TYPE CLASS ...

... AND THE WAY IT'S DONE IN SCALA

... AND HEY, JAVA HAS IT TOO

By Kai Chen / kai@sorrentocorp.com

REVIEW: IMPLICITS IN SCALA

1. Implicit Conversion
2. Implicit Parameters
3. Keyword *implicit* can appear before
   - val
   - def
   - object
   - class

RIGHT ARROW

```scala
val map1 = Map("a" -> 1)

// needs a Tuple2[String, Int]
// also String doesn't have a '->' method
val map2: Map[String, Int] = Map.apply(???)

// because we have in scala.Predefs
@inline implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
    new ArrowAssoc(x)

// "a" is converted to any2ArrowAssoc[String]("a")
val map3: Map[String, Int] = Map.apply(
    (any2ArrowAssoc[String]("a")).->(1))

// which is converted to new ArrowAssoc[String]("a")
val map4: Map[String, Int] = Map.apply(
    (new ArrowAssoc[String]("a")).->(1))
```

SYNTAX SIMULATION!

A.K.A PIMP MY LIBRARY PATTERN

STREAM CONS

```
val fib: Stream[Int] = {
    def loop(h: Int, n: Int): Stream[Int] = h #:: loop(n, h + n)
    loop(1, 1)
}

// in Stream.scala
implicit def consWrapper[A](stream: => Stream[A]): ConsWrapper[A] =
    new ConsWrapper[A](stream)

// ConsWrapper has a cons operator ...
class ConsWrapper[A](tl: => Stream[A]) {
  def #::(hd: A): Stream[A] = cons(hd, tl)
  def #:::(prefix: Stream[A]): Stream[A] = prefix append tl
}

// call-by-name enables us to implement lazy evaluation
def loop(h: Int, n: Int): Stream[Int] =
          (new ConsWrapper[Int](loop(n, h + n))).#::(h)

// which enventually turns our code into
def loop(h: Int, n: Int): Stream[Int] = new Cons(h, loop(n, h + n))

// where the tail isn't evaluated until invoked
```

ORDERING

```scala
val list = List(1,3,5,2,4,6)
list.sorted

val map = Map("a" -> 1L, "b" -> 2L, "Bill" -> 81000000000l)
map.toList.sorted

// defined in collection.SeqLike.scala
def sorted[B >: A](implicit ord: Ordering[B]): Repr = { ... }
// Ordering[T] is an implicit parameter

implicit def Tuple2[T1, T2](
      implicit ord1: Ordering[T1], ord2: Ordering[T2]):
          Ordering[(T1, T2)] = ...
// Ordering is defined for the tuple only when
// Ordering is defined for each element
```

TYPE CLASS!

MORE TIDBITS

- Implicit conversion can obscure code
- Helper method implicitly[T]
- Implicit object, implicit class
- Simulate control structure (loan pattern)

WHAT IS A TYPE CLASS?

- Grew out of design of Haskell in 1988 by Philip Wadler
- Off-the-shelf solution for arithmetic and equality
- Ad-hoc polymorphism ≈ overloading
  - Not parametric polymorphism (e.g. List[T].length)
  - Function defined over several different kinds of types
  - Acting in a different way for each type
- Membership is open-ended

EXAMPLES IN HASKELL

```
class Ord a where
    (≤)   ::    a → a → Bool


class Show a where
    show  ::    a → String


class Read a where
    read  ::    String → a


class Eq a where
    (==)  ::    a -> a -> Bool
```

SO WHAT?

```
class Ord a where                    public interface Comparable< T > {
    (≤)   ::    a → a → Bool             public int compareTo(T o);
                                     }

class Show a where                   public String toString () {
    show  ::    a → String               return getClass().getName() + ...
                                     }

class Read a where                   public static int parseInt (String s) {
    read  ::    String → a               return parseInt(s, 10);
                                     }

class Eq a where                     public boolean equals (Object obj) {
    (==)  ::    a -> a -> Bool           return (this == obj);
                                     }
```

WHAT IS IT GOOD FOR?

- Behavior only where it makes sense
  - Type-safe equality
  - Recursive and composable equality
- Retroactive extension
  - JSON or XML Serialization
  - Extend third-party libraries
- Multiple behavior implementation, e.g. sorting

EQUALITY

- Apples should only be compared to other Apples

  ```
  if ( apple == orange ) {...}
  ```

  should generate a compiler error
- Recursive and composable equality function

  Scala provides this for case classes
- Can still get simple referential equality if needed

  Probably via a method call not named '=='

RETROACTIVE EXTENSION

- It's like adding methods in the bytecode
- Retrofit existing types to work in AwesomeMatrix
  - won't work if AW is designed with IMatrixable
  - easy if it takes a type class MatrixElem[T]
- Provide alternative algorithms
  - java.lang.Comparator.compare(a, b)
  - not possible if all we have is java.lang.Comparable.compareTo(o)

TANGENTIAL

- Algebraic Data Types @ Type-Level
- Concept-based Generic Programming
- Not the same as Extension Object
- Not the same as IAdaptable (in Eclipse)
  - In IAdaptable, extension is anticipated
  - IAdaptable uses dynamic dispatch
  - But type class method dispatch is always static

SCALA MECHANICS FOR TYPE CLASS

1. Define the behaviors in a trait

   and it must have a parameterized type

2. Provide a companion object

   it could extend from the trait if it makes sense

3. Package implicits in the companion

   These are types for which
   the required behaviors
   have already been implemented

4. Module works with this trait

SCALA EXAMPLE 1

SCALA.MATH.ORDERING

```
trait Ordering[T] extends Comparator[T] with ... Equiv[T] with ... {
    def compare(x: T, y: T): Int

    override def equiv(x: T, y: T): Boolean = compare(x, y) == 0
    override def lteq(x: T, y: T): Boolean = compare(x, y) <= 0
    ...

    class Ops(lhs: T) {
      def <=(rhs: T) = lteq(lhs, rhs)
      def max(rhs: T): T = Ordering.this.max(lhs, rhs)
      ...
    }

    implicit def mkOrderingOps(lhs: T): Ops = new Ops(lhs)
}
```

EXAMPLE 1 (CONT'D)

ORDERING COMPANION

```
object Ordering extends LowPriorityOrderingImplicits {
    def apply[T](implicit ord: Ordering[T]) = ord
    def by[T, S](f: T => S)(implicit ord: Ordering[S]): Ordering[T] =
      fromLessThan((x, y) => ord.lt(f(x), f(y)))

    trait StringOrdering extends Ordering[String] {
      def compare(x: String, y: String) = x.compareTo(y)
    }
    implicit object String extends StringOrdering

    trait LongOrdering extends Ordering[Long] {
      def compare(x: Long, y: Long) =
        if (x < y) -1 else if (x == y) 0 else 1
    }
    implicit object Long extends LongOrdering
}
```

SCALA EXAMPLE 2

SCALAZ.ORDER, ORDERING

```
trait Order[F] extends Equal[F] { self =>
    def order(x: F, y: F): Ordering
}

object Ordering extends OrderingInstances with OrderingFunctions {
  case object LT extends Ordering(-1, "LT") { def complement = GT }
  case object EQ extends Ordering(0,  "EQ") { def complement = EQ }
  case object GT extends Ordering(1,  "GT") { def complement = LT }
}

trait Order[F] ... {
    ...
    implicit def orderMonoid[A] = new Monoid[Order[A]] { ... }
}
```

EXAMPLE 2 (CONT'D)

TYPE CLASS INSTANCES

```
implicit val intInstance:
        Monoid[Int] with Enum[Int] with Show[Int] =
    new Monoid[Int] with Enum[Int] with Show[Int] {
  override def shows(f: Int) = f.toString

  def append(f1: Int, f2: => Int) = f1 + f2
  def zero: Int = 0

  def order(x: Int, y: Int) = if (x < y) Ordering.LT
      else if (x == y) Ordering.EQ else Ordering.GT

  def succ(b: Int) = b + 1
  def pred(b: Int) = b - 1
  ...
  override def min = Some(Int.MinValue)
  override def max = Some(Int.MaxValue)
}
```

SCALA EXAMPLE 3

PLAY'S JSON API LIBRARY

```
sealed trait JsValue
case object JsNull extends JsValue
class JsUndefined(err: => String) extends JsValue
case class JsBoolean(value: Boolean) extends JsValue
case class JsNumber(value: BigDecimal) extends JsValue
case class JsString(value: String) extends JsValue
case class JsArray(value: Seq[JsValue] = List()) ...
case class JsObject(fields: Seq[(String, JsValue)]) ...

object Json {
  def parse(input: String): JsValue = ...

  def toJson[T](o: T)(implicit tjs: Writes[T]): JsValue = ...

  def fromJson[T](json: JsValue)
      (implicit fjs: Reads[T]): JsResult[T] = ...
}
```

EXAMPLE 3 (CONT'D)

... AND THE TYPE CLASS IS

```scala
trait Reads[A] {
    def reads(json: JsValue): JsResult[A]
}

trait Writes[-A] {
    def writes(o: A): JsValue
}

implicit object IntReads extends Reads[Int] { ... }
implicit val DefaultJodaDateReads = jodaDateReads("yyyy-MM-dd")
implicit object JsObjectMonoid extends Monoid[JsObject]

// macro implemented for case class serialization
implicit val myCaseClassFmt = Json.format[myCaseClass]
```

TYPE CLASS EXAMPLE 4: SCALAZ.MONOID

```
implicit def orderMonoid[A] = new Monoid[Order[A]] {
  def zero: Order[A] = new Order[A] {
    def order(x: A, y: A): Ordering = Monoid[Ordering].zero
  }
  def append(f1: Order[A], f2: => Order[A]): Order[A] =
      new Order[A] {
      def order(x: A, y: A): Ordering =
      Semigroup[Ordering].append(f1.order(x, y), f2.order(x, y))
  }
}


final class SemigroupOps[F](val self: F)
  (implicit val F: Semigroup[F]) extends Ops[F] {
  final def |+|(other: => F): F = F.append(self, other) // ...
}
```

CONCLUSION

- For Library Developers
  - Probably superior to using just an interface
  - Fits perfectly in composable and chainable processing
    Monadic systems, servlet filtering
- For Library Users
  - Systems designed with type class are MUCH easier
    if the provided type class instances are sufficient
  - May need to provide one's own type class instance
    Good to know because you will end up debugging it