Type Classes in Scala

§

Made with Obsidian

Type deep-dive | Category computer-science | Technologies Scala, VS Code

Website Post Link

Formally, a type class is a type-system construct that supports ad hoc polymorphism. This is achieved by adding constraints to type variables in parametrically polymorphic types. In simpler terms, a type class is a construct that lets us add constraints to generic functions.

Type classes are useful when we have generic methods that may accept parameters of different types and/or return results of different types and would like to perform some kind of operation to our parameters without having to worry about explicitly defining the type-specific action ourselves. These constructs are widely used in type-drive programming since they provide flexibility while keeping our implementations safe.

In this Deep Dive, we'll discuss what type classes are, why they are useful, how they are implemented, and finally, go over some practical examples in order to solidify the theory discussed.

We'll be using Scala worksheets which can be found in the Deep Dive Repo.

§

Table of Contents

§

What are type classes?

In the context of Scala, a type class is a generic trait that comes with given instances for type instances of that trait. In a broader context, a type class is a construct that enables ad-hoc polymorphism, more commonly known as overloading. This is confusing, so let us first define a problem we're trying to solve and then provide a solution using type classes.

Let us imagine we have multiple integer lists and would like to define a method that adds all the elements inside the list and returns the result as an integer value:

Code

```scala
// Defining a type-specific function
def addIntegers(xs: List[Int]): Int = {
    xs.reduceLeft(_ + _)
}

val myListIntegers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

addIntegers(myListIntegers)
```

Output

```
res0: Int = 45
```

This is a very straightforward method, where, in one line, we define an operation that will add all the elements of our list and return the required value.

The problem is that this is not a generic function, meaning this will only work with lists of integer values. We can confirm this by trying to perform the same operation with a list of strings:

Code

```scala
val myListStrings: List[String] = List("1", "2", "3", "4", "5", "6", "7", "8", "9")

addIntegers(myListStrings)
```

And as expected, we get an incorrect type result:

Output

```
Found:    (MdocApp.this.myListStrings : List[String])
Required: List[Int]
```

We can solve this problem by declaring a generic method that expects a parameter of unknown type. This method would have a signature as such:

Code

```scala
addGeneric[T](xs: List[T]): T
```

Where:

- The type `T` signifies any possible type under our "*definition*" of what can be computed as a sum of elements.

But wait a minute, how are we to implement this function? How can we tell the Scala compiler that whenever we send a list of integers, it should compute the addition, but whenever we send a list of strings, it should concatenate the elements instead?

Well, here's where type classes come in handy, but first, we have to take a look at implicits.

§

What are implicits?

Implicits are a very powerful concept in Scala; they let the compiler **infer** certain values based on their type. This concept is confusing, so let us explain it with yet another example.

Let us imagine we have a very powerful function that computes the square of a given number:

Code

```
// Defining implicits
def myExplicitMethod(x: Int): Int =
    x * x

myExplicitMethod(7)
```

As expected, we get the square of 7:

Output

```
res0: Int = 49
```

But what happens when we slightly change this function by adding the `implicit` keyword to one of our arguments?

Code

```
def myInferringMethod(implicit x: Int): Int =
    x * x
```

Now, when we call our function, Scala will try to find a parameter that matches the input argument type in our `myInferringMethod` function, given that we also define an `implicit` variable alongside.

Code

```scala
def myInferringMethod(implicit x: Int): Int =
    x * x

implicit val myInt1: Int = 7

myInferringMethod
```

Output

```
res1: Int = 49
```

So, we did not need to explicitly call our function with an argument; the compiler noticed that we declared an implicit variable matching the type of our argument's signature and used that as input.

Of course, if we declare multiple implicit variables with the same type, Scala's compiler will complain, telling us that what we're doing results in an ambiguous evaluation; the compiler doesn't know which value to use for the implicit call.

This is not super useful when dealing with simple functions and variables. However, we can hopefully start seeing the connection between using type classes and implicits.

§

Type classes & implicits together

If we recall from our last examples, we have the following:

- A generic function that accepts a list of elements of type `T` as an argument.
- The need to add those elements, depending on their type.
- The possibility of using the `implicit` keyword lets the compiler infer which value to use.

Let us start by defining a `trait` that includes all the possible operations that can be done with the types we want to implement.

1. Defining a type class

Type classes are nothing more than traits with one or more implicit method definitions. We want to define a `trait` that extends the functionality of our original sum function safely, meaning we want to delimit the element types we can accept and have ready a well-defined method for each case:

Code

```
trait SumOfLists[T]:
    def sumElements(xs: List[T]): T

implicit object SumOfInts extends SumOfLists[Int]:
    override def sumElements(xs: List[Int]): Int = xs.reduceLeft(_ + _)

implicit object SumOfFloats extends SumOfLists[Double]:
    override def sumElements(xs: List[Double]): Double = xs.reduceLeft(_ + _)
implicit object SumOfStrings extends SumOfLists[String]:
    override def sumElements(xs: List[String]): String = xs.mkString("")
```

2. Ad-hoc polymorphism

Ad-hoc polymorphism is the fruit of type classes used in functions with implicit arguments. This might sound unintelligible, but it's not that hard once we include an example. Continuing from our sum function, we would now want to redesign it to accept the traits we just declared; this will finally create a truly generic function but with properly defined methods for each type case. And the best of all is that it will be dynamic, meaning the compiler will decide which method to use depending on the type we throw as an argument.

This is known as ad-hoc polymorphism. Formally, we can define it as a programming concept used to describe functions with the same name that are executed differently, depending on the variable or argument type. It is also referred to as “*function overloading*” or “*method overloading*".

Once we have our trait, we can define a polymorphic generic function:

Code

```
def addGeneric[T](xs: List[T])(implicit addElements: SumOfLists[T]): T =
    addElements.sumElements(xs)
```

Note that we don't need to specify any type whatsoever, except for `T`; the compiler handles the type inference for us, so that when we call our function using lists with elements of different types, we get the result we expect:

Code

```
val myListInts: List[Int] = List(1, 2, 3, 4, 5, 6)
val myListFloats: List[Double] = List(1.1, 2.2, 3.3, 4.4, 5.5, 6.6)
val myListStrings: List[String] = List("1", "2", "3", "4", "5", "6", "7", "8", "9")

addGeneric(myListInts)
addGeneric(myListFloats)
addGeneric(myListStrings)
```

Output

```
res3: Int = 21
res4: Double = 23.1
res5: String = 123456789
```

Also, note that this implementation is safe since if the compiler cannot find an implicit matching of the types of the elements inside the list, it will throw an error at compile time. In short, we have full control of what can and cannot be done inside our defined method:

Code

```
addGeneric(List(true, true, false))
```

Output

```
No given instance of type MdocApp.this.SumOfLists[Boolean] was found for parameter addElements of
method addGeneric in class MdocApp
```

This is horrible, right? So much work for a simple, flexible function. This is the price we must pay when using strongly statically typed languages. The rules are rigorous and make our code much more efficient in terms of execution and type safety. Still, we need to declare things differently than in more flexible languages such as Python, JavaScript, etc.

§

Some practical examples

Now that we have understood the very basics of type classes, we can proceed with some simple examples that will hopefully transmit why these constructs are extremely valuable when working with a strong statically-typed language like Scala.

1. Requesting data from an external API

Sometimes we work with external APIs and stumble upon poorly-documented ones where we don't fully know the return type of the object requested. If we're working with strong statically-typed languages such as Scala, this is our worst nightmare since one simple misalignment can completely break our program.

We can solve this by using a type class that acts according to the object type we're getting. But first, we must ask ourselves how our external data can be structured. What data structures could be used to transfer the data through this mysterious API?

Well, below are some ideas:

- `List`
  - A list of values of the same type, but we can have multiple lists of multiple types.
- `Array`
  - Same as with lists
- `Enum`
  - Can have multiple different types.
- `Map`
  - Can have keys associated with the values.

Great. So now we have infinite options to choose from, and this will take forever to write. The best approach here is to get at least some information about our object and try to build from there.

For the sake of this example, let us imagine that the developers of this external API actually provided some hints of the output structure. They said it might be one of the following:

- A `Map` of key-value pairs, where each key is of type `String`, and each value is an object of type `Array` with the following possible element types:
  - `Int`
  - `String`
  - `Double`
- A list of lists (*up to 2 levels of depth*), where each nested list can contain elements of the following type:
  - `Int`
  - `String`

So now our options are narrowed down; we know that we are expecting structures as such:

Code

```
// Example data structures
val exampleStructureOne = List(
    List(7, 14, 21, 28, 35),
    List(5, 1, 78, 43, 21)
    )

val exampleStructureTwo = List(
    List("Janusz", "Martha", "Emma"),
    List("Juarez", "Joan", "Dillon", "Leo")
)

val exampleStructureThree = Map(
    "Item 1" -> List(1, 2, 3, 4, 5),
    "Item 2" -> List(1, 2, 3, 4, 5),
    "Item 3" -> List(1, 2, 3, 4, 5),
    "Item 4" -> List(1, 2, 3, 4, 5),
    "Item 5" -> List(1, 2, 3, 4, 5)
)

val exampleStructureFour = Map(
    "Item 1" -> List("Virginia", "Woolf"),
    "Item 2" -> List("James", "Baldwin"),
    "Item 3" -> List("Thomas", "Hardy"),
    "Item 4" -> List("Clarice", "Lispector"),
    "Item 5" -> List("Edward", "Gibbon")
)
```

This sounds fairly reasonable. So, we need to worry about how we are destructuring these possibilities, and how are we letting the compiler know which possibility to expect.

Well, we can start with a type class as a first step:

Code

```
// Define a type class
trait parseData[T, S]:
    def extractData(obj: T): S

implicit object parseListInt extends parseData[List[List[Int]], List[Int]]:
    override def extractData(obj: List[List[Int]]): List[Int] = obj.flatten(x => x)

implicit object parseListString extends parseData[List[List[String]], List[String]]:
    override def extractData(obj: List[List[String]]): List[String] = obj.flatten(x => x)

implicit object parseMapInt extends parseData[Map[String, List[Int]], List[Int]]:
    override def extractData(obj: Map[String, List[Int]]): List[Int] = obj.values.flatMap(x =>
x).toList

implicit object parseMapString extends parseData[Map[String, List[String]], List[String]]:
    override def extractData(obj: Map[String, List[String]]): List[String] = obj.values.flatMap(x
=> x).toList
```

Here, we're covering four possibilities:

- `ListInt`
- `ListString`
- `MapInt`
- `MapString`

And we're flattening the structures so that we get a one-dimensional `List` object in return. So, the only thing left is to declare our extractor method:

Code

```
def getExtData[T, S](r: T)(implicit extractMethod: parseData[T, S]): S =
    extractMethod.extractData(r)
```

And now, we call it with our example structures:

Code

```
getExtData(exampleStructureOne)
getExtData(exampleStructureTwo)
getExtData(exampleStructureThree)
getExtData(exampleStructureFour)
```

Output

```
res6: List[Int] = List(7, 14, 21, 28, 35, 5, 1, 78, 43, 21)
res7: List[String] = List(Janusz, Martha, Emma, Juarez, Joan, Dillon, Leo)
res8: List[Int] = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
res9: List[String] = List(Edward, Gibbon, Virginia, Woolf, Clarice, Lispector, Thomas, Hardy,
James, Baldwin)
```

We can imagine that this precise signature (*that of the trait*) would probably not be the best approach if we're trying to extract data from an API return object since we're effectively losing any dimensionality or structuring the API provider organized for us.

The ideal approach here would be to build another Map, for example, containing all homogenized structures (*instead of a list of values*), but the point is made; this construct is extremely versatile and powerful, and most importantly, it's safe.

2. Implementing an ordering function for two classes

Sometimes we must abstract certain types that the language does not provide by default. For example, we can divide integer numbers by performing a division operation, but we cannot directly express its rational form (*i.e., its fraction form*). To do this, we can create a new class and assign specific operations typically associated with a rational number.

The problem is that when we define a specific method for that class, we might also want to use it in other classes.

For example, we can define an ordering operation that orders rational numbers in ascending or descending order. We might also want to order some other custom type, for example, a list of addresses; these are two totally unrelated objects. However, in real life, we sometimes encounter this scenario where we would like to use one method for many objects and would like to delegate the decision of how to apply each method to the compiler.

Let us start by defining our classes:

Code

```
// Define classes
case class Rational(numer: Int, denom: Int)
case class Address(name: String, addr: String)
```

Great. Now, we need to define our type class (*trait*):

Code

```
trait Ordering[T]:
    def order(x: List[T]): List[T]
```

Very nice. Now, let us start defining variations of our `compare` method for each of the two options:

- Rational numbers
- Address books

Code

```
implicit object orderAddresses extends Ordering[Address]:
    override def order(x: List[Address]): List[Address] =
        x.sortBy(y => y._2)

implicit object orderRationals extends Ordering[Rational]:
    override def order(x: List[Rational]): List[Rational] =
        x.sortBy(y => y.numer/y.denom)
```

Lastly, let us define a function for doing the comparison, as well as some class instances:

Code

```
def sortElements[T](element: List[T])(implicit orderingMethod: Ordering[T]): List[T] =
    orderingMethod.order(element)

sortElements(myAddressBook)
sortElements(myRationals)
```

Output

```
res10: List[Address] = List(Address(Paul,Budapest), Address(Emma,Stockholm),
Address(John,Vienna))
res11: List[Rational] = List(Rational(1,2), Rational(7,11), Rational(5,4))
```

So, in the first case, we're sorting ascendingly by the person's location, starting from Budapest and ending in Vienna. However, in the second case, we're actually performing a division operation, comparing the resulting floats, and then using that as a sorting parameter.

This is not the most elegant way of sorting rational numbers; we can play a little bit with numerators and denominators, specifically denoting something as such:

```
val xn = x.numer * y.denom
val yn = y.numer * x.denom
```

Where `x` & `y` are two rational numbers, and if `xn` > `yn`, then `x` is larger than `y`. However, that's not the point here. The main takeaway is that type classes are extremely flexible, even for two totally unrelated objects; the trick is knowing how to represent each operation as a separate method, and the rest is history.

§

Conclusions

In this segment, we discussed what type classes are, what problems they solve when talking about statically typed languages, and why they are useful. We then discussed how type classes can be used in conjunction with implicits and what role type classes play in ad-hoc polymorphism. We closed this segment by introducing a couple of examples that could be extrapolated to real-life situations.

Type classes are an elegant construct present in various languages such as C++, Haskell, Rust, and other more esoteric languages such as Idris & Agda. Type classes let us write code in a flexible way that is still safe.

§

References

- Type Classes in Scala, Baeldung

- Type Classes, Scala Documentation
- Haskell Wiki, OOP vs Type Classes

§

Copyright