Implementing Type classes

A *type class* is an abstract, parameterized type that lets you add new behavior to any closed data type without using sub-typing. This can be useful in multiple use-cases, for example:

- expressing how a type you don't own (from the standard or 3rd-party library) conforms to such behavior
- expressing such a behavior for multiple types without involving sub-typing relationships (one `extends` another) between those types (see: ad hoc polymorphism for instance)

Therefore in Scala 3, *type classes* are just *traits* with one or more parameters whose implementations are not defined through the `extends` keyword, but by given instances. Here are some examples of common type classes:

Semigroups and monoids

Here's the `Monoid` type class definition:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T

trait Monoid[T] extends SemiGroup[T]:
  def unit: T
```

An implementation of this `Monoid` type class for the type `String` can be the following:

```
given Monoid[String] with
  extension (x: String) def combine (y: String): String = x.concat(y)
  def unit: String = ""
```

Whereas for the type `Int` one could write the following:

```
given Monoid[Int] with
  extension (x: Int) def combine (y: Int): Int = x + y
  def unit: Int = 0
```

This monoid can now be used as *context bound* in the following `combineAll` method:

```
def combineAll[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))
```

To get rid of the `summon[...]` we can define a `Monoid` object as follows:

```
object Monoid:
  def apply[T](using m: Monoid[T]) = m
```

Which would allow to re-write the `combineAll` method this way:

```
def combineAll[T: Monoid](xs: List[T]): T =
  xs.foldLeft(Monoid[T].unit)(_.combine(_))
```

Functors

A `Functor` for a type provides the ability for its values to be "mapped over", i.e. apply a function that transforms inside a value while remembering its shape. For example, to modify every element of a collection without dropping or adding elements. We can represent all types that can be "mapped over" with `F`. It's a type constructor: the type of its values becomes concrete when provided a type argument. Therefore we write it `F[_]`, hinting that the type `F` takes another type as argument. The definition of a generic `Functor` would thus be written as:

```
trait Functor[F[_]]:
  def map[A, B](x: F[A], f: A => B): F[B]
```

Which could read as follows: "A `Functor` for the type constructor `F[_]` represents the ability to transform `F[A]` to `F[B]` through the application of function `f` with type `A => B`". We call the `Functor` definition here a *type class*. This way, we could define an instance of `Functor` for the `List` type:

```
given Functor[List] with
  def map[A, B](x: List[A], f: A => B): List[B] =
    x.map(f) // List already has a `map` method
```

With this `given` instance in scope, everywhere a `Functor` is expected, the compiler will accept a `List` to be used.

For instance, we may write such a testing method:

```
def assertTransformation[F[_]: Functor, A, B](expected: F[B], original: F[A], r
  assert(expected == summon[Functor[F]].map(original, mapping))
```

And use it this way, for example:

```
assertTransformation(List("a1", "b1"), List("a", "b"), elt => s"${elt}1")
```

That's a first step, but in practice we probably would like the `map` function to be a method directly accessible on the type `F`. So that we can call `map` directly on instances of `F`, and get rid of the `summon[Functor[F]]` part. As in the previous example of Monoids, `extension` methods help achieving that. Let's re-define the `Functor` type class with extension methods.

```
trait Functor[F[_]]:
  extension [A](x: F[A])
    def map[B](f: A => B): F[B]
```

The instance of `Functor` for `List` now becomes:

```
given Functor[List] with
  extension [A](xs: List[A])
    def map[B](f: A => B): List[B] =
      xs.map(f) // List already has a `map` method
```

It simplifies the `assertTransformation` method:

```
def assertTransformation[F[_]: Functor, A, B](expected: F[B], original: F[A], r
  assert(expected == original.map(mapping))
```

The `map` method is now directly used on `original`. It is available as an extension method since `original`'s type is `F[A]` and a given instance for `Functor[F[A]]` which defines `map` is in scope.

Monads

Applying `map` in `Functor[List]` to a mapping function of type `A => B` results in a `List[B]`. So applying it to a mapping function of type `A => List[B]` results in a

`List[List[B]]`. To avoid managing lists of lists, we may want to "flatten" the values in a single list.

That's where `Monad` comes in. A `Monad` for type `F[_]` is a `Functor[F]` with two more operations:

- `flatMap`, which turns an `F[A]` into an `F[B]` when given a function of type `A ⇒ F[B]`,
- `pure`, which creates an `F[A]` from a single value `A`.

Here is the translation of this definition in Scala 3:

```
trait Monad[F[_]] extends Functor[F]:

  /** The unit value for a monad */
  def pure[A](x: A): F[A]

  extension [A](x: F[A])
    /** The fundamental composition operation */
    def flatMap[B](f: A => F[B]): F[B]

    /** The `map` operation can now be defined in terms of `flatMap` */
    def map[B](f: A => B) = x.flatMap(f.andThen(pure))

end Monad
```

List

A `List` can be turned into a monad via this `given` instance:

```
given listMonad: Monad[List] with
  def pure[A](x: A): List[A] =
    List(x)
  extension [A](xs: List[A])
    def flatMap[B](f: A => List[B]): List[B] =
      xs.flatMap(f) // rely on the existing `flatMap` method of `List`
```

Since `Monad` is a subtype of `Functor`, `List` is also a functor. The Functor's `map` operation is already provided by the `Monad` trait, so the instance does not need to define it explicitly.

Option

`Option` is an other type having the same kind of behaviour:

```scala
given optionMonad: Monad[Option] with
  def pure[A](x: A): Option[A] =
    Option(x)
  extension [A](xo: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] = xo match
      case Some(x) => f(x)
      case None => None
```

Reader

Another example of a `Monad` is the *Reader* Monad, which acts on functions instead of data types like `List` or `Option`. It can be used to combine multiple functions that all need the same parameter. For instance multiple functions needing access to some configuration, context, environment variables, etc.

Let's define a `Config` type, and two functions using it:

```scala
trait Config
// ...
def compute(i: Int)(config: Config): String = ???
def show(str: String)(config: Config): Unit = ???
```

We may want to combine `compute` and `show` into a single function, accepting a `Config` as parameter, and showing the result of the computation, and we'd like to use a monad to avoid passing the parameter explicitly multiple times. So postulating the right `flatMap` operation, we could write:

```scala
def computeAndShow(i: Int): Config => Unit = compute(i).flatMap(show)
```

instead of

```scala
show(compute(i)(config))(config)
```

Let's define this m then. First, we are going to define a type named `ConfigDependent` representing a function that when passed a `Config` produces a `Result`.

```scala
type ConfigDependent[Result] = Config => Result
```

The monad instance will look like this:

```scala
given configDependentMonad: Monad[ConfigDependent] with

  def pure[A](x: A): ConfigDependent[A] =
```

```
    config => x

  extension [A](x: ConfigDependent[A])
    def flatMap[B](f: A => ConfigDependent[B]): ConfigDependent[B] =
      config => f(x(config))(config)

end configDependentMonad
```

The type `ConfigDependent` can be written using type lambdas:

```
type ConfigDependent = [Result] =>> Config => Result
```

Using this syntax would turn the previous `configDependentMonad` into:

```
given configDependentMonad: Monad[[Result] =>> Config => Result] with

  def pure[A](x: A): Config => A =
    config => x

  extension [A](x: Config => A)
    def flatMap[B](f: A => Config => B): Config => B =
      config => f(x(config))(config)

end configDependentMonad
```

It is likely that we would like to use this pattern with other kinds of environments than our `Config` trait. The Reader monad allows us to abstract away `Config` as a type *parameter*, named `Ctx` in the following definition:

```
given readerMonad[Ctx]: Monad[[X] =>> Ctx => X] with

  def pure[A](x: A): Ctx => A =
    ctx => x

  extension [A](x: Ctx => A)
    def flatMap[B](f: A => Ctx => B): Ctx => B =
      ctx => f(x(ctx))(ctx)

end readerMonad
```

Summary

The definition of a *type class* is expressed with a parameterised type with abstract members, such as a `trait`. The main difference between subtype polymorphism and ad-hoc polymorphism with *type classes* is how the definition of the *type class* is implemented, in relation to the type it acts upon. In the case of a *type class*, its implementation for a concrete type is expressed through a `given` instance definition, which is supplied as an implicit argument alongside the value it acts upon. With subtype polymorphism, the implementation is mixed into the parents of a class, and only a single term is required to perform a polymorphic operation. The type class solution takes more effort to set up, but is more extensible: Adding a new interface to a class requires changing the source code of that class. But contrast, instances for type classes can be defined anywhere.

To conclude, we have seen that traits and given instances, combined with other constructs like extension methods, context bounds and type lambdas allow a concise and natural expression of *type classes*.

Right-... Type Cl...