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 ;)

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

```
scala> trait Serializable
defined trait Serializable

scala> case class Car(brand: String) extends Serializable
defined class Car
```

typeclasses add another step to polymorphism:

```
scala> trait Serializable[A] {
     |   def serialize(a: A): String
     | }
defined trait Serializable

scala> case class Car(brand: String) // we don't extend fr
defined class Car

scala> implicit val serializableCar: Serializable[Car] = ne
     |   def serialize(car: Car): String = s"""{"brand": "
     | }
serializableCar: Serializable[Car] = $anon$1@7c91083f
```

Abstraction

Let's try to identify a pattern in those functions:

```
scala> def sum(a: Int, b: Int): Int = a + b
sum: (a: Int, b: Int)Int

scala> def concat(a: String, b: String): String = a + b
concat: (a: String, b: String)String

scala> def and(a: Boolean, b: Boolean): Boolean = a && b
and: (a: Boolean, b: Boolean)Boolean
```

declaration

```
scala> trait Sumable[A] {
     |   def sum(a: A, b: A): A
     | }
defined trait Sumable
```

implementation

```
scala> implicit val intSumSumable: Sumable[Int] = new Sumab
     |   def sum(a: Int, b: Int): Int = a + b
     | }
intSumSumable: Sumable[Int] = $anon$1@358be1a9
```

implementation

```
scala> implicit val stringSumable: Sumable[String] = new Su
     |   def sum(a: String, b: String): String = a + b
     | }
stringSumable: Sumable[String] = $anon$1@1842f045
```

implementation

```
scala> implicit val booleanAndSumable: Sumable[Boolean] =
     |   def sum(a: Boolean, b: Boolean): Boolean = a && b
     | }
booleanAndSumable: Sumable[Boolean] = $anon$1@7bf4eec7
```

Laws

Another important feature of typeclasses are laws. Laws are properties that ensure that typeclass instances are correct.

For example, we can derive from our `Sumable` that it's associative.

```
sum(a, sum(b, c)) == sum(sum(a, b), c)
```

Typeclasses, together with laws, provide

- Recognizability: when we see the use of `Sumable` 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 `Sumable`, we'll be able to use all functions that operate on `Sumables`.

Using typeclasses

```scala
def needsTypeclassContextBound[A: Sumable] = ???
def needsTypeclassImplicit[A](implicit x: Sumable[A]) = ???
```

Typeclasses

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

We've already seen a very common typeclass in our examples, `Sumable`. It's normally called `Semigroup` in Functional programming.

Semigroup

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

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
     | }
defined trait Monoid
```

What could be the `identity` element for the three semigroups we created?

implementation

```
scala> implicit val intSumMonoid: Monoid[Int] = new Monoid
     |   def identity: Int = 0
     |   def combine(a: Int, b: Int): Int = a + b
     | }
intSumMonoid: Monoid[Int] = $anon$1@3a449be
```

implementation

```
scala> implicit val stringMonoid: Monoid[String] = new Mono
     |   def identity: String = ""
     |   def combine(a: String, b: String): String = a + b
     | }
stringMonoid: Monoid[String] = $anon$1@6dcda08f
```

implementation

```
scala> implicit val booleanAndMonoid: Monoid[Boolean] = ne
     |   def identity: Boolean = true
     |   def combine(a: Boolean, b: Boolean): Boolean = a
     | }
booleanAndMonoid: Monoid[Boolean] = $anon$1@2ddd7a28
```

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.

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

Monad

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

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

Since monads have a flatMap method, we can use any Monad[F] in a for comprehension!

Exercise 5