Basics

`Comparable` / `Ordered`

In java there's a `Comparable` interface that defines a `compareTo` method used for sorting/comparing objects.

For the unaware, comparing in Java works as follows - the `compareTo` method returns a value x such that: $x > 0 \iff this > that$,

$x < 0 \iff this < that$

and $x = 0$ otherwise.

On the other hand Scala, defines a `Ordered` trait that extends the Java `Comparable` interface. Because mixin composition in Scala is possible, the `Ordered` trait defines some additional concrete methods that depend on the `compare` method.

```
trait Ordered[T] extends Comparable[T] {
  abstract def compare(that: T): Int
  def <(that: T): Boolean = // ...
  def <=(that: T): Boolean = // ...
  def >(that: T): Boolean = // ...
  def >=(that: T): Boolean = // ...
  def compareTo(that: T): Int = // ...
}
```

By default the `compareTo` method is an alias for the abstract `compare` method (this is mostly because in Scala &lt 2.8 `Ordered` didn't extend Comparable).

As you can see, `Ordered[T]` in scala gives us comparison methods and we don't have to resort to ugly `this.compareTo(that) > 0` kludges.

Anyway, let's define a simple `Value` case class that holds integers:

```
case class Value(i: Int) extends Ordered[Value] {
  def compare(that: Value) = this.i - that.i
}
```

Because `Value` extends `Ordered` we can use methods that expect a natural ordering of objects - look at this interpreter session:

```
scala> val valueList = List(Value(1), Value(2), Value(3), Value(2), Value(1))
valueList: List[Value] = List(Value(1), Value(2), Value(3), Value(2), Value(1))
scala> valueList.sorted

res0: List[Value] = List(Value(1), Value(1), Value(2), Value(2), Value(3))
```

```
scala> valueList.min
res1: Value = Value(1)
```

By contrast, we can't sort on classes not defining the `Ordered` trait:

```
scala> case class UnOrderedValue(i: Int)
scala> val unOrderedValueList = List(UnOrderedValue(1), UnOrderedValue(2),
     | UnOrderedValue(3), UnOrderedValue(2), UnOrderedValue(1))
unOrderedValueList: List[UnOrderedValue] = List(UnOrderedValue(1), UnOrderedVal
ue(2), UnOrderedValue(3), UnOrderedValue(2), UnOrderedValue(1))
scala> unOrderedValueList.sorted
<console>:11: error: No implicit Ordering defined for UnOrderedValue.
              unOrderedValueList.sorted
scala> unOrderedValueList.min
<console>:11: error: No implicit Ordering defined for UnOrderedValue.
              unOrderedValueList.sorted
```

`Ordering` / `Comparator`

Java/Scala also defines another kind of abstraction which is used to compare two values. In java this interface is called `Comparator` and in scala it's `Ordering`:

```
trait Ordering[T] extends Comparator[T] {
  abstract def compare(x: T, y: T): Int
  // concrete methods
}
```

`Ordering` can be thought as a more general interface then `Ordered` - you can have many `Ordering`s available for a class, which for `Ordered` isn't possible without subtyping.

Generic sorting

Let's create a generic container:

```
trait Box[T] {
  def value: T
}
```

We can sort on it using this sort class:

```
class Sort[T](ordering: Ordering[Box[T]]) {
  def apply(boxes: Seq[Box[T]]) = {
    boxes.sorted(ordering)
  }
}
```

For `Sort` to work, we have to define an `Ordering` on `Box`es:

```
class BoxOrdering[T](ordering: Ordering[T]) extends Ordering[Box[T]] {
  def compare(x: Box[T], y: Box[T]): Int = ordering.compare(x.value, y.value)
}
```

Now we can define a trivial box type:

```
case class IntBox(value: Int) extends Box[Int]
```

And we can sort sequences of `IntBox` using:

```
scala> val list = List(IntBox(1), IntBox(2), IntBox(3), IntBox(2), IntBox(1))
list: List[IntBox] = List(IntBox(1), IntBox(2), IntBox(3), IntBox(2), IntBox(1))

scala> val sort = new Sort(new BoxOrdering(scala.math.Ordering.Int))
sort: Sort[Int] = Sort@5e0e9cd0

scala> sort(list)
res5: Seq[Box[Int]] = List(IntBox(1), IntBox(1), IntBox(2), IntBox(2), IntBox(3))
```

This isn't good as we can make it more general, but now let's talk implicits.

Implicits 101

Implicit parameters

An implicit parameter `T` of a method, can be omitted when the argument can be deduced by the compiler - when an `implicit` instance of the type `T` is in scope.

Trivial example for `Int`s:

```
scala> def add(x: Int)(implicit y: Int) = x + y
add: (x: Int)(implicit y: Int)Int

scala> add(3)(4)
res9: Int = 7

scala> implicit val x: Int = 4
x: Int = 4

scala> add(3)
res10: Int = 7
```

Implicit definitions

Another type of implicits are implicit definitions which are used to create conversions between objects of type `A` to `B`. If we have a method that expects a parameter of type `B` we can give it an instance of `A` if there exists a implicit method in scope:

```
implicit def aToB(a: A): B = ???
```

Concretely, lets go back and modify the `IntBox` class giving it a `+` method:

```
case class IntBox(value: Int) extends Box[Int] {
  def +(other: IntBox) = IntBox(value + other.value)
}
```

If we wanted `+` to work on ints we could extend the functionality by defining an implicit conversion between `Int` and `IntBox`

```
implicit def intToIntBox(int: Int): IntBox = IntBox(int)
```

Now both versions will work:

```
scala> val a = IntBox(3)
a: IntBox = IntBox(3)

scala> val b = IntBox(4)
b: IntBox = IntBox(4)

scala> a + b.value
res16: IntBox = IntBox(7)

scala> a + b
res17: IntBox = IntBox(7)
```

Sorting on Ordered

Coming back to our `Box` example - the scala library defines an implicit conversion between `Ordered[T]` and `Ordering[T]` and vice-versa. We can use this to define a universal sorting function on `Box`es:

```
object boxSort {
  def apply[T](boxes: Seq[Box[T]])(implicit ordering: Ordering[T]) = {
    val boxOrdering = new BoxOrdering(ordering)
    boxes.sorted(boxOrdering)
  }
}
```

And now, sorting works as expected:

```
scala> val list = List(IntBox(1), IntBox(2), IntBox(3), IntBox(2), IntBox(1))
list: List[IntBox] = List(IntBox(1), IntBox(2), IntBox(3), IntBox(2), IntBox(1))

scala> boxSort(list)
res18: Seq[Box[Int]] = List(IntBox(1), IntBox(1), IntBox(2), IntBox(2), IntBox(3))
```

Type classes

I've written a little post about typeclasses (`Ordering` is in fact a type class) you can check it out here

Edit history

2013-03-28

- Fixed typos/spelling errors.
- Added link to "Polymorhpism, and type classes in Scala."

2016-03-15

- Fixed the code snippet in the `Ordering`/`Comparator` section - thanks to Marcus Bosten for pointing that out

Home

You can contact me at: bartosz.witkowski@like-a-boss.net