Type Classes Demystified

Vlad Patryshev

Why this topic?

This term is casually mentioned here and there, with only scarce definitions, which also vary from language to language. Wikipedia defines a type class as just an interface. If you are a Java programmer, you can close the Wikipedia tab right away and rest assured that you know what a type class is. You may be almost there... but once you switch to Scala, you discover that no, not only type class is not an interface or trait; this notion turns out to be obscure and complicated. Once I heard a two-hour talk on type classes in Scala, and in the end I asked the speaker, so, can you give us an exact definition of type classes? He could not.

Another Wikipedia article defines type classes as a tool to provide ad-hoc polymorphism. You may be unaware, what is it, ad-hoc polymorphism? It is very simple. If you have a function, and it is defined differently for different types of its arguments. Example: `1+2` and `"x"+"y"`. We have a polymorphism, but it is materialized by providing ad-hoc implementations. This is different from parametric polymorphism - e.g. you can define `map` on a `list` once, without bothering to redefine it for all kinds of lists you have, and from subtype polymorphism where different subtypes define different implementations.

Here I'm going to explain the notion of type classes as I understand them, and I insist on this understanding being the right one.

Let's start with the problems that many of you encountered many times, and which don't seem to have an easy, or any solution. Oh, and I'll be using Scala throughout the text; some of the examples are more generic, but Scala definitely has a very good solution for the problem of specifying type classes, so hang on.

Problems

Case 1. Have a set, calculate the sum of its elements. Seems to be simple, but how can we do it in a generic way? We cannot calculate the sum of a set of dates or phone numbers; there must be a function for adding up two elements of the set. So we have to delimit the type `T` in `Set[T]`. (The term "monoid" can be omitted for now.)
Case 2. Have a list, need to sort it. How do we know its elements can be ordered? Java has a practical answer: for `List<T>` to be sortable, you either need `T` extends `Comparable<T>`, or pass a comparator every time you want to sort.

Solutions

Already mentioned above,

Subtyping

The following code demonstrates how we could do it.

```
trait Comparable[T] { def ≤(other: T): Boolean }
trait StringOrder extends Comparable[String] {
  ...
}
trait Addable[T] { def +(other: T): T }
trait StringAddition extends Addable[String] {
 ...
}
class String extends StringOrder with StringAddition
```

Now we can add strings and sort strings. But then, first, we have to extend an existing class (Scala has a trick for doing this, called “pimp my library”), and second, how can we provide locale-specific order? How do we know if `"año" ≤ "ave"` - the answer depends on many things and may change with time, too.

Structural types

Take this simple case: we only want to deal with such data types that have `size`. Array is good, Set is good, String is almost good; Rectangle may be not so good, it has two sizes.

The Scala-specific solution to specify that a class has `size` method looks like this:

```
def summarySize(containers: {def size: Int}*) =
      (0/:containers)(_+_.size)
```

Here we require that a container the function accepts must have an `Int`-valued method `size`. Relying on this, we can add up sizes of all the containers without bothering of their other aspects, like in the following example:

```
summarySize(1::5::Nil, Set("x", "y", "z"), "Hello")
```

This gives us more freedom: we do not have to interfere with inheritance tree. Its applicability is limited by the fact that the method we call should be already defined under the same name. Would not work if we in one place use `+` to add, and in another `*` to multiply, and want to view it as essentially the same kind of operation.

Pimping

Aka “pimp my library” pattern. We use implicits to produce an instance of a rich class, given an instance of a poor class, like in this example:

```
trait Comparable[T] { def ≤(other: T): Boolean }
trait NaturalOrder extends Comparable[Being] {
  ...
}
implicit def order(b:Being): NaturalOrder = new NaturalOrder {...}
```

The class `Being` does not have any comparison defined on it; but when `≤` is called on its instance, the compiler provides a call of `order` method which creates a new instance of `BeingOrder`, which actually participates in the operation of comparison.

This works, but we have a couple of caveats. How exactly do we compare `"año"` and `"ave"`? What if the operation involves two types (like in multiplying a matrix by a scalar)?

A Better Solution

This of course involves type classes, but instead of jumping to the explanation of how this code works, let’s grow this solution out of what we have.

Refactoring to Type Class

Step 1

As a starting point we will take a solution that is very similar to Pimping:

```
trait Comparable[T] { def ≤(other: T): Boolean }

object SpanishStrings {
  val coll = Collator.getInstance(new Locale("ES"))
  def compare(a: String, b: String) = coll.compare(a,b)<0
}
```

```
implicit def order(a:String) = new Comparable[String] {
  def ≤(b: String) = SpanishStrings.compare(a, b)
}
```

Here I have an object, `SpanishStrings` that encapsulates collation order, and use this object to do the comparison.

Step 2

This object that hides the functionality is important for us, so we introduce a trait `CanCompare` to abstract its behavior. `SpanishStrings` now extends this trait.

```
trait CanCompare[T] { def compare(a:T, b:T): Boolean }
trait Comparable[T] { def ≤(other: T): Boolean }

object SpanishStrings extends CanCompare[String] {
  val coll = Collator.getInstance(new Locale("ES"))
  def compare(a: String, b: String) = coll.compare(a,b)
}

implicit def order(a:T, c: CanCompare[T]) =
  new Comparable[T] {
    def ≤(b: T) = c.compare(a, b)
  }
```

Step 3

To stop following Java pattern of passing around everything we have, I now declare `SpanishStrings` to be `implicit`:

```
trait CanCompare[T] { def compare(a:T, b:T): Boolean }
trait Comparable[T] { def ≤(other: T): Boolean }

implicit object SpanishStrings extends CanCompare[String] {
  val coll = Collator.getInstance(new Locale("ES"))
  def compare(a: String, b: String) = coll.compare(a,b)<=0
}
implicit def order(a:T)(implicit c: CanCompare[T]) = {
  new Comparable[T] {
    def ≤(b: T) = c.compare(a, b)
```

```
  }
}
```

This code will compile *only* if there is a *witness object* (in our case, `SpanishStrings`) that implements `CanCompare[T]` for a specific `T` (in our case, `String`).

Step 4

The last four lines, order method, can be equivalently rewritten like this (it is all syntactic sugar):

```
implicit def order[T:CanCompare](a:T) =
  new Comparable[T] {
    def ≤(b: T) = implicitly[CanCompare[T]].compare(a, b)
  }
```

Now we can check if `"año" ≤ "ave"`... but let's focus for a moment on this expression, `[T:CanCompare]`. This syntactic sugar can be equivalently expressed as the code in Step 3: requiring an existence of a witness object of type `CanCompare[T]`. Now that we do not have a name for that object, we have to mention it as `implicitly[CanCompare[T]]`.

Informally, the expression looks like we declare `T` to be an instance of a "higher level" type `CanCompare`. That's exactly our goal: we have introduced a ***type class*** `CanCompare`, and we require that `T` belongs to this class, similar to how `a:T` means that a belongs to type `T`.

We are done with refactoring.

So...

What Exactly is Type Class?

It is just a class of types, a subclass of the class of all types. Out of all the types there are, we find a way to specify only types having certain features.

How can we possibly do it, specify a certain class of types?

- Trait (interface): `class Record extends Serializable`
  Here we are saying that type `Record` belongs to a class of types that extends `Serializable`.

- Parameterized type, e.g. `List[_]`; a type that belongs to this class is any `List`, parameterized by list element type: `List[String], List[Int],`

`List[List[Set[Date]]]`.

- type class pattern, e.g. `def sum[M: Monoid](coll: Seq[M]) {...}` we provide witness object that ensures the functionality we need. In this sense it is, yes, a mechanism for ad-hoc polymorphism.

Of course, there may be other ways of specifying a subclass of types... not that I know of any.

Simple Example

Have various data structures, need to send them over as `JSON`; all we need is a method that converts, e.g., a `Record` to `JSON`, to cheat the library code. What we do not want is adding a jsonification method to types like `Record`, since it has absolutely nothing to do with the web, and if we break Demeter's law, she may complain to her son-in-law, Hades. I'd prefer not to.

```
trait JSON { def serialize: String }
def SendInfoToBigBrother(info: JSON)

class Record(name: String, phone: String, mail: String)...

trait CanBeJson[T] { def toJsonString(T): String }

implicit object RecordIsJSON extends CanBeJson[Record] {
 def toJsonString(rec: Record): String = ....
}

implicit def I_am_Json[T: CanBeJson](data: T) = new JSON {
 val converter = implicitly[CanBeJson[T]]
 def serialize = converter.toJsonString
}
```

The code looks familiar, right? We say `[T:CanBeJson]`, which requires the existence of a witness object implementing the method we need, `toJsonString`; and we provide this object for type `Record`. As a result, we can `SendInfoToBigBrother(new Record(...))`.

Inspiring Example. **`java.util.ArrayList`** is a **`Functor`**

This is probably simpler than you expect, in spite of the word *Functor*. Let's define it first:

```
trait Functor[F[_]] {
```

```
 def map[X,Y](f: X => Y): F[X] => F[Y]
}
```

This is a type class; it says that a parameterized type `F` is a functor when there is a method map that takes a function from `X` to `Y` and turns it to a function from `F[X]` to `F[Y]`. Of course all collections in Scala have map method, so they are functors; but how about `java.util.ArrayList`? Of course java library does not provide such a method. But we will.

```
implicit object JAL_a_Functor extends Functor[ArrayList] {
 def map[X,Y](f: X => Y) = (xs: ArrayList[X]) => {
   val ys = new ArrayList[Y]
   for (i <- 0 until xs.size) ys.add(f(xs.get(i)))
   ys
 }
}
```

This is our witness object, it has a method `map`. But it is just an object; let's continue with the transformer method that takes whatever there is, and gives us an instance that has a method `map` - if it can.

```
implicit def fops[F[_]: Functor, A](fa: F[A]) = new {
  val witness = implicitly[Functor[F]]
  final def map[B](f:A=>B):F[B] = witness.map(f)(fa)
}
```

The method specifies that `F` must belong to `Functor` type class. Then it takes an instance of `F[A]` for a given `A`, and returns an instance that has `map()`, that is; so it is a functor.

Here what it can produce:

```
val testList = new ArrayList(
                   Arrays.asList("this", "is", "a", "test"))
val transformed = testList.map(_.toUpperCase)
```

We will have `ArrayList("THIS", "IS, "A", "TEST")`

Wrapping Up

Scala is a moving target (and work in progress). While we can be struggling with digesting the notion of type classes, it is now ready to use implicit macros to automatically generate implementations; e.g. we would not need to write an implementation for json serialization for `Record`, it will be built automagically.

Type classes look different in different languages. In C++ they are known as concepts. Even

Scheme (http://wiki.call-cc.org/eggref/4/typeclass) and JavaScript (https://github.com/hallettj/etiquette.js) have an implementation of type classes.

Here are two most important links:

- P.Wadler, How to make ad-hoc polymorphims less ad hoc: http://tiny.cc/yomtuw
- M.Odersky, Poor Man's Type Classes: http://tiny.cc/ebmtuw