www.autoscout24.com

Show && Tell Type Classes

16.02.2016 Matthew Lloyd

A Quick Reminder - Implicits

Implicit Parameters – We can provide parameters to a method by specifying the parameters to be implicitly provided:

```
{
  implicit val intX: Int = 5

  def implicitExample(n: Int)(implicit x: Int) = {
    n + x
  }

  implicitExample(5) // 10
}
```

The intX is provided to the method automatically as it is available within scope.

A Quick Reminder – Implicits

Implicit Conversion – We can provide methods to implicitly convert between types:

```
{
  import scala.language.implicitConversions

  implicit def intToString(n: Int): String = n.toString

  def needsString (s: String) = "I see: " + s

  needsString(6)
}
```

By bringing in the intToString method to scope, we can call needsString with an int.

What are Typeclasses?

A Type Class is an 'interface' that defines behaviour.

Allows us to define new behaviour for any type.

Not to be confused with traditional 'classes'.

Typeclass doesn't belong to the traditional hierarchy of types.

Instead of extending you create 'instances' for types.

What can I use them for?

Cross cutting concerns that are not type specific for example:

Serializing types to be transferred (e.g. Json / XML)

Equality of types

Converting to and from String

Where do we start?

Let's take Equality as an example. We might want to compare Different types to see if they are equal. E.g:

```
6.eql(RomanNumeral("VI"))
RomanNumeral("X").eql(10)
```

However Int doesn't have any way of comparing between Ints and Roman Numerals...

So how do we add functionality?

Well first we need to 'enhance' our integers.
So we need to create a 'wrapper' for Ints:

```
trait WrappedInt {
  def eql(other: RomanNumeral): Boolean
}
```

So we have a WrappedInt that defines the operation, now lets define an implicit conversion for Ints:

```
object WrappedInt {
  implicit def IntToWrappedInt(v: Int): WrappedInt = new WrappedInt {
    def eql(other: RomanNumeral): Boolean = other.value == v
  }
}
```

So how do we add functionality?

This now means now through implicit conversion we can do the following:

```
6.eql(RomanNumeral("VI"))
```

Which is the same as:

```
WrappedInt.IntToWrappedInt(6).eq(RomanNumeral("VI"))
```

How do we make it more generic?

So lets take a step back and think about a more generic Equals trait:

```
trait Equals[A, B] {
  def equal(a: A, b: B): Boolean
}
```

Simple enough we take 2 things and return a boolean if they are the same.

How do we make it more generic?

Let's make our WrappedInt use the Equals trait instead:

```
object WrappedInt {
  implicit def IntToWrappedInt(v: Int)(implicit tc: Equals[Int, RomanNumeral]): WrappedInt = new WrappedInt {
    def eql(other: RomanNumeral): Boolean = tc.eql(v, other)
  }
}
```

We pull in an implementation of Equals as an implicit parameter.

We can then use this to implement our eql method.

The key part here is the implementation of equality is external.

So what did we just do?

We have created a new trait to abstract out the idea of equality between any two items. **Equals is our Type Class**.
A representation of equality that can work for any type.

Our WrappedInt class gives us a way to **promote** standard integers **into** the Type Class.

This is done via implicit conversions and implicit parameters.

implicit conversion adds the new operations.
implicit parameters bring the typeclass instance in to scope.

But WrappedInt is silly, can't that be generic too?

Yes it can, but we need to think of it slightly differently.

Instead of it 'wrapping an Int'

It should be 'adding operations to a value'

```
trait EqualsOps[A, B] {
  def self: A
  implicit def tc: Equals[A, B]
  final def eql(other: B): Boolean = tc.equal(self, other)
}

object EqualsOps {
  implicit def aToEquals[A, B](v: A)(implicit instance: Equals[A, B]) = new EqualsOps[A, B] {
    def self = v
    implicit def tc = instance
  }
}
```

Okay, so I can add methods to any other type...
But where does the Equals implementation come from?

To make things easier, we can define a helper method:

```
object Equals {
  def makeInstance[A, B](f: (A, B) => Boolean): Equals[A, B] = new Equals[A, B] {
    def equal(a: A, b: B): Boolean = f(a, b)
  }
}
```

Okay, so I can add methods to any other type...
But where does the Equals implementation come from?

This enables us to easily create a new Equals implementation like so:

```
implicit val intRomanNumeralEquals: Equals[Int, RomanNumeral] =
  Equals.makeInstance((n: Int, rn: RomanNumeral) => n == rn.value)
```

These implementations of Equals are known as **Type Class Instances**

A Quick Recap

```
trait Equals[A, B] {
  def equal(a: A, b: B): Boolean
}
```

Equals is our **Type Class.**

It allows us to define operations that we can add to other types.

Type Classes exist **outside** the existing Type Hierarchy.

A Quick Recap

```
trait EqualsOps[A, B] {
  def self: A
  implicit def tc: Equals[A, B]
  final def eql(other: B): Boolean = tc.equal(self, other)
}

object EqualsOps {
  implicit def aToEquals[A, B](v: A)(implicit instance: Equals[A, B]) = new EqualsOps[A, B] {
    def self = v
    implicit def tc = instance
  }
}
```

Operations Objects **elevate values into our Type Class**.

This is the glue that joins the values with our type specific implementations.

A Quick Recap

```
trait EqualsOps[A, B] {
  def self: A
  implicit def tc: Equals[A, B]
  final def eql(other: B): Boolean = tc.equal(self, other)
}

object EqualsOps {
  implicit def aToEquals[A, B](v: A)(implicit instance: Equals[A, B]) = new EqualsOps[A, B] {
    def self = v
    implicit def tc = instance
  }
}
```

Operations Objects **elevate values into our Type Class**.

This is the glue that joins the values with our type specific implementations.

Note: The final keyword is important!

A Quick Recap

```
implicit val intRomanNumeralEquals: Equals[Int, RomanNumeral] =
  Equals.makeInstance((n: Int, rn: RomanNumeral) => n == rn.value)

implicit val intHexNumeralEquals: Equals[Int, HexNumeral] =
  Equals.makeInstance((n: Int, rn: HexNumeral) => n == rn.value)
```

Type Class Instances are the implementations for types.

You can have **multiple** instances per Typeclass

Single vs Multi parameter Type classes

Single Parameter Type Classes are said to define a **Set of Types**.
E.g. toString, toJson, fromJson, Integral, Fractional, Float etc...

Multi Parameter Type Classes are said to define a **Relationship between Types**.
E.g. Equality, Ordering, Addition, Concatenation etc...

Taking it further...

```
trait Equals[A, B] {
  def equal(a: A, b: B): Boolean = !notEqual(a, b)
  def notEqual(a: A, b: B): Boolean = !equal(a, b)
}
```

You can define Type Classes to have **self-referential implementations**.

When you define your Type Class instance, you only need to override equal, and you get notEqual for free!

Taking it further...

```scala
import scala.language.higherKinds

trait HigherKindedEqualsOps[A, B, M[_]] {
  def self: M[A]
  implicit def tc: Equals[M[A], M[B]]
  final def eql(other: M[B]): Boolean = tc.equal(self, other)
  final def neql(other: M[B]): Boolean = tc.notEqual(self, other)

  final def ===(other: M[B]): Boolean = eql(other)
  final def =/=(other: M[B]): Boolean = neql(other)
}
```

By defining **Higher Kinded** Operations Objects you can compare container types.

Taking it further...

```
implicit def ListEquals[A, B](implicit tc: Equals[A, B]): Equals[List[A], List[B]] = EqHelper.single({
  case (Nil, Nil) => true
  case (as, bs) if as.length == bs.length => (as zip bs).forall { case (a, b) => a === b }
  case _ => false
})
```

Which means you can implement generic comparisons between **Lists** or **Options**.

```
Option(227).eql(Option(HexNumeral("0xE3")))

List(227).eql(List(HexNumeral("0xE3")))
```

Taking it further...

```
implicit def ListEquals[A, B](implicit tc: Equals[A, B]): Equals[List[A], List[B]] = EqHelper.single({
  case (Nil, Nil) => true
  case (as, bs) if as.length == bs.length => (as zip bs).forall { case (a, b) => a.eql(b) }
  case _ => false
})
```

Which means you can implement generic comparisons between **Lists** or **Options**.

```
Option(227).eql(Option(HexNumeral("0xE3")))
```

```
List(227).eql(List(HexNumeral("0xE3")))
```

Note how this accepts the implementation for Equals[A, B]