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 declaration+implementation

```
trait Serializable

case class Car(brand: String) extends Serializable
```

Typeclasses

typeclasses add another step to polymorphism:

```
trait Encoder[A] {
  def encode(a: A): String
}
case class Car2(brand: String) // we don't extend from the
implicit val serializableCar2: Encoder[Car2] = new Encoder
  def encode(car: Car2): String = s"""{"brand": "${car.bran
}
// serializableCar2: Encoder[Car2] = repl.Session$App$$anon
```

More Abstraction

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

```
def sum(a: Int, b: Int): Int = a + b
def concat(a: String, b: String): String = a + b
def and(a: Boolean, b: Boolean): Boolean = a && b
```

declaration

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

implementation

```
implicit val intSumSumable: Sumable[Int] = new Sumable[Int
  def sum(a: Int, b: Int): Int = a + b
}
// intSumSumable: Sumable[Int] = repl.Session$App$$anon$2@7
```

implementation

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 `Sumable`s.

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

```
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.

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

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

implementation

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

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
  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.

```
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

```
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.

```
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