Scala Course

Typeclasses

47 Deg

2021-02-03

Typeclasses

Facilitate polymorphism and abstraction. Unlike OO polymorphism, typeclasses allows us to expand the functionality of **existing types**.

In Java, if `String` doesn't implement the interface you want, you can't do anything. With typeclasses you can do it ;)

Typeclasses

With OO polymorphism we have 2 steps, interface and datatype declaration + implementation

```
trait Encoder {
  def encode: String
}

case class Car(brand: String) extends Encoder {
  def encode: String =
    s"""{"brand": "$brand"}"""
}

Car("Honda").encode
// res0: String = "{\"brand\": \"Honda\"}"
```

Typeclasses

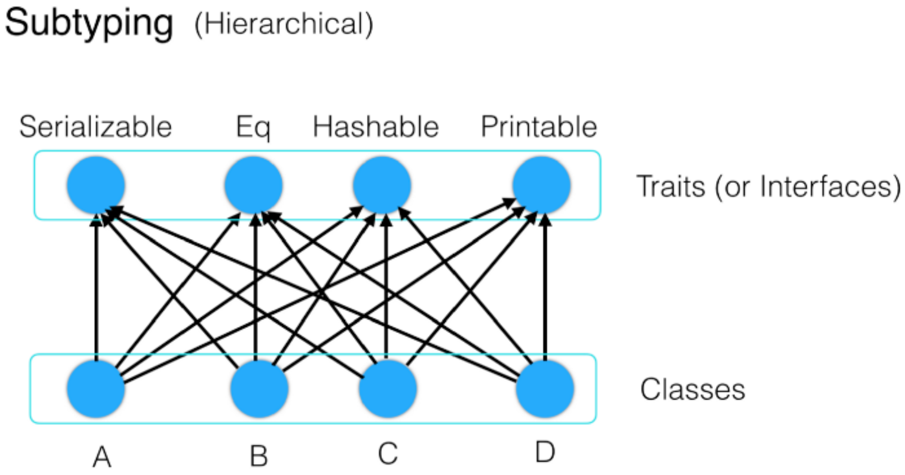


Figure 1: subtyting

Typeclasses

We will see how combining types of implicits
Type expansion + Type class pattern, we can expand the functionality.

Typeclasses

But first, we need to see **Typeclass** is

Typeclasses

Type Classes solve problems of OOP polymorphism using **parametric types** and **ad hoc polymorphism**.

This leads to less coupled and more extensible code

Ad hoc: When you need it

Typeclasses

Problems we want to avoid which arise using subtyping polymorphism:

- Classes coupling
- Complexity when adding new functionality to an already existing inheritance chain and existing class

Typeclasses

declaration of the typeclass

```scala
trait Encoder[A] {
  def encode(a: A): String
}
```

Type class pattern

And then, **Type class pattern**

Type class pattern

implementation

```scala
// we don't extend from the typeclass
case class Car(brand: String)

implicit val serializableCar: Encoder[Car] =
  new Encoder[Car] {
    def encode(car: Car): String =
      s"""{"brand": "${car.brand}"}"""
  }
// serializableCar: Encoder[Car] = repl.MdocSession$App1
```

Type class pattern

Usage

```scala
def encode[A](a: A)(implicit E: Encoder[A]): String =
  E.encode(a)

encode(Car("Honda"))
// res2: String = "{\"brand\": \"Honda\"}"
```

Typeclasses

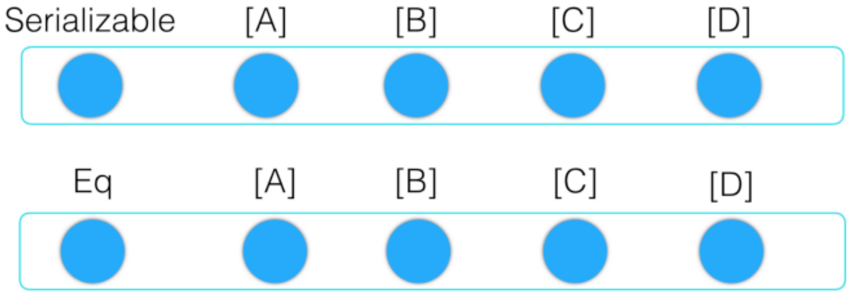
And combining with **Type expansion** with can have same as polymorphism.

```
implicit class Serializable[A](a: A) {
   def encode(implicit E: Encoder[A]): String =
     E.encode(a)
}

Car("Honda").encode
// res3: String = "{\"brand\": \"Honda\"}"
```

Typeclasses

Type Classes (Linear)



Hashable …

Context Bound

```scala
object Encoder {
  def apply[A](implicit E: Encoder[A]): Encoder[A] = E
}

implicit class SerializableCB[A: Encoder](a: A) {
   def encodeCB: String = Encoder[A].encode(a)
}

Car("Honda").encodeCB
// res4: String = "{\"brand\": \"Honda\"}"
```

Context Bound

They are the same signature

```
def encode[A: Encoder](a: A): String = ???
def encode[A](a: A)(implicit E: Encoder[A]): String = ???
```

Laws

Another important feature of typeclasses are laws.

Laws are properties that ensure that typeclass instances are correct.

Typeclasses

There are lots of typeclasses libraries for Scala, but we'll use `cats` in our examples.

Let me show you the most common

Semigroup

Semigroup has an associative binary operation

```scala
trait Semigroup[A] {
  def combine(a: A, b: A): A
}
```

Monoid

Monoids are semigroups that have an identity element. What's an identity element? one that used in combine has no effect.

```scala
trait Monoid[A] extends Semigroup[A] {
  def identity: A
  def combine(a: A, b: A): A
}
```

Monoid

What could be the `identity` element for `String`, `Int`, and `Boolean`?

Monoid

implementation

```scala
implicit val intSumMonoid: Monoid[Int] =
  new Monoid[Int] {
    def identity: Int = 0
    def combine(a: Int, b: Int): Int = a + b
  }
```

Monoid

implementation

```scala
implicit val stringMonoid: Monoid[String] =
  new Monoid[String] {
    def identity: String = ""
    def combine(a: String, b: String): String = a + b
  }
```

Monoid

implementation

```scala
implicit val booleanAndMonoid: Monoid[Boolean] =
  new Monoid[Boolean] {
    def identity: Boolean = true
    def combine(a: Boolean, b: Boolean): Boolean =
      a && b
  }
```

Monoid

Laws

Now that we have `identity` we can add a couple of more laws to Monoid:

```
sum(a, sum(b, c)) == sum(sum(a, b), c) // associativity
sum(a, identity) == a // right identity
sum(identity, a) == a // left identity
```

Eq

We need to be able to compare values of types, but the solution we got in the JVM was not perfect (`Object.equals`).

FP proposes a new way of comparing objects, the `Eq` typeclass:

```scala
trait Eq[A] {
  def eqv(a: A, b: A): Boolean
}
```

This approach to object equality has another benefit: trying to compare values of different types will result in a compiler error.

Show

The same happens with the string representation of values. Java tries to solve it with `Object.toString`, but that's not perfect either. We might not want to be able to print passwords, for example.

```scala
trait Show[A] {
  def show(a: A): String
}
```

Foldable

Is a typeclass whose type parameter is a type constructor that can be folded to produce a value.

```
trait Foldable[F[_]] {
  def foldLeft[A, B](
    fa: F[A], b: B
  )(f: (B, A) => B): B
}
```

Functor

Is a typeclass for type constructors that can be mapped over. Let's see how it's declared.

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

Applicative

Applicatives are Functors that have two features:

- can put pure values into its context
- can map a function lifted to its context over all its elements

```scala
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def ap[A, B](fn: F[A => B])(fa: F[A]): F[B]
}
```

Although the most using methods from `Applicative` are:

```scala
def traverse[G[_], A, B](fa:F[A])(f: A => G[B]): G[F[B]]
def sequence[G[_], A, B](fga: F[G[A]]): G[F[A]]
```

We can see `traverse` as `sequence` + map

Applicative

Traverse

```
import cats.implicits._
List("1","2","3").traverse(_.toIntOption)
// res5: Option[List[Int]] = Some(value = List(1, 2, 3))
```

Applicative

Traverse

```scala
import cats.implicits._
List("one","2","3").traverse(_.toIntOption)
// res6: Option[List[Int]] = None
```

Monad

Monads are Applicatives that can sequence operations with a `flatMap` method.

```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B](fn: A => F[B])(fa: F[A]): F[B]
}
```

Typeclasses

Laws

As we have already seen, Typeclasses provide us properties. This can be:

- **Recognizability**: when we see the use of `Semigroup` we'll know that it's an associative binary operation, regardless of the type.
- **Generality**: If we create a datatype, and we see it's `Semigroup`, we'll be able to use all functions that operate on `Semigroup`.

Typeclasses

All these are a simplified subset.
You can find more typeclasses in Cats library documentation.

Bonus track

For Comprehension

```
for {
  x <- List(1,2,3)
  y <- List(1,2,3)
} yield (x -> y)
// res7: List[(Int, Int)] = List(
//   (1, 1),
//   (1, 2),
//   (1, 3),
//   (2, 1),
//   (2, 2),
//   (2, 3),
//   (3, 1),
//   (3, 2),
//   (3, 3)
// )
```

For Comprehension

```
List(1, 2, 3)
    .flatMap(x =>
      List(1, 2, 3)
        .map(y => x -> y)
    )
// res8: List[(Int, Int)] = List(
//   (1, 1),
//   (1, 2),
//   (1, 3),
//   (2, 1),
//   (2, 2),
//   (2, 3),
//   (3, 1),
//   (3, 2),
```

Bonus track

For Comprehension

```scala
for {
  x <- List(1,2,3)
  y <- List(1,2,3)
} println(s"$x,$y")
// 1,1
// 1,2
// 1,3
// 2,1
// 2,2
// 2,3
// 3,1
// 3,2
// 3,3
```

Bonus track

For Comprehension

```scala
List(1, 2, 3)
    .foreach(x =>
      List(1, 2, 3)
        .foreach(y => println(s"$x,$y"))
    )
// 1,1
// 1,2
// 1,3
// 2,1
// 2,2
// 2,3
// 3,1
// 3,2
// 3,3
```

Exercise 5