HList

implicit def hnilInstance: MyTC[HNil] =
  ???

implicit def hlistInstance[H, T <: HList](
  implicit
  hInstance: Lazy[MyTC[H]], // wrap in Lazy
  tInstance: MyTC[T]
): MyTC[H :: T] = ???

// Step 4. Define instances for Coproduct

implicit def cnilInstance: MyTC[CNil] =
  ???

implicit def coproductInstance[H, T <: Coproduct](
  implicit
  hInstance: Lazy[MyTC[H]], // wrap in Lazy
  tInstance: MyTC[T]
): MyTC[H :+: T] = ???

// Step 4. Define an instance for Generic

implicit def genericInstance[A, R](
  implicit
  generic: Generic.Aux[A, R],
  rInstance: Lazy[MyTC[R]]  // wrap in Lazy
): MyTC[A] = ???
```

In the next chapter we'll cover some useful theory, programming patterns, and debugging techniques to help write code in this style. In Chapter 5 we will revisit type class derivation using a variant of `Generic` that allows us to inspect field and type names in our ADTs.

Chapter 4

Working with types and implicits

In the last chapter we saw one of the most compelling use cases for shapeless: automatically deriving type class instances. There are plenty of even more powerful examples coming later. However, before we move on, we should take time to dicuss some of theory we've overlooked and establish a set of patterns for writing and debugging type- and implicit-heavy code.

4.1 Theory: dependent types

Last chapter we spent a lot of time using `Generic`, the type class for mapping ADT types to generic representations. However, we haven't yet discussed an important bit of theory that underpins `Generic` and much of shapeless: *dependent types*.

To illustrate this, let's take a closer look at `Generic`. Here's a simplified version of the definition:

```
package shapeless

trait Generic[A] {
  type Repr
  def to(value: A): Repr
```

```
  def from(value: Repr): A
}
```

Instances of `Generic` reference two other types: a type parameter `A` and a type member `Repr`. Suppose we implement a method `getRepr` as follows. What type will we get back?

```scala
import shapeless.Generic

def getRepr[A](value: A)(implicit gen: Generic[A]) =
  gen.to(value)
```

The answer is it depends on the instance we get for gen. In expanding the call to `getRepr`, the compiler will search for a `Generic[A]` and the result type will be whatever `Repr` is defined in that instance:

```scala
case class Vec(x: Int, y: Int)
case class Rect(origin: Vec, size: Vec)

getRepr(Vec(1, 2))
// res1: shapeless.::[Int,shapeless.::[Int,shapeless.HNil]] = 1 :: 2 ::
     HNil

getRepr(Rect(Vec(0, 0), Vec(5, 5)))
// res2: shapeless.::[Vec,shapeless.::[Vec,shapeless.HNil]] = Vec(0,0)
     :: Vec(5,5) :: HNil
```

What we're seeing here *dependent typing*: the return type of `getRepr` is dependent on types defined in its value parameters. Suppose we had specified `Repr` as type parameter on `Generic` instead of a type member:

```scala
trait Generic2[A, Repr]

def getRepr2[A, R](value: A)(implicit generic: Generic2[A, R]): R =
  ???
```

We would have had to pass the desired value of `Repr` to `getRepr` as a type parameter, effectively making `getRepr` useless.

The intuitive take-away from this is that type parameters are useful as "inputs" and type members are useful as "outputs".

4.2 Dependently typed functions

Shapeless uses dependent types all over the place: in `Generic`, in `Witness` (which we will see next chapter), and in a host of other implicit values that operate on `HLists`.

For example, shapeless provides a type class called `Last` that returns the last element in an `HList`. Here's a simplified version of its definition:

```scala
package shapeless.ops.hlist

trait Last[L <: HList] {
  type Out
  def apply(in: L): Out =
    ??? // definition omitted for brevity
}
```

We can summon instances of `Last` to inspect `HLists` in our code. In the two examples below note that the `Out` types are dependent on the `HList` types we started with:

```scala
import shapeless.{HList, ::, HNil}

import shapeless.ops.hlist.Last

val last1 = implicitly[Last[String :: Int :: HNil]]
// last1: shapeless.ops.hlist.Last[shapeless.::[String,shapeless.::[Int,
     shapeless.HNil]]] = shapeless.ops.hlist$Last$$anon$34@24948eba

val last2 = implicitly[Last[Int :: String :: HNil]]
// last2: shapeless.ops.hlist.Last[shapeless.::[Int,shapeless.::[String,
     shapeless.HNil]]] = shapeless.ops.hlist$Last$$anon$34@19609594
```

Once we have summoned instances of `Last`, we can use them at the value level:

```
last1("foo" :: 123 :: HNil)
// res1: last1.Out = 123

last2(321 :: "bar" :: HNil)
// res2: last2.Out = bar
```

We get two forms of protection against errors. The implicits defined for `Last` ensure we can only summon instances if the input `HList` has at least one element:

```
implicitly[Last[HNil]]
// <console>:15: error: Implicit not found: shapeless.Ops.Last[shapeless
     .HNil]. shapeless.HNil is empty, so there is no last element.
//        implicitly[Last[HNil]]
//                  ^
```

In addition, the type parameters on the instances of `Last` check whether we pass in the expected type of `HList`:

```
last1(321 :: "bar" :: HNil)
// <console>:16: error: type mismatch;
//  found   : shapeless.::[Int,shapeless.::[String,shapeless.HNil]]
//  required: shapeless.::[String,shapeless.::[Int,shapeless.HNil]]
//        last1(321 :: "bar" :: HNil)
//                  ^
```

As a further example, let's implement our own type class, called `Second`, that returns the second element in an `HList`:

```
trait Second[H <: HList] {
  type Out
  def apply(value: H): Out
}

implicit def hlistSecond[A, B, Rest <: HList]: Second[A :: B :: Rest] =
  new Second[A :: B :: Rest] {
    type Out = B
    def apply(value: A :: B :: Rest): B =
```

```
      value.tail.head
  }
```

We can summon instances of `Second` subject to similar constraints to `Last`:

```
val second1 = implicitly[Second[String :: Boolean :: Int :: HNil]]
// second1: Second[shapeless.::[String,shapeless.::[Boolean,shapeless
     .::[Int,shapeless.HNil]]]] = $anon$1@16453177

val second2 = implicitly[Second[String :: Int :: Boolean :: HNil]]
// second2: Second[shapeless.::[String,shapeless.::[Int,shapeless.::[
     Boolean,shapeless.HNil]]]] = $anon$1@2954f0fa

implicitly[Second[String :: HNil]]
// <console>:17: error: could not find implicit value for parameter e:
     Second[shapeless.::[String,shapeless.HNil]]
//        implicitly[Second[String :: HNil]]
//                  ^
```

And use them at the value level in the same way:

```
second1("foo" :: true :: 123 :: HNil)
// res7: second1.Out = true

second2("bar" :: 321 :: false :: HNil)
// res8: second2.Out = 321

second1("baz" :: HNil)
// <console>:18: error: type mismatch;
//  found   : shapeless.::[String,shapeless.HNil]
//  required: shapeless.::[String,shapeless.::[Boolean,shapeless.::[Int,
     shapeless.HNil]]]
//        second1("baz" :: HNil)
//                      ^
```

4.2.1 Chaining dependent functions

Dependently typed functions provide a means of calculating one type from another. We can *chain* dependently typed functions to perform calculations involving multiple steps. For example, we should be able to use a `Generic` to calculate a Repr for a case class, and use a Last to calculate the type of the last element. Let's try coding this:

```
def lastField[A](input: A)(
  implicit
  gen: Generic[A],
  last: Last[gen.Repr]
): last.Out = last.apply(gen.to(input))
// <console>:19: error: illegal dependent method type: parameter may
     only be referenced in a subsequent parameter section
//          gen: Generic[A],
//          ^
```

Unfortunately our code doesn't compile. This is the same problem we had in Section **??** with our definition of genericEncoder. We worked around the problem by lifting the free type variable out as a type parameter:

```
def lastField[A, Repr <: HList](input: A)(
  implicit
  gen: Generic.Aux[A, Repr],
  last: Last[Repr]
): last.Out = last.apply(gen.to(input))

lastField(Rect(Vec(1, 2), Vec(3, 4)))
// res10: Vec = Vec(3,4)
```

As a general rule, we always write code in this style. By encoding all the free variables as type parameters, we enable the compiler to unify them with appropriate types. This goes for more subtle constraints as well. For example, suppose we wanted to summon a Generic for a case class of exactly one field. We might be tempted to write this:

```
def getWrappedValue[A, Head](input: A)(
  implicit
  gen: Generic.Aux[A, Head :: HNil]
): Head = gen.to(input).head
```

The result here is more insidious. The method definition compiles but the compiler can never find the implicits its needs at the call site:

```
case class Wrapper(value: Int)

getWrappedValue(Wrapper(42))
// <console>:21: error: could not find implicit value for parameter gen:
      shapeless.Generic.Aux[Wrapper,shapeless.::[Head,shapeless.HNil]]
//        getWrappedValue(Wrapper(42))
//                       ^
```

The error message hints at the problem:

> error: could not find implicit value for parameter gen:
>
> `Generic.Aux[Wrapper, Head :: HNil]`

The clue is in the appearance of the type `Head`. This is the name of a type parameter in the method: it shouldn't be appearing in the type the compiler is trying to unify. The problem is that the `gen` parameter is over-constrained: the compiler can't find a `Repr` *and* ensure its length at the same time. `Nothing` also often provides a clue, appearing when the compiler fails to unify covariant type parameters.

The solution to our problem above is to separate implicit resolution into steps:

1. find a `Generic` with a suitable `Repr` for `A`;
2. provide that the `Repr` has a `Head` type.

Here's a revised version of the method using `=:=` to constrain `Repr`:

```
def getWrappedValue[A, Repr <: HList, Head, Tail <: HList](input: A)(
  implicit
  gen: Generic.Aux[A, Repr],
  ev: (Head :: Tail) =:= Repr
): Head = gen.to(input).head
// <console>:21: error: could not find implicit value for parameter c:
     shapeless.ops.hlist.IsHCons[gen.Repr]
//        ): Head = gen.to(input).head
//                             ^
```

This doesn't compile because the head method in the method body requires an implicit parameter of type `IsHCons`. This is a much simpler error message to fix—we just need to learn a tool from shapeless' toolbox. `IsHCons` is a shapeless type class that splits an `HList` into a `Head` and `Tail`. We can use `IsHCons` instead of `=:=`:

```
import shapeless.ops.hlist.IsHCons

def getWrappedValue[A, Repr <: HList, Head, Tail <: HList](in: A)(
  implicit
  gen: Generic.Aux[A, Repr],
  isHCons: IsHCons.Aux[Repr, Head, Tail]
): Head = gen.to(in).head
```

This fixes the bug. Both the method definition and the call site now compile as expected:

```
getWrappedValue(Wrapper(42))
// res13: Int = 42
```

The take home point here isn't that we solved the problem using `IsHCons`. Shapeless provides a lot of tools like this, and we can supplement them where necessary with our own type classes. The important point is the process we used to write code that compiles and is capable of finding solutions. We'll finish off this section with a step-by-step guide summarising our findings so far.

4.3 Summary

When coding with shapeless, we are often trying to find a target type that depends on the types we start with. This relationship is called *dependent typing*.

Problems involving dependent types can be conveniently expressed using implicit search, allowing the compiler to resolve intermediate and target types given a starting point at the call site.

We often have to use multiple steps to calculate a result (e.g. using a `Generic` to get a `Repr`, then using another type class to get to another type). When we do this, there are a few rules we can follow to ensure our code compiles and works as expected:

1. We should extract every intermediate type out to a type parameter. Many type parameters won't be used in the result, but the compiler needs them to know which types it has to unify.

2. The compiler resolves implicits from left to right, backtracking if it can't find a working combination. We should write implicits in the order we need them, using one or more type variables to connect them to previous implicits.

3. The compiler can only solve for one constraint at a time, so we mustn't over-constrain any single implicit.

4. We should state the return type explicitly, specifying any type parameters and type members that may ne needed elsewhere. Type members are often important. If we don't state them in the return type, they won't be available to the compiler for further implicit resolution.

5. The `Aux` type alias pattern is useful for keeping code readable. We should look out for `Aux` aliases when using tools from the shapeless toolbox, and implement `Aux` aliases on our own dependently typed functions.

> TODO: Section on debugging using `implicitly`?
>
> Section on debugging using `reify`?

Chapter 5

Accessing names during implicit derivation

Often, the type class instances we define need access to more than just types. In this chapter we will look at a variant of `Generic` called `LabelledGeneric` that gives us access to field names and type names.

To begin with we have some theory to cover. `LabelledGeneric` uses some clever techniques to expose name information at the type level. To understand these techniques we must discuss *literal types*, *singleton types*, *phantom types*, and *type tagging*.

5.1 Literal types

As Scala developers, we are used to the notion that a value may have multiple types. For example, the string `"hello"` has at least three types: `String`, `AnyRef`, and `Any`[1]:

[1] `String` has a bunch of other types like `Serializable` and `Comparable` but let's ignore those for now.

```
"hello" : String
// res0: String = hello

"hello" : AnyRef
// res1: AnyRef = hello

"hello" : Any
// res2: Any = hello
```

Interestingly, `"hello"` also has another type: a "singleton type" that belongs exclusively to that one value. This is similar to the singleton type we get when we define a companion object:

```
object Foo

Foo
// res3: Foo.type = Foo$@70c40426
```

Singleton types applied to literal values are called *literal types*. We don't normally interact with them because the default behaviour of the compiler is to "cast" literals to their nearest non-singleton type. So, for example, these two expressions are essentially equivalent:

```
"hello"
// res4: String = hello

("hello" : String)
// res5: String = hello
```

Shapeless provides a few tools for working with literal types. First, there is a `narrow` macro that converts a literal expression into a singleton-typed literal expression:

```
import shapeless.syntax.singleton._

var x = 42.narrow
// x: Int(42) = 42
```

Note the type of `x` here: `Int(42)` is a literal type. It is a subtype of `Int` that only contains the value 42. If we attempt to assign a different number to x, we get a compile error:

```
x = 43
// <console>:16: error: type mismatch;
//  found   : Int(43)
//  required: Int(42)
//        x = 43
//            ^
```

However, x is still an `Int` according to normal subtyping rules. If we operate on x we get a regular type of result:

```
x + 1
// res6: Int = 43
```

We can use `narrow` on any literal in Scala:

```
1.narrow
// res7: Int(1) = 1

true.narrow
// res8: Boolean(true) = true

"hello".narrow
// res9: String("hello") = hello

// and so on...
```

However, we can't use it on compound expressions: the compiler has to be able to determine the literal value straight from the source:

```
math.sqrt(4).narrow
// <console>:17: error: Expression scala.math.`package`.sqrt(4.0) does
     not evaluate to a constant or a stable reference value
//        math.sqrt(4).narrow
//                 ^
// <console>:17: error: value narrow is not a member of Double
//        math.sqrt(4).narrow
```

```
//                      ^
```

Literal types in Scala

Until recently, Scala had no syntax for writing literal types. The types were there in the compiler, but we couldn't express them directly in code. As of Lightbend Scala 2.12.1, Lightbend Scala 2.11.9, and Typelevel Scala 2.11.8, however, we now have direct syntax support for literal types. In these versions of Scala we can write declarations like the following:

```scala
val theAnswer: 42 = 42
```

The type 42 is the same as the type `Int(42)` we saw in printed output earlier. You'll still see `Int(42)` in output for legacy reasons, but the canonical syntax going forward is 42.

5.2 Type tagging and phantom types

Shapeless uses literal types to model the names of fields in case classes. It does this by "tagging" the types of the fields with the literal types of their names. Before we see how shapeless does this, we'll do it ourselves to show that there's no magic (well… minimal magic, at any rate). Suppose we have a number:

```scala
val number = 42
```

This number is an `Int` in two worlds: at runtime, where it has methods like `+` and `*`, and at compile-time, where the compiler uses the type to calculate which pieces of code work together and to search for implicits.

We can modify the type of `number` at compile time without modifying its runtime behaviour by "tagging" it with a "phantom type". Phantom types are types with no run-time semantics, like this:

```scala
trait Cherries
```

We can tag number using asInstanceOf. We end up with a value that is both an Int and a Cherries at compile-time, and an Int at run-time:

```scala
val numCherries = number.asInstanceOf[Int with Cherries]
// numCherries: Int with Cherries = 42
```

Shapeless uses this trick to tag the types of fields in a case classes with the singleton types of their names. If you find using asInstanceOf uncomfortable then don't worry: there's explicit syntax for tagging that avoids such unsavoriness:

```scala
import shapeless.labelled.{KeyTag, FieldType}
import shapeless.syntax.singleton._

val someNumber = 123

val numCherries = "numCherries" ->> someNumber
// numCherries: Int with shapeless.labelled.KeyTag[String("numCherries")
     ,Int] = 123
```

Here we are tagging someNumber with the phantom type KeyTag["numCherries", Int]. The tag encodes both the name and type of the field, both of which are useful when searching for entries in a Repr using implicit resolution. Shapeless also provides us with the FieldType type alias to make it easy to extract the key tag and value from a type:

```scala
type FieldType[K, V] = V with KeyTag[K, V]
```

Now we understand how shapeless tags the type of a value with its field name. But the key tag is just a phantom type: how do we convert it to a value we can use at runtime? Shapeless provides a type class called Witness for this purpose. If we combine Witness and FieldType, we get something very compelling—the ability extract the field name from a tagged field:

```scala
import shapeless.Witness

val numCherries = "numCherries" ->> 123
// numCherries: Int with shapeless.labelled.KeyTag[String("numCherries")
     ,Int] = 123

// Get the tag from a tagged value:
def getFieldName[K, V](value: FieldType[K, V])
    (implicit witness: Witness.Aux[K]): K =
  witness.value

getFieldName(numCherries)
// res14: String = numCherries

// Get the untagged value from a tagged value:
def getFieldValue[K, V](value: FieldType[K, V]): V =
  value

getFieldValue(numCherries)
// res16: Int = 123
```

5.2.1 Records and *LabelledGeneric*

Shapeless includes a set of tools for working with data structures called *records*. Records are `HLists` of items that are each tagged with type-level identifiers:

```scala
import shapeless.{HList, ::, HNil}

val garfield = ("cat" ->> "Garfield") ::  ("orange" ->> true) :: HNil
// garfield: shapeless.::[String with shapeless.labelled.KeyTag[String("
     cat"),String],shapeless.::[Boolean with shapeless.labelled.KeyTag[
     String("orange"),Boolean],shapeless.HNil]] = Garfield :: true ::
     HNil
```

For clarity, the type of `garfield` is as follows:

```scala
// FieldType["cat",    String]  ::
// FieldType["orange", Boolean] ::
// HNil
```

TODO: Insert link to records chapter if we have one.

We don't need to go into depth regarding records here, suffice to say that records are the generic representation used by the `LabelledGeneric` type class that we will discuss next. `LabelledGeneric` tags each item in a product or coproduct with the corresponding field or type name from the concrete ADT (although the names are represented as `Symbols`, not `Strings`). Accessing names without using reflection is incredibly compelling, so let's derive some type class instances using `LabelledGeneric`.

5.3 Deriving product instances with *LabelledGeneric*

We'll use a running example of JSON encoding to illustrate `LabelledGeneric`. We'll define a `JsonEncoder` type class that converts values to a JSON AST. This is the approach taken by argonaut, circe, play-json, spray-json, and many other Scala JSON libraries.

First we'll define our JSON data type:

```
sealed trait JsonValue
case class JsonObject(fields: List[(String, JsonValue)]) extends
     JsonValue
case class JsonArray(items: List[JsonValue]) extends JsonValue
case class JsonString(value: String) extends JsonValue
case class JsonNumber(value: Double) extends JsonValue
case class JsonBoolean(value: Boolean) extends JsonValue
case object JsonNull extends JsonValue
```

then the type class for encoding values as JSON:

```
trait JsonEncoder[A] {
  def encode(value: A): JsonValue
}
```

then a few basic instances:

```scala
def createEncoder[A](func: A => JsonValue): JsonEncoder[A] =
  new JsonEncoder[A] {
    def encode(value: A): JsonValue =
      func(value)
  }

implicit val stringEncoder: JsonEncoder[String] =
  createEncoder(str => JsonString(str))

implicit val doubleEncoder: JsonEncoder[Double] =
  createEncoder(num => JsonNumber(num))

implicit val intEncoder: JsonEncoder[Int] =
  createEncoder(num => JsonNumber(num))

implicit val booleanEncoder: JsonEncoder[Boolean] =
  createEncoder(bool => JsonBoolean(bool))

implicit def listEncoder[A]
    (implicit enc: JsonEncoder[A]): JsonEncoder[List[A]] =
  createEncoder(list => JsonArray(list.map(enc.encode)))

implicit def optionEncoder[A]
    (implicit enc: JsonEncoder[A]): JsonEncoder[Option[A]] =
  createEncoder(opt => opt.map(enc.encode).getOrElse(JsonNull))
```

Ideally, when we encode ADTs as JSON, we would like to use the correct field names in the output JSON:

```scala
case class IceCream(name: String, numCherries: Int, inCone: Boolean)

val iceCream = IceCream("Sundae", 1, false)

// Ideally we'd like to produce something like this:
val iceCreamJson: JsonValue =
  JsonObject(List(
    "name"        -> JsonString("Sundae"),
    "numCherries" -> JsonNumber(1),
    "inCone"      -> JsonBoolean(false)
  ))
```

This is where `LabelledGeneric` comes in. Let's summon an instance for `Ice-`

`Cream` and see what kind of representation it produces:

```
import shapeless.LabelledGeneric

val gen = LabelledGeneric[IceCream].to(iceCream)
// gen: shapeless.::[String with shapeless.labelled.KeyTag[Symbol with
     shapeless.tag.Tagged[String("name")],String],shapeless.::[Int with
     shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String("
     numCherries")],Int],shapeless.::[Boolean with shapeless.labelled.
     KeyTag[Symbol with shapeless.tag.Tagged[String("inCone")],Boolean],
     shapeless.HNil]]] = Sundae :: 1 :: false :: HNil
```

For clarity, the full type of the `HList` is:

```
// String  with KeyTag[Symbol with Tagged["name"], String]     ::
// Int     with KeyTag[Symbol with Tagged["numCherries"], Int] ::
// Boolean with KeyTag[Symbol with Tagged["inCone"], Boolean]  ::
// HNil
```

The type here is slightly more complex than we have seen. Instead of representing the field names with literal string types, shapeless is representing them with symbols tagged with literal string types. The details of the implementation aren't particularly important: we can still use `Witness` and `FieldType` to extract the tags, but they come out as `Symbols` instead of `Strings`.

5.3.1 Instances for *HLists*

Let's define `JsonEncoder` instances for `HNil` and `::`. These encoders are going to generate and manipulate `JsonObjects`, so we'll introduce a new type of encoder to make that easier:

```
trait JsonObjectEncoder[A] extends JsonEncoder[A] {
  def encode(value: A): JsonObject
}

def createObjectEncoder[A](fn: A => JsonObject): JsonObjectEncoder[A] =
  new JsonObjectEncoder[A] {
    def encode(value: A): JsonObject =
```

```
    fn(value)
}
```

The definition for `HNil` is then straightforward:

```
import shapeless.{HList, ::, HNil, Lazy}

implicit val hnilEncoder: JsonObjectEncoder[HNil] =
  createObjectEncoder(hnil => JsonObject(Nil))
```

The definition of `hlistEncoder` involves a few moving parts so we'll go through it piece by piece. We'll start with the definition we might expect if we were using regular `Generic`:

```
implicit def hlistObjectEncoder[H, T <: HList](
  implicit
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonEncoder[H :: T] = ???
```

`LabelledGeneric` will give us an `HList` of tagged types, so let's start by introducing a new type variable for the key type:

```
import shapeless.Witness
import shapeless.labelled.FieldType

implicit def hlistObjectEncoder[K, H, T <: HList](
  implicit
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = ???
```

In the body of our method we're going to need the value associated with `K`. We'll add an implicit `Witness[K]` to do this for us:

```
implicit def hlistObjectEncoder[K, H, T <: HList](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
```

```
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = {
  val fieldName = witness.value
  ???
}
```

We can access the value of K using `witness.value`, but the compiler has no way of knowing what type of tag we're going to get. `LabelledGeneric` uses `Symbols` as the tag types, so we'll put a type bound on K and use `symbol.name` to convert it to a `String`:

```
implicit def hlistObjectEncoder[K <: Symbol, H, T <: HList](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = {
  val fieldName: String = witness.value.name
  ???
}
```

The rest of the definition uses the principles we covered in Chapter 3:

```
implicit def hlistObjectEncoder[K <: Symbol, H, T <: HList](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :: T] = {
  val fieldName: String = witness.value.name
  createObjectEncoder { hlist =>
    val head = hEncoder.value.encode(hlist.head)
    val tail = tEncoder.encode(hlist.tail)
    JsonObject((fieldName, head) :: tail.fields)
  }
}
```

5.3.2 Instances for concrete products

Finally let's turn to our generic instance. This is identical to the definitions we've seen before, except that we're using `LabelledGeneric` instead of `Generic`:

```
import shapeless.LabelledGeneric

implicit def genericObjectEncoder[A, H <: HList](
  implicit
  generic: LabelledGeneric.Aux[A, H],
  hEncoder: Lazy[JsonObjectEncoder[H]]
): JsonEncoder[A] =
  createObjectEncoder { value =>
    hEncoder.value.encode(generic.to(value))
  }
```

And that's all we need! With these definitions in place we can serialize instances of any case class and retain the field names in the resulting JSON:

```
implicitly[JsonEncoder[IceCream]].encode(iceCream)
// res14: JsonValue = JsonObject(List((name,JsonString(Sundae)), (
     numCherries,JsonNumber(1.0)), (inCone,JsonBoolean(false))))
```

5.4 Deriving coproduct instances with *LabelledGeneric*

Applying `LabelledGeneric` with `Coproducts` involves a mixture of the concepts we've covered already. Let's start by examining a `Coproduct` type derived by `LabelledGeneric`. We'll re-visit our Shape ADT from Chapter 2:

```
import shapeless.LabelledGeneric

sealed trait Shape
final case class Rectangle(width: Double, height: Double) extends Shape
final case class Circle(radius: Double) extends Shape

LabelledGeneric[Shape].to(Circle(1.0))
```

```
// res5: shapeless.:+:[Rectangle with shapeless.labelled.KeyTag[Symbol
     with shapeless.tag.Tagged[String("Rectangle")],Rectangle],shapeless
     .:+:[Circle with shapeless.labelled.KeyTag[Symbol with shapeless.
     tag.Tagged[String("Circle")],Circle],shapeless.CNil]] = Inr(Inl(
     Circle(1.0)))
```

Here is that `Coproduct` type in a more readable format:

```
// Rectangle with KeyTag[Symbol with Tagged["Rectangle"], Rectangle] :+:
// Circle    with KeyTag[Symbol with Tagged["Circle"],    Circle]    :+:
// CNil
```

As you can see, the result is a `Coproduct` of the subtypes of `Shape`, each tagged with the type name. We can use this information to write `JsonEncoders` for `:+:` and `CNil`:

```
import shapeless.{Coproduct, :+:, CNil, Inl, Inr, Witness, Lazy}
import shapeless.labelled.FieldType

implicit val cnilObjectEncoder: JsonObjectEncoder[CNil] =
  createObjectEncoder(cnil => throw new Exception("Mass hysteria!"))

implicit def coproductObjectEncoder[K <: Symbol, H, T <: Coproduct](
  implicit
  witness: Witness.Aux[K],
  hEncoder: Lazy[JsonEncoder[H]],
  tEncoder: JsonObjectEncoder[T]
): JsonObjectEncoder[FieldType[K, H] :+: T] = {
  val typeName = witness.value.name
  createObjectEncoder {
    case Inl(h) =>
      JsonObject(List(typeName -> hEncoder.value.encode(h)))

    case Inr(t) =>
      tEncoder.encode(t)
  }
}
```

`coproductEncoder` follows the same pattern as `hlistEncoder`. We have three type parameters: `K` for the type name, `H` for the value at the head of the `HList`, and `T` for the value at the tail. We use `FieldType` and `:+:` in the result type to declare the relationships between the three, and we use a `Witness` to access the runtime value of the type name. The result is an object containing a single key/value pair: the key being the type name and the value the result:

```
val shape: Shape = Circle(1.0)

implicitly[JsonEncoder[Shape]].encode(shape)
// res8: JsonValue = JsonObject(List((Circle,JsonObject(List((radius,
     JsonNumber(1.0)))))))
```

5.5 Summary

In this chapter we discussed `LabelledGeneric`, a variant of `Generic` that exposes type and field names in its generic representations.

The names exposed by `LabelledGeneric` are encoded at as type-level tags, so we can target them during implicit resolution. We started the chapter discussing *literal types* and the way shapeless uses them in its tags. We also discussed the `Witness` type class, which is used to reify literal types as values.

Finally, we brought `LabelledGeneric`, literal types, and `Witness` together to build a `JsonEcoder` library that includes sensible names in its output.

The key take home point from this chapter is that none of this code uses runtime reflection: it's all done with types, implicits, and a small set of macros that are internal to shapeless. The code we're generating is consequently very fast and reliable at runtime.

Chapter 6

TODOs

Still to do:

- Mention SI-7046 in Chapter 3
- Complete the "debugging" section in Chapter 4
- Maybe type class instance selection in Chapter 4
  - Low and high priority
  - `LowPriorityImplicit`
- Built-in `HList` and `Coproduct` operations
  - Migrating case class as a basic example
  - Polymorphic functions
  - Mapping over `HLists` as an example
- Counting with `Nat`
  - Generating `Arbitrary` instances as an example
- Built-in record operations
- Performance
  - `cachedImplicit`
  - Maybe `Cached`

    - Maybe

- Check cross references
- **SHIP IT!**