

Scala Polymorphic Object-oriented

By Modestas R

```
trait Ordering[T] {  
  def compare(a: T, b: T): Int  
}
```

```
case class Version( major: Int, minor: Int  
  override def toString: String = s"v$majo  
}
```

```
object Version {  
    implicit val ord = new Ordering[Version]  
  
    def compare(a: Instant, b: Instant): Int = {  
        val diff1 = a.major - b.major  
  
        if( diff1.isZero ) {  
            val diff2 = a.minor - b.minor  
  
            if( diff2.isZero ) a.patch - b.patch  
            else diff2  
        } else diff1  
    }  
}
```

```
implicit class OrderingOps[T](a: T)(implicit ord: Ordering[T]) {
  def >(b: T): Boolean = ord.compare(a, b) > 0
  def <(b: T): Boolean = ord.compare(a, b) < 0
  def >=(b: T): Boolean = !ord.compare(a, b) < 0
  def <=(b: T): Boolean = !ord.compare(a, b) > 0
  def max(b: T): T = if( >(b) ) a else b
  def min(b: T): T = if( <(b) ) a else b
}
```

Ordering[Version]



```
graph LR; A[Ordering[Version]] --> B[ ]
```

```
> Version(11,2,3) > Version(3,2,1)
true
```

```
> Version(3,2,3) < Version(3,2,1)
false
```

```
> Version(11,2,3) >= Version(3,2,1)
false
```

```
> Version(11,2,3) <= Version(3,2,1)
true
```

```
> Version(11,2,3) max Version(3,2,1)
"v11.2.3"
```

```
> Version(11,2,3) min Version(3,2,1)
"v3.2.1"
```

```
sealed abstract class List[T] {  
  def min(implicit ord: Ordering[T]): T  
  def max(implicit ord: Ordering[T]): T  
  
  def sort(implicit ord: Ordering[T]): List[T]  
}
```



```

case class Cons[T](head: T, tail: List[T]) extends List[T]
  def min(implicit ord: Ordering[Nothing]): T =
    tail.fold(head)(_ min _)

  def max(implicit ord: Ordering[Nothing]): T =
    tail.fold(head)(_ max _)

  def sort(implicit ord: Ordering[T]): List[T] =
    val (smalls, bigs) = tail.partition(_ < head)
    smalls.sort ++ List(head) ++ bigs.sort
}

case object Nil extends List[Nothing] {
  def min(implicit ord: Ordering[Nothing]): Nothing =
    throw new NoSuchElementException("Cannot find minimum")
  def max(implicit ord: Ordering[Nothing]): Nothing =
    throw new NoSuchElementException("Cannot find maximum")

  def sort(implicit ord: Ordering[Nothing]): List[Nothing] =
    Nil
}

```

Ordering[Version]



```
> val versions = List( Version(11,2,3), Version(3
```

```
> versions.min
```

```
v1.22.3
```

```
> versions.max
```

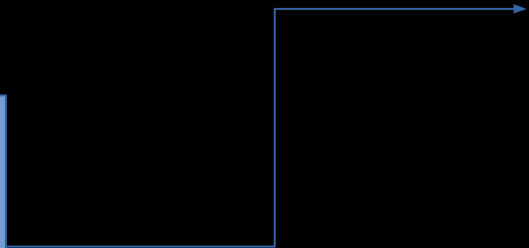
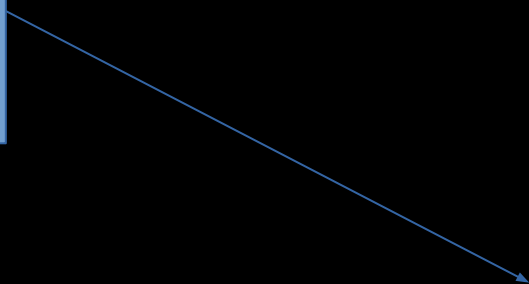
```
v11.2.3
```

```
> versions.sort
```

```
List(v1.22.3, v3.2.1, v11.2.3)
```

Ordering[A]

Ordering[B]



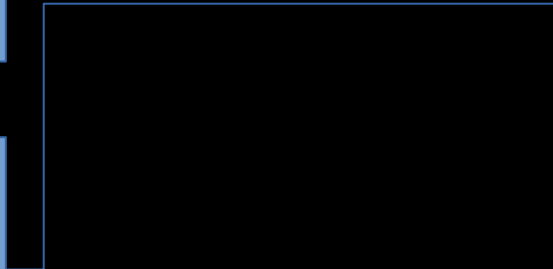
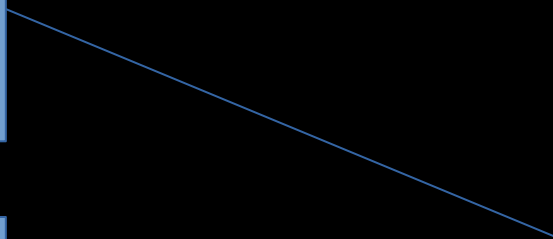
```
implicit def orderingPair[A, B](implicit
  ordA: Ordering[A],
  ordB: Ordering[B]
): Ordering[(A, B)] = new Ordering {

  def compare(lhs: (A, B), rhs: (A,B)): Int = {
    val diff = ordA.compare( lhs._1, rhs._1)
    if( diff.isZero ) ordB.compare( lhs._2, rhs._2)
    else diff
  }
}
```

Ordering[A]

Ordering[B]

Ordering[C]



```
object Ordering {  
  def by[A, B](f: A => B)(implicit ord: Orderi  
    new Ordering {  
      def compare(lhs: A, rhs: A): Int =  
        ord.compare( f(lhs), f(rhs) )  
    }  
}
```

```
object Version {  
  implicit val ord = Ordering by unapply  
}
```



```
trait Numeric[N] {  
  def add(a: N, b: N): N  
  def subtract(a: N, b: N): N  
  def multiply(a: N, b: N): N  
  def divide(a: N, b: N): N  
  
  def zero: N  
  def unit: N  
}
```

```
case class Complex( re: Double, im: Double

  override def toString: String = {
    val sign = if (im.isNegative) "" else
s"$re$sign{$im}i"
  }
}
```

```
object Complex {
  implicit num = new Numeric {

    def add(a: Complex, b: Complex): Complex = Complex( a.re +

    def subtract(a: Complex, b: Complex): Complex = Complex(

    def multiply(a: Complex, b: Complex): Complex = Complex(
      re = a.re * b.re - a.im * b.im,
      im = a.re * b.im + a.im * b.re
    )

    def divide(a: Complex, b: Complex): Complex = {
      val abs = b.re * b.re + b.im * b.im
      Complex(
        (a.im * b.im - a.re * b.re) / abs,
        (a.re * b.im + a.im * b.re) / abs
      )
    }

    def zero: Complex = Complex(0, 0)
    def unit: Complex = Complex(1, 0)
  }
}
```

```
implicit class NumericOps[N](a: N)(implicit num: Numeric[N]) {  
  def +(b: N): Boolean = num.add(a, b)  
  def -(b: N): Boolean = num.subtract(a, b)  
  def *(b: N): Boolean = num.multiply(a, b)  
  def /(b: N): Boolean = num.divide(a, b)  
}
```

Numeric[Complex]



```
graph LR; A[Numeric[Complex]] --> B[ ]
```

```
> Complex(11,2) + Complex(3,2)  
14+4i
```

```
> Complex(3,2) - Complex(3,2)  
0+0i
```

```
> Complex(3,4) * Complex(4,3)  
25+0i
```

```
> Complex(4,-3) / Complex(3,4)  
0-1i
```

```
sealed abstract class List[T] {  
    def sum(implicit num: Numeric[T]): T = fold(nu  
    def prod(implicit num: Numeric[T]): T = fold(n  
}
```

```
> val nums = List( Complex(3,4), Complex(4,3), Co
```

```
> nums.sum
```

```
7+7.2i
```

```
> nums.prod
```

```
0+5i
```



```
trait Arbitrary[T] {  
  val gen: Gen[T]  
}
```

Arbitrary[Version]

Arbitrary[(Version, String)]

Arbitrary[String]

