# Scala Ecosystem: Cats

Mikhail Mutcianko, Alexey Otts

СПБгУ, СП

20 апреля 2020 г.

# Typeclasses

# The Type Class

A *type class* is an interface or API that represents some functionality we want to implement.

In Cats a type class is represented by a trait with at least one type parameter. For example, we can represent generic "serialize to JSON" behaviour as follows:

# The Type Class

```scala
// Define a very simple JSON AST
sealed trait Json
final case class JsObject(get: Map[String, Json]) extends Json
final case class JsString(get: String) extends Json
final case class JsNumber(get: Double) extends Json
final case object JsNull extends Json

// The "serialize to JSON" behaviour is encoded in this trait
trait JsonWriter[A] {
  def write(value: A): Json
}
```

# Type Class Instances

The *instances* of a type class provide implementations of the type class for the types we care about, including types from the Scala standard library and types from our domain model.

In Scala we define instances by creating concrete implementations of the type class and tagging them with the `implicit` keyword:

# Type Class Instances

```scala
case class Person(name: String, email: String)

object JsonWriterInstances {
  implicit val personWriter: JsonWriter[Person] =
    new JsonWriter[Person] {
      def write(value: Person): Json =
        JsObject(Map(
          "name" -> JsString(value.name),
          "email" -> JsString(value.email)
        ))
    }
  // etc...
}
```

# Type Class Interfaces

A type class *interface* is any functionality we expose to users. Interfaces are generic methods that accept instances of the type class as implicit parameters.

There are two common ways of specifying an interface: *Interface Objects* and *Interface Syntax*.

- Interface Objects

  methods in objects that direcly apply to some value

- Interface Syntax

  intoduce implicit classes to add methods to already existing types

# Type Class Interfaces

## Interface Objects

```
object Json {
  def toJson[A](value: A)(implicit w: JsonWriter[A]): Json =
    w.write(value)
}
```

## Interface Syntax

```
object JsonSyntax {
  implicit class JsonWriterOps[A](value: A) {
    def toJson(implicit w: JsonWriter[A]): Json =
      w.write(value)
  }
}
```

# Importing Cats

- **type classes** are defined in `cats` package

```
import cats.Show
```

- **type class instances** in `cats.instances`

```
import cats.instances.int._    // for Show
import cats.instances.string._ // for Show
val showInt:    Show[Int]    = Show.apply[Int]
```

- **interface syntax** in `cats.syntax`

```
import cats.syntax.show._ // for show
val shownInt = 123.show
```

- all of the standard **type class** instances and all of the **syntax**

```
import cats.implicits._
```

# Example: Eq

We can use Eq to define type-safe equality between instances of any given type:

```scala
trait Eq[A] {
  def eqv(a: A, b: A): Boolean
  // other concrete methods based on eqv...
}
```

The interface syntax, defined in cats.syntax.eq provides two methods for performing equality checks provided there is an instance Eq[A] in scope:

- === compares two objects for equality
- =!= compares two objects for inequality

# Monoids and Semigroups

# Definition of a Monoid

Formally, a monoid for a type `A` is:

- an associative operation `combine` with type `(A, A) => A`
- an identity element `empty` of type `A`

Here is a simplified version of the definition from Cats:

```scala
trait Monoid[A] {
  def combine(x: A, y: A): A
  def empty: A
}
```

# Definition of a Semigroup

A *semigroup* is just the `combine` part of a monoid. While many semigroups are also monoids, there are some data types for which we cannot define an `empty` element.

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}
trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

# Monoid in Cats

`Monoid` follows the standard Cats pattern for the user interface: the companion object has an `apply` method that returns the type class instance for a particular type. For example, if we want the monoid instance for `String`, and we have the correct implicits in scope, we can write the following:

```scala
import cats.Monoid
import cats.instances.string._ // for Monoid

Monoid[String].combine("Hi ", "there")  // res0: String = "Hi there"
Monoid[String].empty                    // res1: String = ""
```

# Monoid Syntax

Cats provides syntax for the `combine` method in the form of the |+| operator. Because `combine` technically comes from `Semigroup`, we access the syntax by importing from `cats.syntax.semigroup`:

```scala
import cats.instances.string._ // for Monoid
import cats.syntax.semigroup._ // for |+|
// stringResult: String = "Hi there"
val stringResult = "Hi " |+| "there" |+| Monoid[String].empty

import cats.instances.int._ // for Monoid
val intResult = 1 |+| 2 |+| Monoid[Int].empty // intResult: Int = 3
```

# Examples

```scala
import cats.instances.{int, map, tuple, option}._ // not a valid syntax

Option(1) |+| Option(2) // res1: Option[Int] = Some(3)

val map1 = Map("a" -> 1, "b" -> 2)
val map2 = Map("b" -> 3, "d" -> 4)
map1 |+| map2 // res2: Map[String, Int] = Map("b" -> 5, "d" -> 4, "a" -> 1)

val tuple1 = ("hello", 123)
val tuple2 = ("world", 321)
tuple1 |+| tuple2 // res3: (String, Int) = ("helloworld", 444)
```

# Functors

# Definition of a Functor

Informally, a *functor* is anything with a `map` method. You probably know lots of types that have this: `Option`, `List`, and `Either`, to name a few.

```
List(1, 2, 3).map(n => n + 1) // res0: List[Int] = List(2, 3, 4)
```

Formally, a *functor* is a type `F[A]` with an operation map with type `(A => B) => F[B]`.

```
package cats
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

# Functor in Cats

```scala
import cats.Functor
import cats.instances.list._
val list1 = List(1, 2, 3)
val list2 = Functor[List].map(list1)(_ * 2) // list2: List[Int] = List(2, 4, 6)
```

`Functor` also provides the `lift` method, which converts a function of type `A => B` to one that operates over a functor and has type `F[A] => F[B]`:

```scala
val func = (x: Int) => x + 1
val liftedFunc = Functor[Option].lift(func) // liftedFunc: Option[Int] => Option[Int]
liftedFunc(Option(1))                       // res1: Option[Int] = Some(2)
```

# Monads

## Monad Definition

A monad's `flatMap` method allows us to specify what happens next, taking into account an intermediate complication. While we have only talked about `flatMap` above, monadic behaviour is formally captured in two operations:

- `pure`, of type `A => F[A]`

  abstracts over constructors, providing a way to create a new monadic context from a plain value

- `flatMap`, of type `(F[A], A => F[B]) => F[B]`

  extracting the value from a context and generating the next context in the sequence

# Monad in Cats

Here is a simplified version of the Monad type class in Cats:

```
trait Monad[F[_]] {
  def pure[A](value: A): F[A]
  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]
}
```

# Monadic Laws

`pure` and `flatMap` must obey a set of laws that allow us to sequence operations freely without unintended glitches and side-effects:

- **Left identity:** calling `pure` with `func` is the same as calling `func`:

```
pure(a).flatMap(func) == func(a)
```

- **Right identity:** passing `pure` to `flatMap` is the same as doing nothing:

```
m.flatMap(pure) == m
```

- **Associativity:** `flatMap` over `f` and `g` is the same as `flatMap` over `f` and then `flatMap g`

```
m.flatMap(f).flatMap(g) == m.flatMap(x => f(x).flatMap(g))
```

# The Monad Type Class

The monad type class is `cats.Monad`. `Monad` extends two other type classes: `FlatMap`, which provides the `flatMap` method, and `Applicative`, which provides `pure`. `Applicative` also extends `Functor`, which gives every `Monad` a `map` method

```scala
import cats.Monad
import cats.instances.option._

val opt1 = Monad[Option].pure(3) // opt1: Option[Int] = Some(3)
val opt2 = Monad[Option].flatMap(opt1)(a => Some(a + 2)) // opt2: Option[Int] = Some(5)
val opt3 = Monad[Option].map(opt2)(a => 100 * a) // opt3: Option[Int] = Some(500)
```

# Monad Syntax

The syntax for monads comes from three places:

- `cats.syntax.flatMap` provides syntax for `flatMap`
- `cats.syntax.functor` provides syntax for `map`
- `cats.syntax.applicative` provides syntax for `pure`

```scala
import cats.instances.option._   // for Monad
import cats.instances.list._     // for Monad
import cats.syntax.applicative._ // for pure

1.pure[Option] // res5: Option[Int] = Some(1)
1.pure[List]   // res6: List[Int] = List(1)
```

# The Identity Monad

The simplest monad is the Identity monad, which just annotates plain values and functions to satisfy the monad laws

```
def sumSquare[F[_]: Monad](a: F[Int], b: F[Int]): F[Int] = ???
sumSquare(3, 4) // error: no type parameters for method sumSquare ...
```

It would be incredibly useful if we could use sumSquare with parameters that were either in a monad or not in a monad at all. Cats provides the `Id` type to bridge the gap:

```
import cats.Id
sumSquare(3 : Id[Int], 4 : Id[Int]) // res1: Id[Int] = 25
```

# The Identity Monad

What's going on? Here is the definition of `Id` to explain:

```
package cats

type Id[A] = A
```

# Either

`Either` is a monad encapsulating two values of some types with a right bias for monadic transformations.

```
import cats.syntax.either._ // for asRight

val a: Either[String, Int] = Right(10) // a: Either[String, Int] = Right(10)
val b = 4.asRight[String]              // b: Either[String, Int] = Right(4)
for {
  x <- a
  y <- b
} yield x*x + y*y                      // res3: Either[String, Int] = Right(25)
```

# Either

## from other types

```scala
Either.fromTry(scala.util.Try("foo".toInt))
// res9: Either[Throwable, Int] = Left(
//   java.lang.NumberFormatException: For input string: "foo"
// )
Either.fromOption[String, Int](None, "Badness")
// res10: Either[String, Int] = Left("Badness")
```

# Transforming Eithers

- orElse and getOrElse to extract values from the right side or return a default

```
"Error".asLeft[Int].getOrElse(0) // res11: Int = 0
```

- ensure to check whether the right-hand value satisfies a predicate

```
-1.asRight[String].ensure("Must be non-negative!")(_ > 0) // Left("Must be non-negative!")
```

- recover and recoverWith methods provide error handling

```
"error".asLeft[Int].recover { case _: String => -1 } // Right(-1)
```

- leftMap and bimap methods to complement map

```
"foo".asLeft[Int].leftMap(_.reverse)      // res16: Either[String, Int] = Left("oof")
6.asRight[String].bimap(_.reverse, _ * 7) // res17: Either[String, Int] = Right(42)
```

- swap method lets us exchange left for right

```
123.asRight[String].swap // res20: Either[Int, String] = Left(123)
```

# Memoization

## memoized

Memoized computations are run once on first access, after which the results are cached.

`cats.Eval` is a monad that allows us to abstract over different *models of evaluation*.

We typically hear of two such models: `eager and lazy`. `Eval` throws in a further distinction of whether or not a result is *memoized*.

- `defs` are lazy and not memoized
- `vals` are eager and memoized
- `lazy vals` are lazy and memoized

# Eval

`Eval` has three subtypes: `Now`, `Later`, and `Always`. We construct these with three constructor methods, which create instances of the three classes and return them typed as `Eval`

```scala
import cats.Eval

val now = Eval.now(math.random + 1000)
// now: Eval[Double] = Now(1000.5540132858998)
val later = Eval.later(math.random + 2000)
// later: Eval[Double] = cats.Later@143718d9
val always = Eval.always(math.random + 3000)
// always: Eval[Double] = cats.Always@628dba99
```

# Chaining Evals

Like all monads, `Eval`'s map and `flatMap` methods add computations to a chain. In this case, however, the chain is stored explicitly as a list of functions. The functions aren't run until we call `Eval`'s `value` method to request a result

```
val greeting = Eval.
  always { println("Step 1"); "Hello" }.
    map { str => println("Step 2"); s"$str world" }
    greeting.value
    // Step 1
    // Step 2
    // res16: String = "Hello world"
```

# Memoizing Evals

`Eval` has a `memoize` method that allows us to memoize a chain of computations. The result of the chain up to the call to `memoize` is cached, whereas calculations after the call retain their original semantics:

```
val saying = Eval.
  always { println("Step 1"); "The cat" }.
  map { str => println("Step 2"); s"$str sat on" }.
  memoize.
  map { str => println("Step 3"); s"$str the mat" }
saying.value // first access // Step 1 // Step 2 // Step 3
// res19: String = "The cat sat on the mat" // first access
saying.value // second access // Step 3
// res20: String = "The cat sat on the mat"
```

# Deferred computations

One useful property of `Eval` is that its `map` and `flatMap` methods are *trampolined*.

This means we can nest calls to `map` and `flatMap` arbitrarily without consuming stack frames. We call this property “stack safety”.

```
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    Eval.defer(factorial(n - 1).map(_ * n))
  }

factorial(50000).value // res: A very big value
```

# Writer

`cats.data.Writer` is a monad that lets us carry a log along with a computation. We can use it to record messages, errors, or additional data about a computation, and extract the log alongside the final result.

```scala
val writer1 = for {
  a <- 10.pure[Logged]
  _ <- Vector("a", "b", "c").tell
  b <- 32.writer(Vector("x", "y", "z"))
} yield a + b

writer1.run // res3: (Vector[String], Int) = (Vector("a", "b", "c", "x", "y", "z"), 42)
```

# Reader

`cats.data.Reader` is a monad that allows us to sequence operations that depend on some input. Instances of `Reader` wrap up functions of one argument, providing us with useful methods for composing them.

One common use for `Readers` is dependency injection. If we have a number of operations that all depend on some external configuration, we can chain them together using a `Reader` to produce one large operation that accepts the configuration as a parameter and runs our program in the order specified.

# Reader

## example

```scala
import cats.data.Reader
case class Cat(name: String, favoriteFood: String)
val catName: Reader[Cat, String]     = Reader(cat => cat.name)
val greetKitty: Reader[Cat, String] = catName.map(name => s"Hello ${name}")
val feedKitty: Reader[Cat, String]  = Reader(cat => s"Have a ${cat.favoriteFood}")
val greetAndFeed: Reader[Cat, String] = for {
    greet <- greetKitty
    feed  <- feedKitty
  } yield s"$greet. $feed."

  greetAndFeed(Cat("Garfield", "lasagne")) // "Hello Garfield. Have a lasagne."
```

# State

`cats.data.State` allows us to pass additional state around as part of a computation. We define `State` instances representing atomic state operations and thread them together using `map` and `flatMap`. In this way we can model mutable state in a purely functional way, without using mutation. Boiled down to their simplest form, instances of `State[S, A]` represent functions of type `S => (S, A)`. `S` is the type of the state and `A` is the type of the result.

```scala
import cats.data.State
val a = State[Int, String] { state => (state, s"The state is $state") }
// a: State[Int, String] = cats.data.IndexedStateT@13a45d18
```

# State

## example

```scala
import cats.data.State
import State._
val program: State[Int, (Int, Int, Int)] = for {
  a <- get[Int]
  _ <- set[Int](a + 1)
  b <- get[Int]
  _ <- modify[Int](_ + 1)
  c <- inspect[Int, Int](_ * 1000)
} yield (a, b, c) // program: State[Int, (Int, Int, Int)]

val (state, result) = program.run(1).value
// state: Int = 3
// result: (Int, Int, Int) = (1, 2, 3000)
```

[1] Scala with Cats, November 2017, Copyright 2014-17 Noel Welsh and Dave Gurnell. CC-BY-SA 3.0.Artwork by Jenny Clements.Published by Underscore Consultg LLP, Brighton, UK.