Ad hoc Polymorphism using Type Classes

wholly based on 'How to write like Cats', a great talk by Monty West

Monty West
https://www.linkedin.com/in/monty-west/

YouTube How to write like Cats by Monty West https://github.com/MontyWest/tech-talk-typeclass

Lightweight, modular, and extensible library for functional programming

slides by @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

Subtype polymorphism
↓
Ad-hoc Polymorphism (Type classes)

Monty West

https://www.linkedin.com/in/monty-west/

The talk is going to focus on, firstly, subtype polymorphism, the classic OOP approach to functionality and stuff, and we are going to move from that, we are going to show that briefly, and then we are going to change it to ad hoc polymorphism using type classes.

And we are going to do this all in the scope of sorting, sorting a list of something, in this case Ints, but your humble sort function, so it should be familiar ground for most people.

```
def sortInts(ls: List[Int]): List[Int]
```

And it is all going to be **live coding**, as well, so I have definitely set myself up for failure on my first talk, but wish me luck.

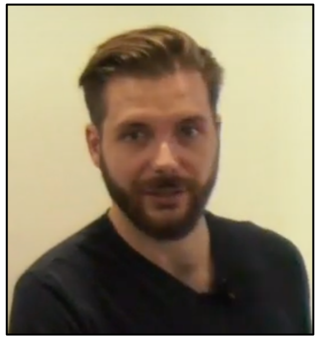
Ok, so, first of all, five points for whoever can name the sort.

```
package object ops {
  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs =>
      sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }
}
```

Yes, Quicksort. So I have this sort, it just sorts Ints, kind of useful, but I have this new class that I want to use, in a List say, a `Person` class, and I have defined this myself.

```
final case class Person(age: Int, name: String)
```

So how do I sort `Persons`?

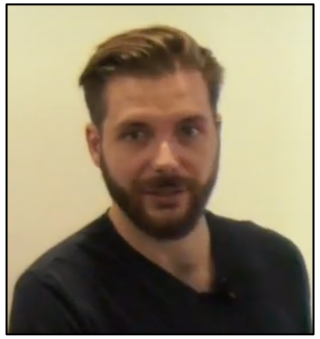


Monty West

Obviously **we could just define a sort function for persons**, but it wouldn't be useful for anything else so as we are all good programmers, **we want to be more generic**, **we want to be able to apply this sort function to an A, an A type**.

So this kind of looks like this:

```
package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil ⇒ Nil
    case _ :: Nil ⇒ ls
    case x :: xs ⇒
      sortInts(xs.filter(a ⇒ a < x)) ++ List(x) ++ sortInts(xs.filterNot(a ⇒ a < x))
  }

  def sort[A](ls: List[A]): List[A] = ls match {
    case Nil ⇒ Nil
    case _ :: Nil ⇒ ls
    case x :: xs ⇒
      sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
  }

}
```



Done, right? Finished, great! So **not quite**: **we don't know how to compare As together, we don't have this less than operation**. We don't have anything like that. **The compiler doesn't know what A is, it can't infer that**.

Monty West
https://www.linkedin.com/in/monty-west/

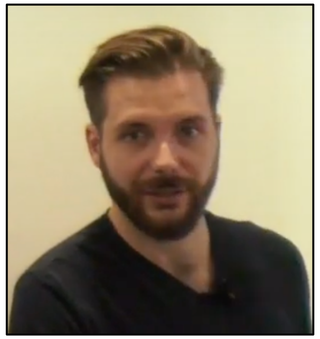
So how would we approach this in a sort of **subtype polymorphism** way?

Well, the first thing we do is we **add an interface**, so I am going to add a **trait**. I am going to call it **Orderable**, and **the intention of this trait is to be a supertype of whatever we want to order**. So in this case it is going to be As but we'll add it to the **Person** class, and what it needs to be able to do is **compare** to another instance of A…

```scala
trait Orderable[A] {

  /**
    * this < other : negative
    * this = other : 0
    * this > other : positive
    *
    * @param other
    * @return
    */
  def compare(other: A): Int

  def <(other: A): Boolean = compare(other) < 0

}
```

I am also going to have a quick **helper** because we saw it just there, we are going to add a **less than function** as well, and it is just going to return a **Boolean**, and that is going to just use our **compare** function.

That should be familiar. That's pretty much how the **Comparable** interface in **Java** works.



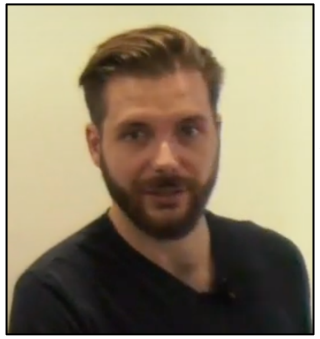
Monty West

So we are going to use this in our sort.

```
def sort[A](ls: List[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
}
```

we import **Orderable** and we introduce a **type bound** for our A

```
def sort[A <: Orderable[A]](ls: List[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
}
```



And that should compile, **so now, anything that is Orderable can be sorted by this function.**

Monty West
https://www.linkedin.com/in/monty-west/

So we have to make our Person class Orderable and we do that by **extending and implementing** Orderable:

```
final case class Person(age: Int, name: String) extends Orderable[Person] {

  /**
    * this < other : negative
    * this = other : 0
    * this > other : positive
    *
    * @param other
    * @return
    */
  override def compare(other: Person): Int = ???

}
```

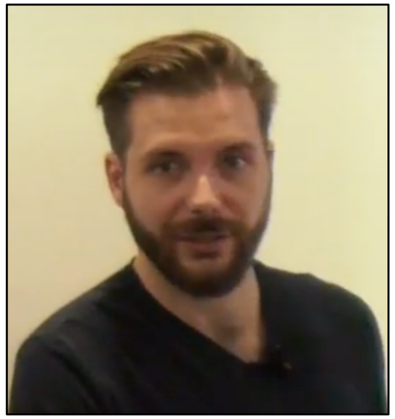
We have a choice to make here about how we sort Persons. **I am just going to choose age because it is the easiest one to use** but **it is noteworthy that we have to make a choice, we have to define the way we want to order Persons and put it on the actual class**.

So in this case it is going to be this:

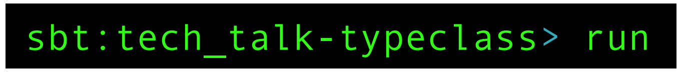
```
  override def compare(other: Person): Int = this.age - other.age
```

The other thing to notice here is that we use '**this**', this is how we compare two things, we have access to our **current** and then we have **another one** passed into this function.

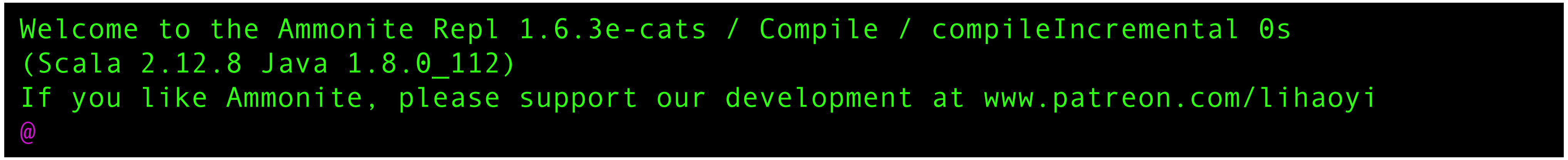
OK, so let's actually use this, let's hope this all compiles. So I am going to run.



Monty West

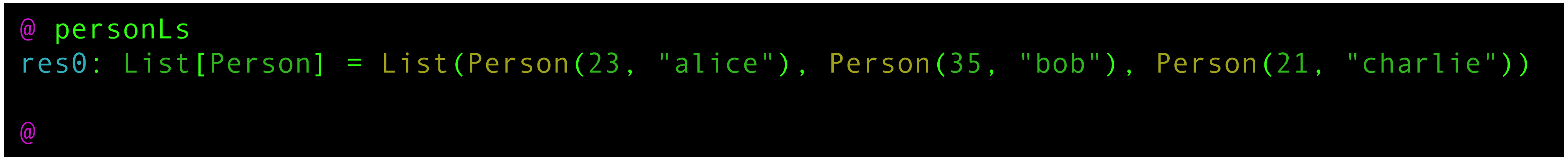


```
sbt:tech_talk-typeclass> run
```



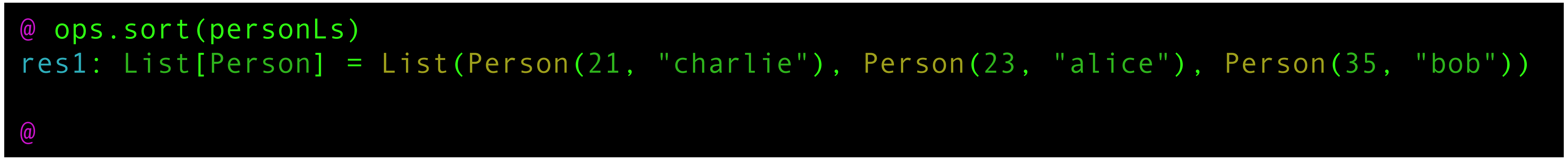
```
Welcome to the Ammonite Repl 1.6.3e-cats / Compile / compileIncremental 0s
(Scala 2.12.8 Java 1.8.0_112)
If you like Ammonite, please support our development at www.patreon.com/lihaoyi
@
```

so I have pre-prepared a list of **persons** here



```
@ personLs
res0: List[Person] = List(Person(23, "alice"), Person(35, "bob"), Person(21, "charlie"))

@
```

and now I can call my sort function with this list and it should sort them

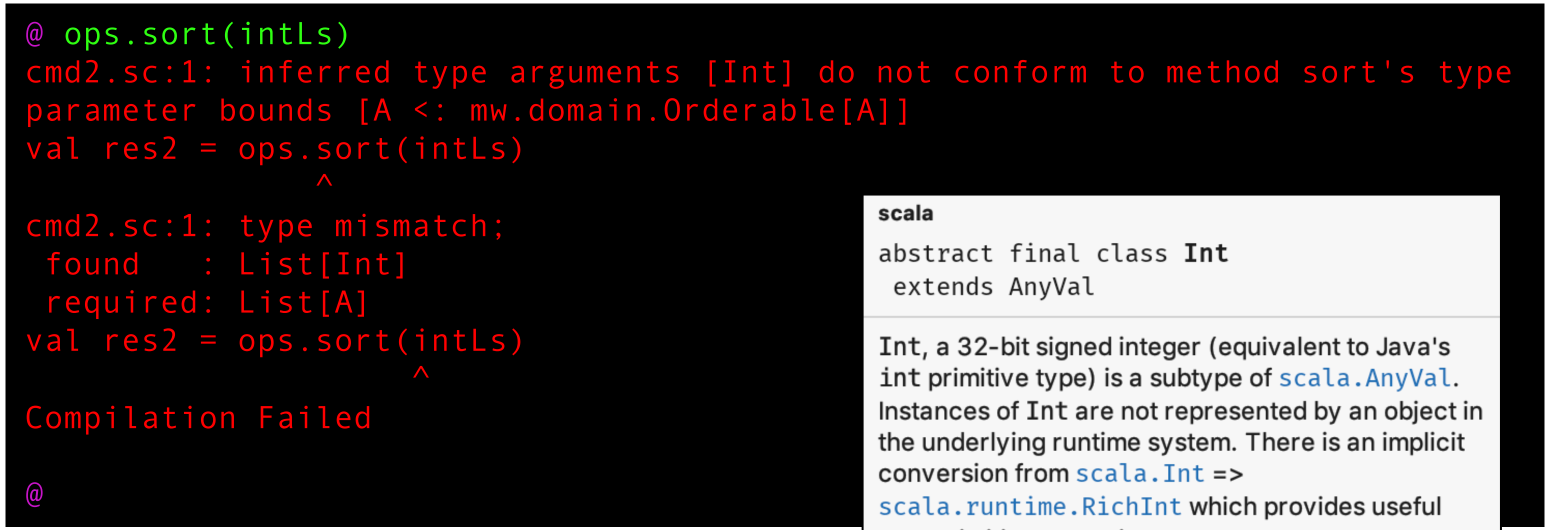


```
@ ops.sort(personLs)
res1: List[Person] = List(Person(21, "charlie"), Person(23, "alice"), Person(35, "bob"))

@
```

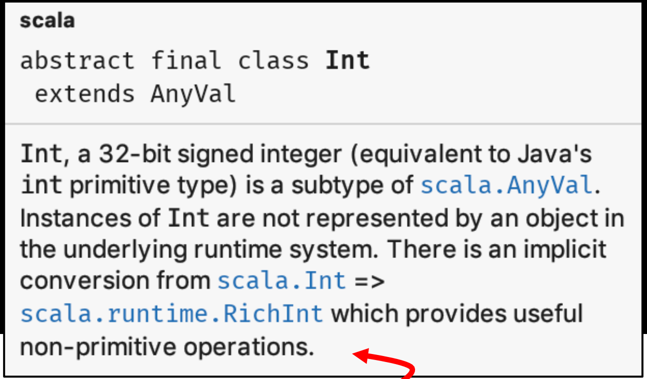
Cool, job done.

Monty West
https://www.linkedin.com/in/monty-west/

So the problem here is that **I also have an Int List and say I want to sort that**, **I want to use my generic sort because I only want to maintain one function and obviously I can't do that because Int is not implementing my interface and if I pass it in it will give me a big error**:



```
@ ops.sort(intLs)
cmd2.sc:1: inferred type arguments [Int] do not conform to method sort's type
parameter bounds [A <: mw.domain.Orderable[A]]
val res2 = ops.sort(intLs)
                  ^
cmd2.sc:1: type mismatch;
 found   : List[Int]
 required: List[A]
val res2 = ops.sort(intLs)
                        ^
Compilation Failed

@
```



scala

```
abstract final class Int
 extends AnyVal
```

Int, a 32-bit signed integer (equivalent to Java's int primitive type) is a subtype of `scala.AnyVal`. Instances of Int are not represented by an object in the underlying runtime system. There is an implicit conversion from `scala.Int` => `scala.runtime.RichInt` which provides useful non-primitive operations.

So **that's not great and also we can't go and fix this**. **We can't go into the Int class of Scala/Java and add this `Orderable` trait and implement it**. **We don't really have any option to, we can't order Ints**.

The other thing is, **if we want to change the sort of `Persons`**, **if we want to sort by name for example, or if we want to reverse the sort**, **we have to actually go into the implementation of `Person` and change that comparison function**, **which is annoying**, **to have to go all the way down into our actual class that's just holding data and change something**.

```
final case class Person(age: Int, name: String)
extends Orderable[Person] {

  /**
    * this < other : negative
    * this = other : 0
    * this > other : positive
    *
    * @param other
    * @return
    */
  override def compare(other: Person): Int =
    this.age - other.age

}
```

```scala
import mw.domain.Orderable

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A <: Orderable[A]](ls: List[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
  }
}
```

```
@ ops.sort(personLs)
```

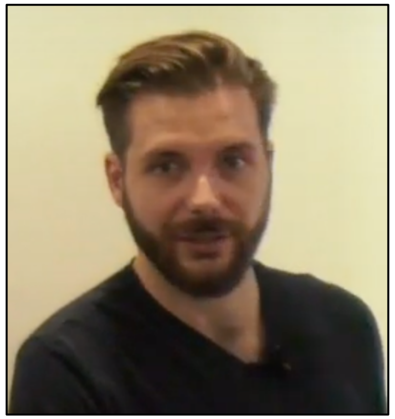
Recap - Version 1 of the code: Subtype **Polymorphism**

@philip_schwarz

```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```scala
final case class Person(age: Int, name: String) extends Orderable[Person] {

  /**
    * this < other : negative
    * this = other : 0
    * this > other : positive
    *
    * @param other
    * @return
    */
  override def compare(other: Person): Int = this.age - other.age

}
```

```scala
trait Orderable[A] {

  /**
    * this < other : negative
    * this = other : 0
    * this > other : positive
    *
    * @param other
    * @return
    */
  def compare(other: A): Int

  def <(other: A): Boolean = compare(other) < 0

}
```



Monty West

So, what can we do about it?

This is where I am going to introduce type classes and ad hoc polymorphism.

So I am just going to **delete all that** . So **let's go all of the way back to when we had this, this sort of like failed generic sort with nothing on it**.

```
def sort[A](ls: List[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
}
```

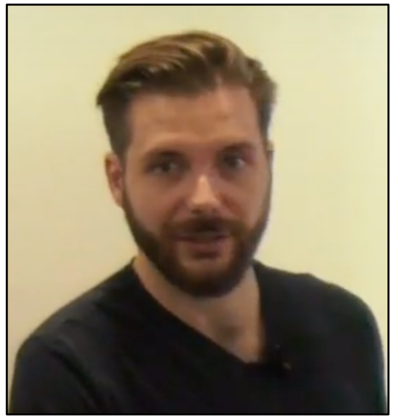
```scala
package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
  }
}
```

```scala
final case class Person(age: Int, name: String)
```

Recap - back to an **intermediate version** of the code with a generic sort that doesn't compile.

@philip_schwarz

```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```


Monty West

So I am going to reintroduce the Orderable trait, with some slight differences.

So instead of just taking one other of whatever, I am trying to compare, it is going to take two, it is going to take both in, so it is going to look like this, I won't write that comment again, that took too long, and I will also put the less than on just for our sort function:

```
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0

}
```

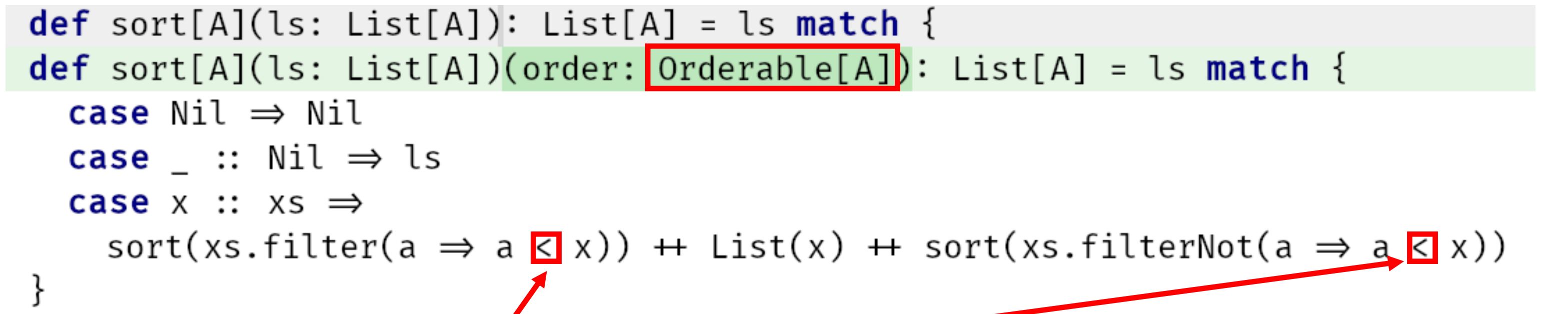
```
trait Orderable[A] {

  /**
   * this < other : negative
   * this = other : 0
   * this > other : positive
   *
   * @param other
   * @return
   */
  def compare(other: A): Int

  def <(other: A): Boolean = compare(other) < 0

}
```

So very similar to last time. This time we are passing in both parameters. The intention here is that this won't be implemented by our Person class, so we won't have access to 'this', so we have to pass them both in.

Monty West

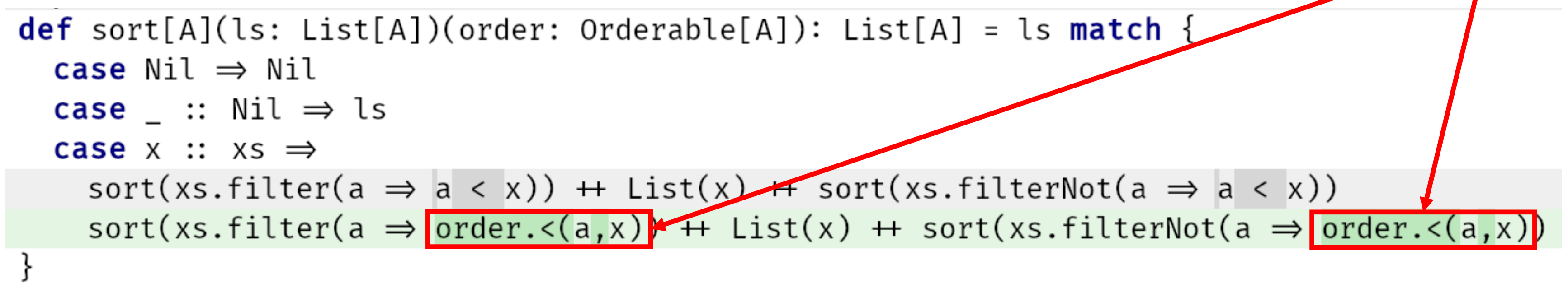
So how are we going to use this **Orderable** trait in our sort function?

So we don't. **Our A is not going to be a Orderable any more, we are not going to get that for free from our A**, **so the only option we really have is to pass this in explicitly**, **so we are going to need some instance of this class, in order to use it, and we are going to pass it in a separate parameter list**.



```
def sort[A](ls: List[A]): List[A] = ls match {
def sort[A](ls: List[A])(order: Orderable[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
}
```

And also, **we don't get this nice syntax anymore**, we are going to have to do something like this, which is **a bit clunky**



```
def sort[A](ls: List[A])(order: Orderable[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
    sort(xs.filter(a ⇒ order.<(a,x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ order.<(a,x)))
}
```

and also, **this is still not compiling**…

Monty West

...because I have two parameter lists and so **I have to pass through this `Orderable` instance each time as well**

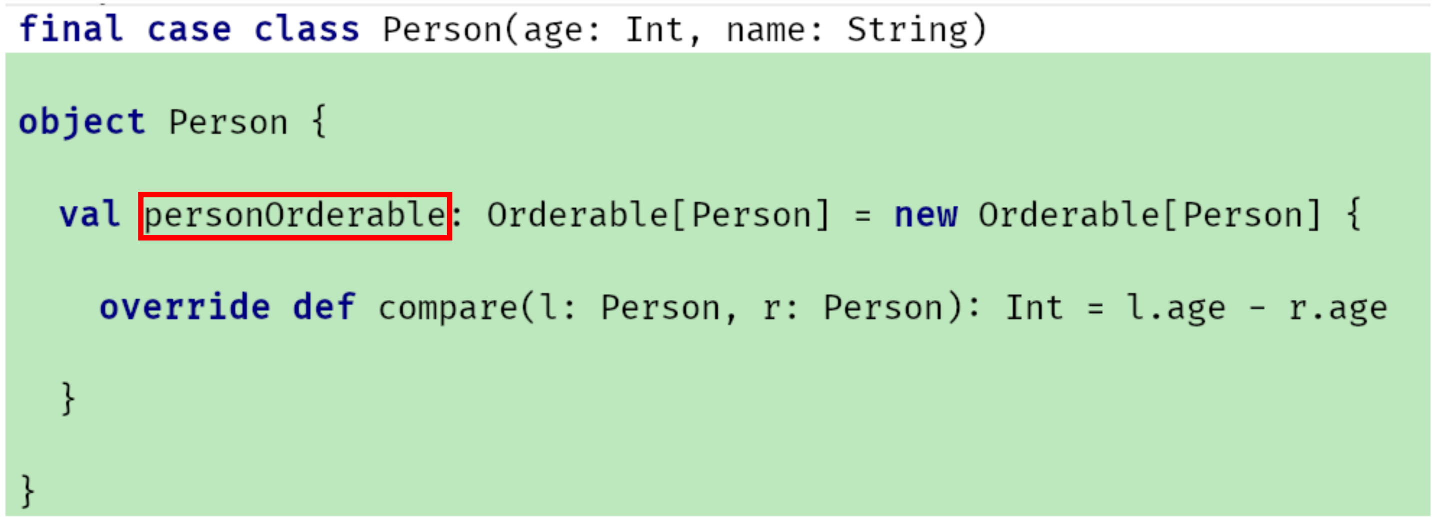
```
def sort[A](ls: List[A])(order: Orderable[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ order.<(a,x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ order.<(a,x))
    sort(xs.filter(a ⇒ order.<(a,x)))(order) ++ List(x) ++ sort(xs.filterNot(a ⇒ order.<(a,x)))(order)
}
```

So **this is getting worse and worse** really. OK, so but we have a sort, it is compiling.

Monty West
https://www.linkedin.com/in/monty-west/

The next stage is to **somehow create this `Orderable` instance for `Person`**.

This isn't going to be a supertype of `Person`, in fact **we don't need to change the `Person` class at all**. You just need somewhere else where you put an instance of this thing. **A common place to see such instances is in the companion object of `Person`** and this will be relevant later when we start introducing implicits, **so I am just going to do that for now and I am going to explain why later**. I am just going to put in a **val** and call it `personOrderable`, and **now we just need to implement this interface**, so **we are going to do the same we did last time, we are going to use age, so it is going to be left age minus right age**:

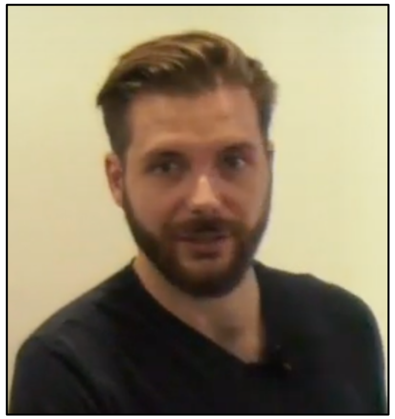


```
final case class Person(age: Int, name: String)

object Person {

  val personOrderable: Orderable[Person] = new Orderable[Person] {

    override def compare(l: Person, r: Person): Int = l.age - r.age

  }

}
```

And this all compiles.

So now we are able to go back to our shell and do exactly what we did last time.

```
sbt:tech_talk-typeclass> run
```



Monty West

OK, so we have out person list

```
Welcome to the Ammonite Repl 1.6.3
(Scala 2.12.8 Java 1.8.0_112)
If you like Ammonite, please support our development at www.patreon.com/lihaoyi
@ personLs
res0: List[Person] = List(Person(23, "alice"), Person(35, "bob"), Person(21, "charlie"))

@
```

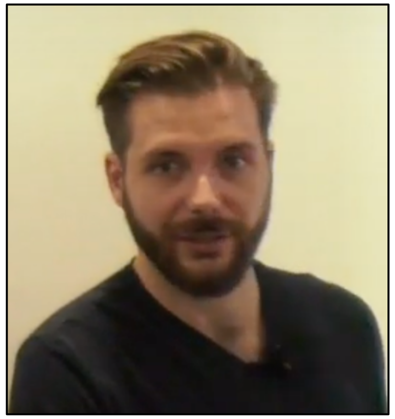
```scala
final case class Person(age: Int, name: String)

object Person {

  val personOrderable: Orderable[Person] = new Orderable[Person] {

    override def compare(l: Person, r: Person): Int = l.age - r.age

  }

}
```

there it is, and hopefully we should be able to do what we did last time, and sort persons, but we need to pass in this instance. So it's a bit clunky,

```
@ ops.sort(personLs)(Person.personOrderable)
res1: List[Person] = List(Person(21, "charlie"), Person(23, "alice"), Person(35, "bob"))

@
```

There we go, that sorts nicely. So we've got the sort, and we've removed the subtyping part, **but we've changed the call site signature**, it's a bit clunky now, you have to remember where things are, we still can't sort Ints, we don't have an instance to put here:

```
@ ops.sort(intLs)(???)
```


Monty West

So this is **the same**, **but potentially worse**. Like, we've got a **clunky call site**

```
@ ops.sort(personLs)(Person.personOrderable)
```

we definitely had some **nastiness of the syntax in the actual sort function**

```
def sort[A](ls: List[A])(order: Orderable[A]): List[A] = ls match {
  case Nil => Nil
  case _ :: Nil => ls
  case x :: xs =>
    sort(xs.filter(a => order.<(a,x)))(order) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))(order)
}
```

yes, it's **not great is it**?

So **can we address this and still unlock some goodness out of it?**

@philip_schwarz

Recap - Version **2** – same as Version **1**, but **potentially worse**. **Clunky call site** and **nastiness in the syntax** of the sort function.

```
@ ops.sort(personLs)(Person.personOrderable)
```

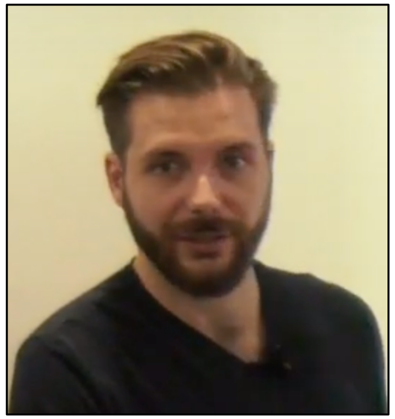
```scala
import mw.domain.Orderable

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => order.<(a,x)))(order) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))(order)
  }
}
```

```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```scala
final case class Person(age: Int, name: String)
```

```scala
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0

}
```

```scala
final case class Person(age: Int, name: String)

object Person {

  val personOrderable: Orderable[Person] = new Orderable[Person] {
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```



Monty West

The first thing I want to address is this call site problem. How can we get around passing this thing explicitly, and that's the clue I suppose. **I am going to introduce some implicits here**.

So in our sort function, **I am going to make this second parameter list implicit**

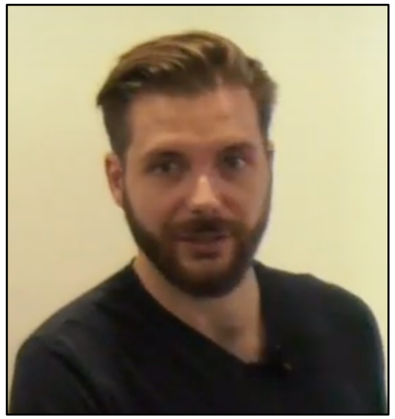
```
def sort[A](ls: List[A])(order: Orderable[A]): List[A] = ls match {
def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ order.<(a,x)))(order) ++ List(x) ++ sort(xs.filterNot(a ⇒ order.<(a,x)))(order)
}
```

And **this does two things for us, it allows for the `Orderable` instance to be passed in implicitly**, so from implicit scope, wherever you are calling it from. **It also means it is added to the implicit scope of this function, <u>so we no longer need this second parameter list here</u>**:

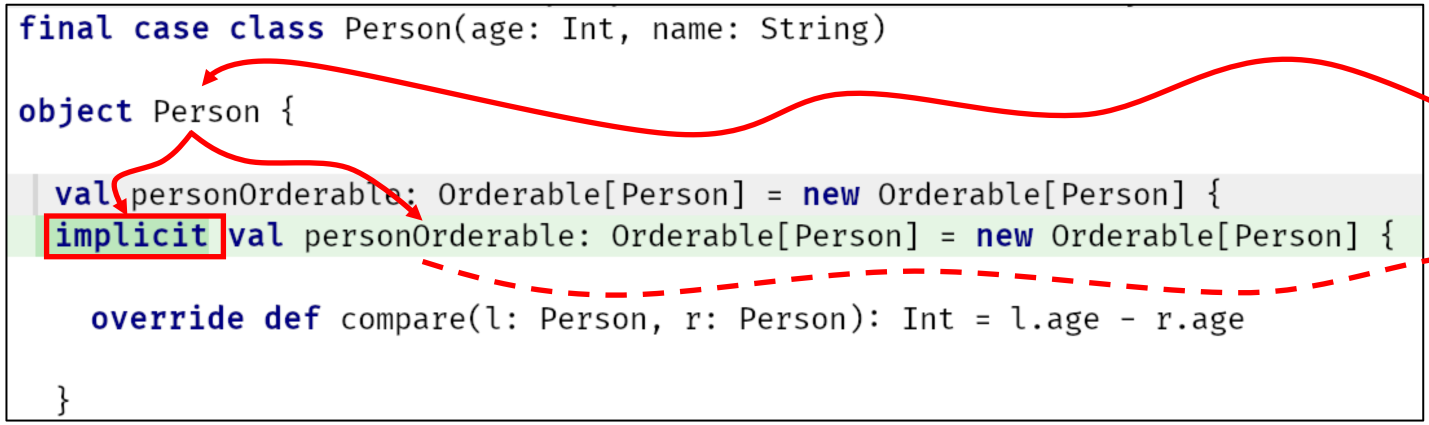
```
def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
  case Nil ⇒ Nil
  case _ :: Nil ⇒ ls
  case x :: xs ⇒
    sort(xs.filter(a ⇒ order.<(a,x)))(order) ++ List(x) ++ sort(xs.filterNot(a ⇒ order.<(a,x)))(order)
    sort(xs.filter(a ⇒ order.<(a,x))) ++ List(x) ++ sort(xs.filterNot(a ⇒ order.<(a,x)))
}
```

And everything will still compile.

And in order to make these Orderable instances available in implicit scope, we can make them implicit where they are defined. So we can make this val implicit:



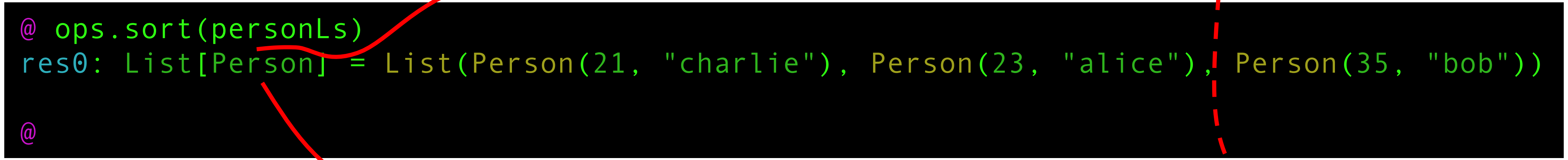
Monty West



```
final case class Person(age: Int, name: String)

object Person {

  val personOrderable: Orderable[Person] = new Orderable[Person] {
  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {

    override def compare(l: Person, r: Person): Int = l.age - r.age

  }
```

Because of implicit magic this all works right, don't worry, I will explain.

So if we run this again we'll see that we can just sort persons exactly like we did with subtype polymorphism with no extra like additions to the call.



```
@ ops.sort(personLs)
res0: List[Person] = List(Person(21, "charlie"), Person(23, "alice"), Person(35, "bob"))

@
```

Like that. So the reason this works is that one of the places where it will look for implicits when it is trying to find one is the companion object of the A, of the class that is parameterising the function.

```
def sort[A](ls: List[A])
           (implicit order: Orderable[A]): List[A]
```

@philip_schwarz

Recap - Version **2b** – Using **implicit personOrderable**

```
import mw.domain.Orderable

package object ops {                @ ops.sort(personLs)

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => order.<(a,x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))
  }
}
```

```
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0

}
```

```
final case class Person(age: Int, name: String)

object Person {

  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

Monty West

https://www.linkedin.com/in/monty-west/

That's one problem out of the way. We've got the call site back as in subtype polymorphism and we are probably like equal now, interfaces, in terms of whoever is using this.

```
@ ops.sort(personLs)
```

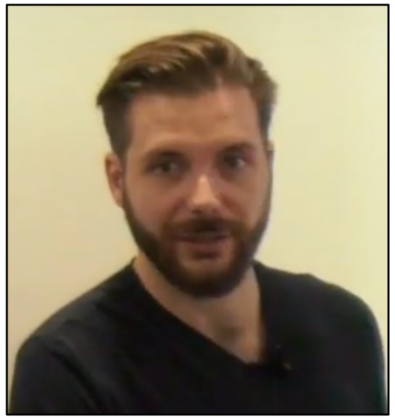
So is there any advantage of this approach?

Well the first one is that we can use our generic sort for Ints now, even though we don't own the Int class, we can define an instance, we didn't have to change Person, so now we can do the same for Int.

Obviously, because we don't own the Int class, we can't add a companion object of Int, so one place you'll commonly see this is in, if you do own the type class itself, you'll see this on the companion object of the type class. And again we are going to make it implicit.

```scala
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0

}

object Orderable {

  implicit val intOrderable: Orderable[Int] = new Orderable[Int] {

    override def compare(l: Int, r: Int): Int = l - r

  }

}
```

OK, so this tiny piece of code will allow us to go back to our shell and sort Ints exactly like we did with Persons, which we couldn't do before with subtype polymorphism. We'd have to write some kind of wrapper around Ints.



Monty West

```
sbt:tech_talk-typeclass> run
```

And then if I show you my Int List

```
@ intLs
res1: List[Int] = List(-5, 8, 10, 2, 5)

@
```

and then call sort

```
@ ops.sort(intLs)
res2: List[Int] = List(-5, 2, 5, 8, 10)

@
```

```scala
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0

}

object Orderable {

  implicit val intOrderable: Orderable[Int] = new Orderable[Int] {

    override def compare(l: Int, r: Int): Int = l - r

  }

}
```

```scala
def sort[A](ls: List[A])
          (implicit order: Orderable[A]): List[A]
```

And the reason this works is because somewhere else it will look for implicit resolution and when you call this function it [the implicit Orderable instance] is on the companion object of the type class of that parameter list.

```scala
import mw.domain.Orderable

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => order.<(a,x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))
  }
}
```

```
@ ops.sort(personLs)
```

```
@ ops.sort(intLs)
```

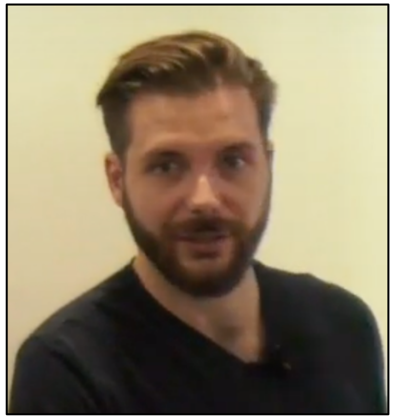
Recap - Version 2c
Using implicit **personOrderable** and implicit **intOrderable**.

@philip_schwar

```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```scala
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0

}

object Orderable {

  implicit val intOrderable: Orderable[Int] = new Orderable[Int] {

    override def compare(l: Int, r: Int): Int = l - r

  }

}
```

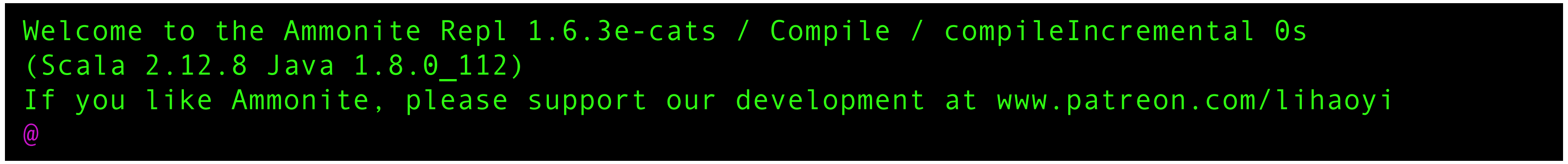
```scala
final case class Person(age: Int, name: String)

object Person {

  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```



Monty West

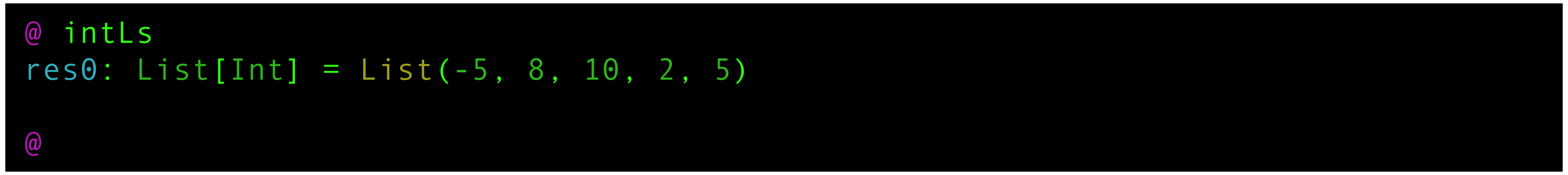
Cool. So where have we got to? Ah, it's worth noting that **if you don't own the typeclass itself, if you don't own this trait, then you can't obviously add it to the companion object of the trait**, **so occasionally you will see something like this, where you have to define an instance for something you don't own, with a typeclass that you also don't own**. And **sometimes you'll see this in an instances package with a companion object**, I just thought I'd quickly show you that,

```scala
package object instances {

  implicit val intOrderable: Orderable[Int] = new Orderable[Int] {
    override def compare(l: Int, r: Int): Int = l - r
  }
}
```



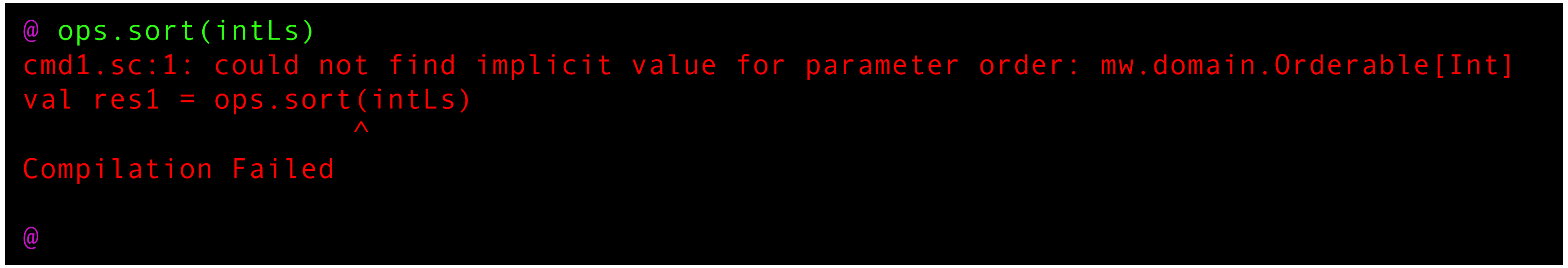
```
Welcome to the Ammonite Repl 1.6.3e-cats / Compile / compileIncremental 0s
(Scala 2.12.8 Java 1.8.0_112)
If you like Ammonite, please support our development at www.patreon.com/lihaoyi
@
```

And **we can still do what we did before**, **there is a slight gotcha here**, I'll just quickly show. So we've still got our **Int List**



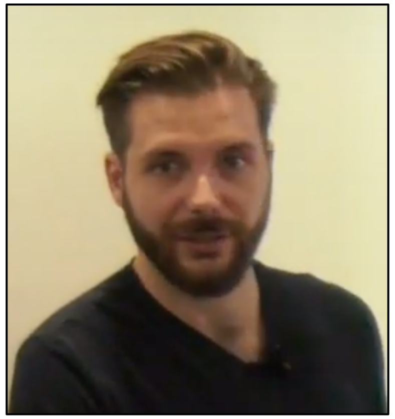
```
@ intLs
res0: List[Int] = List(-5, 8, 10, 2, 5)

@
```

And if we call our sort function, everything should work, everything implicit magic correct, so no:



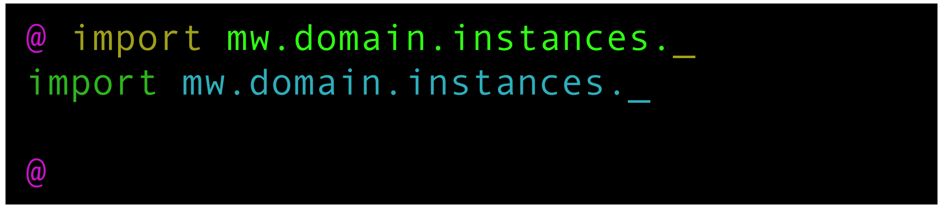
```
@ ops.sort(intLs)
cmd1.sc:1: could not find implicit value for parameter order: mw.domain.Orderable[Int]
val res1 = ops.sort(intLs)
                   ^

Compilation Failed

@
```

The reason is that **we haven't imported this instances package**. So we don't have this nice companion object automatically found by the implicit resolution.



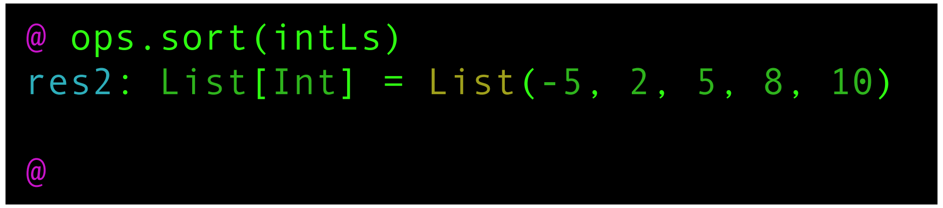
Monty West

So all we need to do is import this **instances package**



```
@ import mw.domain.instances._
import mw.domain.instances._

@
```

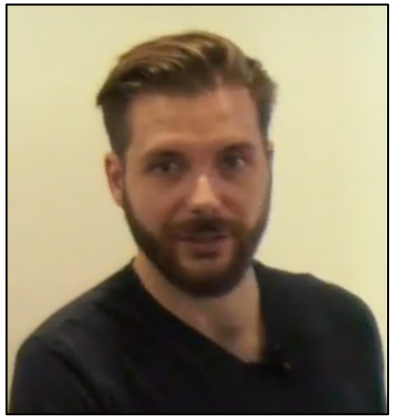
And this is a common pattern you'll see. And now that will sort:



```
@ ops.sort(intLs)
res2: List[Int] = List(-5, 2, 5, 8, 10)

@
```

So, cool, so now **we can use our generic sort function that we have defined, with the type class that we have defined, for a type that we haven't defined and we don't own**.

So slightly more functionality there over **subtype polymorphism**.



Recap - Version **2d** – still using **implicit personOrderable** and **implicit intOrderable**, **even if not owning Orderable**

@philip_schwarz

```
import mw.domain.Orderable

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => order.<(a,x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))
  }
}
```

```
@ ops.sort(personLs)
```

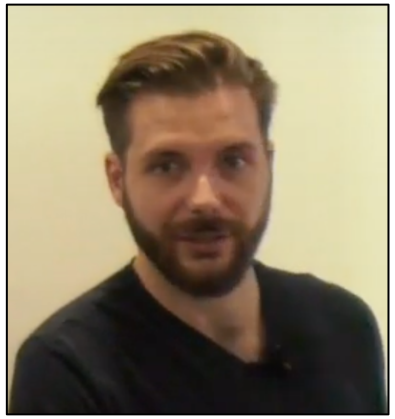
```
@ import mw.domain.instances._
import mw.domain.instances._

@ ops.sort(intLs)
```

```
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0
}
```

```
final case class Person(age: Int, name: String)

object Person {

  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

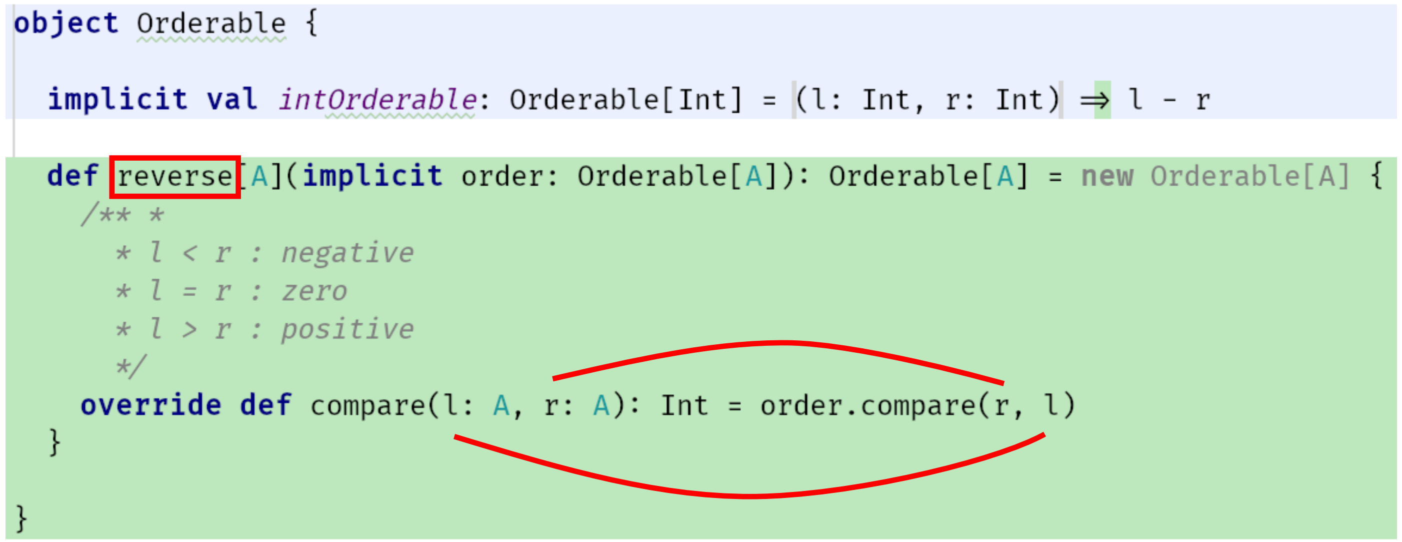
```
package object instances {

  implicit val intOrderable: Orderable[Int] = new Orderable[Int] {
    override def compare(l: Int, r: Int): Int = l - r
  }
}
```


Monty West

What we have done here is something that is quite common in FP generally, which is, we have taken a concept, such as subtype polymorphism, and we turned it into a value. And the benefit we gain from that is **a higher level of abstraction**, **we can now take these values and we can compose them, we can transform them, we can create new values**. And that can **unlock quite a nice level of abstraction** as I said, and also, **reduce some of this boilerplate**, we can **get some nice functionality out of it**.

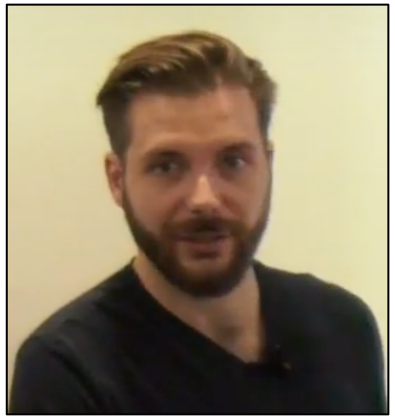
So the first one I want to talk about is what if we want to reverse the order of any of these? So in our **Orderable** trait, what if we want to reverse an order? **How can we go about that?** Does this approach do this better than subtype polymorphism?

So what we can do is we can take an **Orderable** instance in a function and we can transform it and we can return it out the other side. And we can do that to **provide some kind of reverse functionality**. So if I define this function, and it is going to take in an **Orderable** and it is going to take it in implicitly and it is going to spit out an **Orderable**. And **this is suprisingly easy to define**. We have this orderable for A already, so **we just have to compare them in the opposite order**.



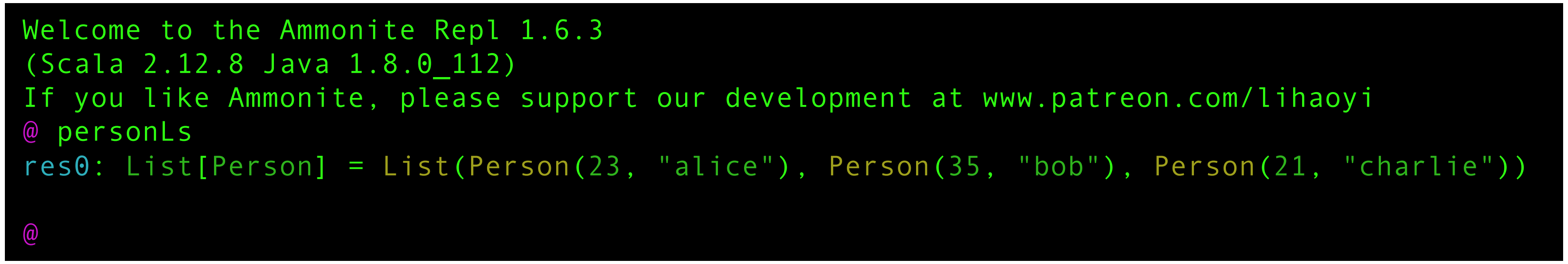
```
object Orderable {

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) ⇒ l - r

  def reverse[A](implicit order: Orderable[A]): Orderable[A] = new Orderable[A] {
    /** *
      * l < r : negative
      * l = r : zero
      * l > r : positive
      */
    override def compare(l: A, r: A): Int = order.compare(r, l)
  }

}
```

so this will take a sort and reverse that sort, **return an instance that does the reverse sort of what we passed in**, **implicitly**.



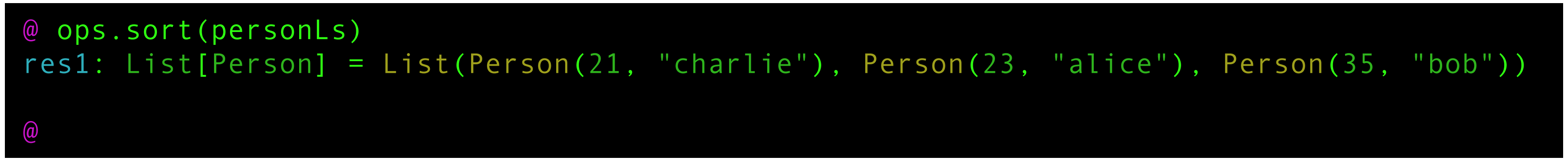
Monty West

So how can we use this? **We don't need to make any changes anywhere else**. If we go back to the shell, so our person list



```
Welcome to the Ammonite Repl 1.6.3
(Scala 2.12.8 Java 1.8.0_112)
If you like Ammonite, please support our development at www.patreon.com/lihaoyi
@ personLs
res0: List[Person] = List(Person(23, "alice"), Person(35, "bob"), Person(21, "charlie"))

@
```

so if I sort the normal way, as defined in the personOrderable instance, we get this way.



```
@ ops.sort(personLs)
res1: List[Person] = List(Person(21, "charlie"), Person(23, "alice"), Person(35, "bob"))

@
```

And now I can do two different ways of applying this reverse. I can do, because the sort function still has this second parameter list, even though it is implicit, we can pass it in explicitly

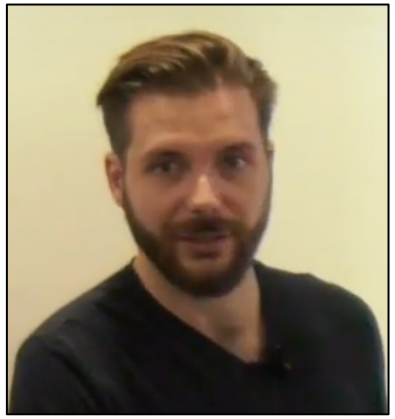
Monty West

So this is me **passing it in explicitly**. What is happening is that this reverse function itself finds the initial person orderable and just … and … another one that is explicitly passed in. So there is some interesting resolution here but not at the sort of function level, at the `Orderable` reverse.

```
@ ops.sort(personLs)(Orderable.reverse)
res2: List[Person] = List(Person(35, "bob"), Person(23, "alice"), Person(21, "charlie"))

@
```

And that has reversed the order. **The other way we can do reverse is in this scope, we can override that implicit**. So if we define an implicit value here, let's call it reverse.

```
@ implicit val reverse: Orderable[Person] = Orderable.reverse
reverse: Orderable[Person] = mw.domain.Orderable$$anon$1@610dd9c4

@
```



And **the type inference in Scala will be clever enough to know that I want the reverse of a `Person`**, just by the type parameter on the actual value. And now if I call just normal sort person, it will have the reverse.

```
@ ops.sort(personLs)
java.lang.NullPointerException
  mw.domain.Orderable$$anon$1.compare(Orderable.scala:26)
  …
```

Oh, not quite …<comment from audience>… ah ok, I'll skip that for now, but you can do that, I have obviously just forgotten how to.

```
import mw.domain.Orderable

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => order.<(a,x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))
  }
}
```

```
@ ops.sort(personLs)
```

```
@ ops.sort(personLs)(Orderable.reverse)
```

Recap - Version 2e - reversing an ordering

@philip_schwarz

```
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```
trait Orderable[A] {

  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0
}

object Orderable {

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r

  def reverse[A](implicit order: Orderable[A]): Orderable[A] = new Orderable[A] {
    override def compare(l: A, r: A): Int = order.compare(r, l)
  }

}
```

```
final case class Person(age: Int, name: String)

object Person {

  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

Monty West
https://www.linkedin.com/in/monty-west/

So, something else we can do is, so **the slight problem here is this boilerplate**

```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    /** *
      * l < r : negative
      * l = r : zero
      * l > r : positive
      */
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

it is annoying. This whole like **create a new instance every time**. **What we can do instead of that** is **take an orderable instance of something that we know how to order and then transform it into something that we don't know how to order yet**. So how about this? I'll explain in a second.

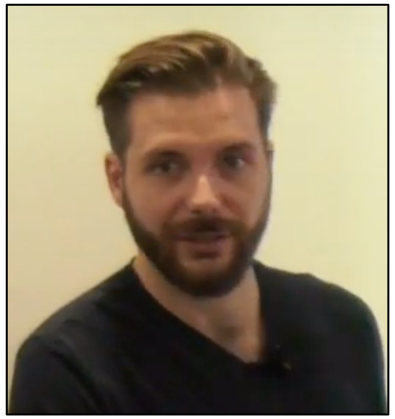
```scala
def by[A,B](f: A => B)(implicit order: Orderable[B]): Orderable[A]
```

So we know how to order Bs, we have an instance for this `Orderable[B]`, and we have a function that takes us from the thing we want to be able to order to that B. And again, this is pretty easy to implement. We take our `Orderable` instance for B and we apply that function to each A that we passed in:

```scala
override def compare(l: A, r: A): Int = order.compare(f(l), f(r))
```

What this allows us to do is reduce a lot of boilerplate and also make our code very readable.

Monty West



So if we go back to our **Person** class, we know how to order **Ints**, we have seen **Ints** already, so instead of this

```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {

    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

now we would just call **Orderable**.by, and this takes a function, so I am going to take a **person** and I am going to pull off the age

```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] =
    Orderable.by(person => person.age)(intOrderable)
}
```

```scala
object Orderable {

  implicit val intOrderable: Orderable[Int] =
    (l: Int, r: Int) => l - r
  …
}
```

A = **Person**; B = **Int**

```scala
object Orderable {
  implicit val intOrderable: Orderable[Int] = …
  …
 def by[A,B](f: A => B)(implicit order: Orderable[B]): Orderable[A] = new Orderable[A] {
    override def compare(l: A, r: A): Int = order.compare(f(l), f(r))
   }
}
```

And that compiles, because we know how to sort **Ints**, that is available on the **Orderable** companion object in implicit scope, and now we are just saying we want to order persons by age and we already know how to sort ints so we don't need to redefine them.

And also **if we use a bit more Scala sugar here this becomes really readable right**?

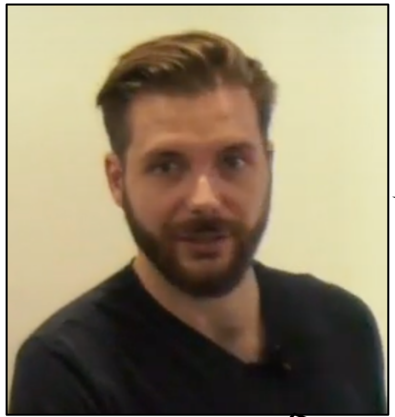
```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] = Orderable.by(_.age)
}
```

Monty West

Orderable by age. It's great.
If we then go back and we **create an Orderable instance for String**, just use the Java or what ever Scala does:

```scala
implicit val stringOrderable: Orderable[String] =
  (l: String, r: String) => l.compareTo(r)
```

Then we can show **how we can change a sort on the fly**. Again start with a **person** list.



```
@ personLs
res0: List[Person] = List(Person(23, "alice"), Person(35, "bob"), Person(21, "charlie"))
@
```

So if we want to sort by name, just in this scope, just for this bit, **we have a default sort that does by age**, that we defined in the **Person** class, **but instead we want to sort by name** for some reason,

```
@ implicit val nameOrderable: Orderable[Person] = Orderable.by(_.name)
nameOrderable: Orderable[Person] = mw.domain.Orderable$$anon$2@6041e4ac
@ ops.sort(personLs)
res2: List[Person] = List(Person(23, "alice"), Person(35, "bob"), Person(21, "charlie"))
```

That was sorted by name rather than by age. So we are able to change the way we use these **type classes**, choose the implementation, in the scope that we want to call it on, **which again we would not be able to do with subtype polymorphism**. **We can change things on the fly**. It may not be the best example here, because I don't know why you would really want to change that sort often, but this can be very useful, this **extra functionality over subtype polymorphism**.

```scala
import mw.domain.Orderable

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => order.<(a,x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a,x)))
  }
}
```

```
@ implicit val nameOrderable: Orderable[Person] = Orderable.by(_.name)
@ ops.sort(personLs)
```

Recap - Version **2f**
Orderable instance creation without boilerplate and on the fly.

@philip_schwarz

```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```scala
trait Orderable[A] {
  def compare(l: A, r: A): Int
  def <(l: A, r: A): Boolean = compare(l, r) < 0
}

object Orderable {
  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r
  implicit val stringOrderable: Orderable[String] = (l: String, r: String) => l.compareTo(r)

  def reverse[A](implicit order: Orderable[A]): Orderable[A] = new Orderable[A] {
    override def compare(l: A, r: A): Int = order.compare(r, l)
  }

  def by[A,B](f: A => B)(implicit order: Orderable[B]): Orderable[A] = new Orderable[A] {
    override def compare(l: A, r: A): Int = order.compare(f(l), f(r))
  }
}
```

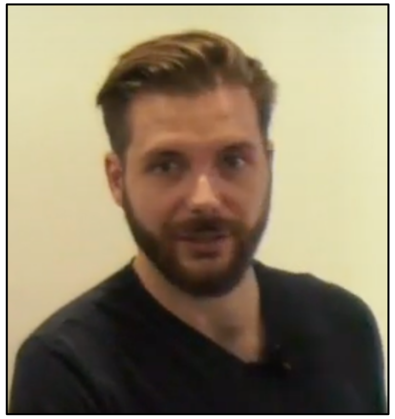
```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] =
    Orderable.by(person => person.age)
}
```

Monty West
https://www.linkedin.com/in/monty-west/

So the last problem we have is that our sort function itself is quite clunky. We've got this second parameter list, we've got this horrible syntax here with order, is there anything we can do about this?

```scala
package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs =>
      sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs =>
      sort(xs.filter(a => order.<(a, x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a, x)))
  }
}
```

Can we make it nicer to use these type classes? And yes, the good news there is that we can.


Monty West

So the first thing I'll address is the syntax. What you'll sometimes see is a syntax package.

And now **I'll have a** package object, and we are going to use an implicit class, I think this pattern has a name, I think it is one of those things in Scala that draws people in from Java, I think it is called pimp my library pattern or something similar, but we are going to use an implicit class, and this will wrap any instance of A that we have an Orderable instance for, and allow us to add functions to it, so we get that nice syntax we saw before.

So we are going to call this OrderableSyntax, and this is going to wrap a value A, but only if we have an Orderable instance for that A. **And the function we want is this** less than. Now this sort of looks like our original subtype polymorphism where we just had an 'other' and a 'this', except that 'this' is passed in sort of explicitly and the class wraps around it

```
package object syntax {
  implicit class OrderableSyntax[A](a: A)(implicit order:Orderable[A]) {
    def <(other: A): Boolean = order.<(a, other)
  }
}
```

OK, so now if we go back to our sort

```
def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
  case Nil => Nil
  case _ :: Nil => ls
  case x :: xs =>
    sort(xs.filter(a => order.<(a, x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a, x)))
}
```

and we remember to import this, now we can us this nice syntax.

```
import mw.domain.syntax._

def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
  case Nil => Nil
  case _ :: Nil => ls
  case x :: xs =>
    sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
}
```

This is starting to look a bit more like what we had before, this nice sort of, use of these type classes can get back to the stage where it feels a lot like using subtype polymorphism and the niceties you have there.

Replacing an implicit `Orderable[A]` parameter with a context bound for `Orderable`

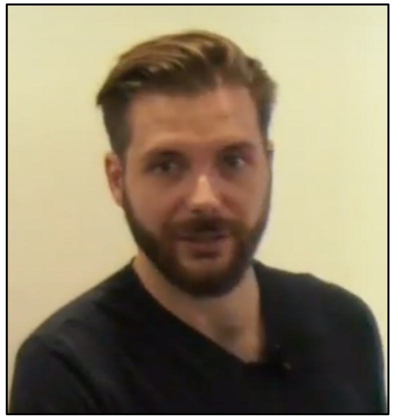
Monty West

```
import mw.domain.syntax._

def sort[A](ls: List[A])(implicit order: Orderable[A]): List[A] = ls match {
  case Nil => Nil
  case _ :: Nil => ls
  case x :: xs =>
    sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
}
```

So the other thing we can do is **we can get rid of this second parameter list**. And there is some **Scala sugar that allows us to do this**, and **it's called a context bound**, and again **it looks a bit like a type bound, there is no arrow there, and we are not passing, there is no like type parameter**, but what this will do is **this will find the `Orderable` instances for `A` and add them to the implicit scope**.

```
 def sort[A: Orderable](ls: List[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs =>
      sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
  }
}
```

What it won't do is give you an explicit value to call, so for example it seems pretty obvious but **we don't have an order anymore**, that doesn't exist, but luckily we don't need it because we are not calling that explicitly, but **there will be times in which you need the instance to do something with, and to get around this and to keep this nice sugar we can add something called a summoner**.



Introducing a **summoner** for `Orderable[A]`

Monty West
https://www.linkedin.com/in/monty-west/

And this is going to look something like this

```scala
object Orderable {

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r
```

I am not going to explain this too much, but **you'll see this a lot on type classes, so just know that it is available to you**

```scala
object Orderable {

  def apply[A](implicit order: Orderable[A]): Orderable[A] = order

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r
```

there you go. Now **what this allows us to do is pull it out of implicit scope and into the value**, **and the syntax for that looks like this**:

```scala
def sort[A: Orderable](ls: List[A]): List[A] = ls match {
  case Nil => Nil
  case _ :: Nil => ls
  case x :: xs =>
    val order = Orderable[A]
    sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
}
```

And now if we wanted to we could go back to that **horrible syntax**, which looks like this:

```scala
sort(xs.filter(a => order.<(a, x))) ++ List(x) ++ sort(xs.filterNot(a => order.<(a, x)))
```

```scala
import mw.domain.Orderable
import mw.domain.syntax._

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A:Orderable](ls: List[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
  }
}
```

Recap - Version **2g**
- Use **context bound** to get rid of 2nd parameter list
- use implicit **syntax class** to get rid of horrible syntax

@philip_schwarz

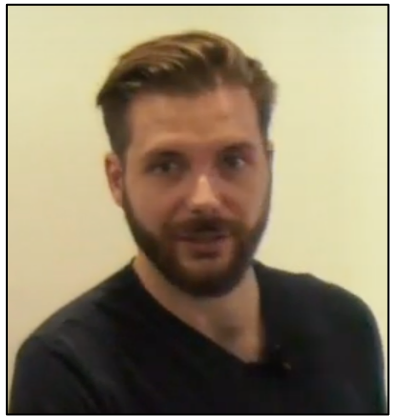
```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

```scala
trait Orderable[A] {
  def compare(l: A, r: A): Int
  def <(l: A, r: A): Boolean = compare(l, r) < 0
}

object Orderable {

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r
  implicit val stringOrderable: Orderable[String] = (l: String, r: String) => l.compareTo(r)

  def reverse[A](implicit order: Orderable[A]): Orderable[A] = new Orderable[A] {
    override def compare(l: A, r: A): Int = order.compare(r, l)
  }
  def by[A, B](f: A => B)(implicit order: Orderable[B]): Orderable[A] = new Orderable[A] {
    override def compare(l: A, r: A): Int = order.compare(f(l), f(r))
  }
}
```

```scala
package object syntax {
  implicit class OrderableSyntax[A](a: A)(implicit order: Orderable[A]) {
    def <(other: A): Boolean = order.<(a, other)
  }
}
```

```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrder: Orderable[Person] =
    Orderable.by (person => person.age)
}
```

Ok, so we have managed to get this to look, if I just undo that, a lot like what we had before, with this nice syntax that we are all familiar with and happy with, this is pretty readable.

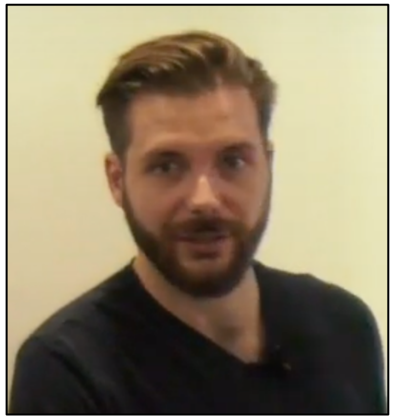

Monty West

```
def sort[A: Orderable](ls: List[A]): List[A] = ls match {
  case Nil => Nil
  case _ :: Nil => ls
  case x :: xs =>
    sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
}
```

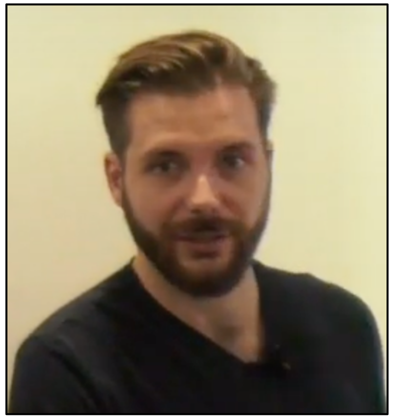
All we had to add was this context bound. For the person itself, that we want to sort, we had to add this couple of lines, separate to the class:

```
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    /** *
      * l < r : negative
      * l = r : zero
      * l > r : positive
      */
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

we didn't have to change the class at all.

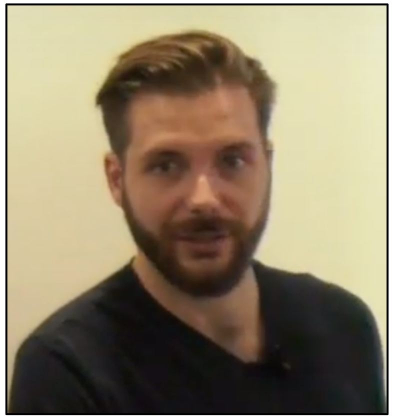


But what we did have to do a lot of was a lot of this stuff, like there is quite a lot of code gone into this type class


Monty West

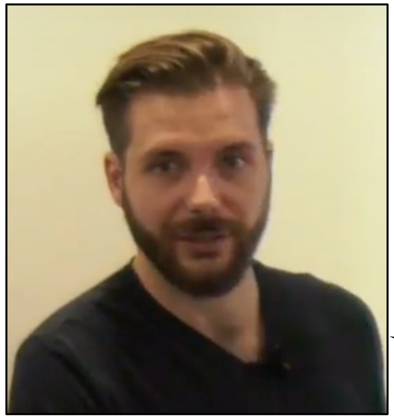
```
trait Orderable[A] {

  /***
   * l < r : negative
   * l = r : zero
   * l > r : positive
   */
  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0
}

object Orderable {

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r

  implicit val stringOrderable: Orderable[String] =
    (l: String, r: String) =>
      if (l == r) 0
      else if (l < r) -1
      else 1

  def by[A, B](f: A => B)(implicit order: Orderable[B]): Orderable[A] =
    (l: A, r: A) => order.compare(f(l), f(r))

  def reverse[A](implicit order: Orderable[A]): Orderable[A] =
    (l: A, r: A) => order.compare(r, l)

}
```

In our actual application code nothing much has changed. We haven’t change our class and we haven’t changed our sort too much. So it would be great if we could get rid of all this boilerplate, if we didn’t have to define this ourselves, so we just delete it.


Monty West

```scala
trait Orderable[A] {

  /***
    * l < r : negative
    * l = r : zero
    * l > r : positive
    */
  def compare(l: A, r: A): Int

  def <(l: A, r: A): Boolean = compare(l, r) < 0
}

object Orderable {

  implicit val intOrderable: Orderable[Int] = (l: Int, r: Int) => l - r

  implicit val stringOrderable: Orderable[String] =
    (l: String, r: String) =>
      if (l == r) 0
      else if (l < r) -1
      else 1

  def by[A, B](f: A => B)(implicit order: Orderable[B]): Orderable[A] =
    (l: A, r: A) => order.compare(f(l), f(r))

  def reverse[A](implicit order: Orderable[A]): Orderable[A] =
    (l: A, r: A) => order.compare(r, l)

}
```

And we delete syntax



Monty West

```
package object syntax {

  implicit class OrderableSyntax[A](a: A)(implicit order:Orderable[A]) {
    def <(other: A): Boolean = order.<(a, other)
  }

}
```

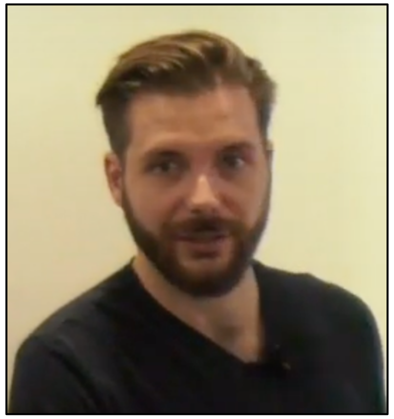
Obviously **these imports are not going to work now**

```
package mw

import mw.domain.Orderable
import mw.domain.syntax._

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil ⇒ Nil
    case _ :: Nil ⇒ ls
    case x :: xs ⇒
      sortInts(xs.filter(a ⇒ a < x)) ++ List(x) ++ sortInts(xs.filterNot(a ⇒ a < x))
  }

  def sort[A: Orderable](ls: List[A]): List[A] = ls match {
    case Nil ⇒ Nil
    case _ :: Nil ⇒ ls
    case x :: xs ⇒
      sort(xs.filter(a ⇒ a < x)(...) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x)(...)
  }
}
```

@philip_schwarz

What Monty West has been doing all along, since he ditched **subtype polymorphism** and switched to **ad hoc polymorphism** using **type classes**, is apply ideas and techniques that are heavily exploited in the **Cats** library. That's why his talk contains the following slides:

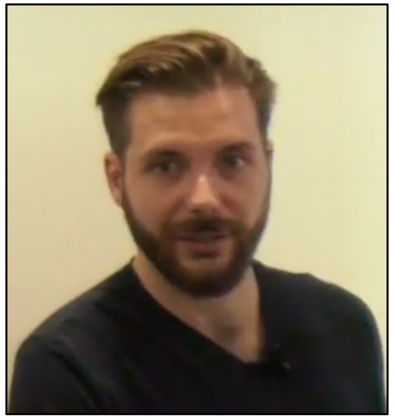
Lightweight, modular, and extensible library for functional programming


Monty West

So what if we still want to sort persons? What can we do if we want to use type classes and stuff?

That's where Cats comes in. So Cats' version of Orderable is called Order, so if I import cats.Order and in the context bound I replace Orderable with Order, and now I don't have this nice syntax,

```
import cats.Order

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil ⇒ Nil
    case _ :: Nil ⇒ ls
    case x :: xs ⇒
      sortInts(xs.filter(a ⇒ a < x)) ++ List(x) ++ sortInts(xs.filterNot(a ⇒ a < x))
  }

  def sort[A: Order](ls: List[A]): List[A] = ls match {
    case Nil ⇒ Nil
    case _ :: Nil ⇒ ls
    case x :: xs ⇒
      sort(xs.filter(a ⇒ a < x)) ++ List(x) ++ sort(xs.filterNot(a ⇒ a < x))
  }
}
```

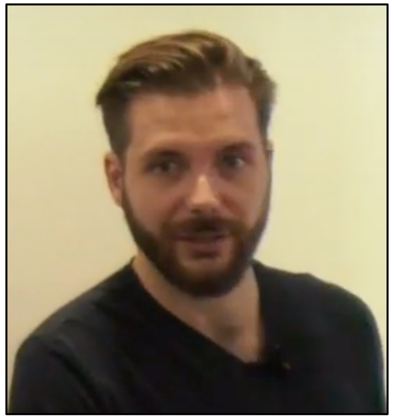


Monty West

But, again, Cats provides, so if I import cats.syntax.order._, now that looks exactly the same, and this is actually a tiny bit of code, I've got two imports and a context bound, I mean, and we can do this sort of generic sorting

```scala
import cats.Order
import cats.syntax.order._

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs =>
      sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))
  }

  def sort[A: Order](ls: List[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs =>
      sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
  }
}
```

for Person, obviously this is a similar problem, we don't have our Orderable any more



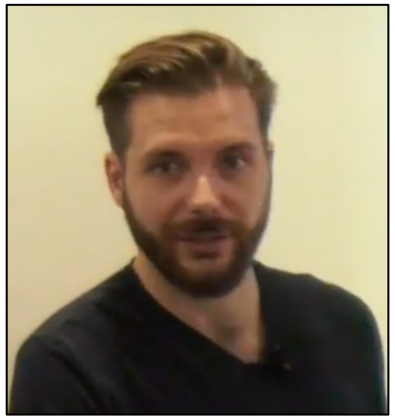
Monty West

```scala
final case class Person(age: Int, name: String)

object Person {
  implicit val personOrderable: Orderable[Person] = new Orderable[Person] {
    /** *
     * l < r : negative
     * l = r : zero
     * l > r : positive
     */
    override def compare(l: Person, r: Person): Int = l.age - r.age
  }
}
```

Cats

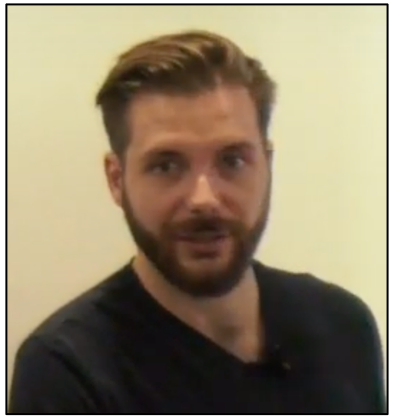
we need to import it.

```scala
import cats.Order

final case class Person(age: Int, name: String)

object Person {

  implicit val personOrder: Order[Person] = Order.by(_.age)(...)

}
```



and this isn't defined because we don't have the Order instance for Age but again Cats gives you a lot of instances for free

```scala
import cats.Order
import cats.instances.int._

final case class Person(age: Int, name: String)

object Person {

  implicit val personOrder: Order[Person] = Order.by(_.age)

}
```


Monty West

Now that compiles and what we have done here is very little work.

We have added a couple of imports, context bounds, added a person instance to implicit scope, a couple of imports and we have done, and we have done a generic sort.

And we have done it all without having to change our Person class. Our Person class can be somewhere completely separate. And we don't have to change that.

We can do this for types we don't own, obviously we don't own the Order type class itself, Cats owns that, so we can add this functionality anywhere.

```scala
import cats.Order
import cats.instances.int._

final case class Person(age: Int, name: String)

object Person {
  implicit val personOrder: Order[Person] =
    Order.by(_.age)
}
```

```scala
val personLs =
  List(
    Person(23, "alice"),
    Person(35, "bob"),
    Person(21, "charlie")
  );

val intLs =
  List(-5, 8, 10, 2, 5);
```

Recapp - Version 3
Using Cats

@philip_schwarz

```scala
import cats.Order
import cats.syntax.order._

package object ops {

  def sortInts(ls: List[Int]): List[Int] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sortInts(xs.filter(a => a < x)) ++ List(x) ++ sortInts(xs.filterNot(a => a < x))

  }

  def sort[A:Order](ls: List[A]): List[A] = ls match {
    case Nil => Nil
    case _ :: Nil => ls
    case x :: xs => sort(xs.filter(a => a < x)) ++ List(x) ++ sort(xs.filterNot(a => a < x))
  }
}
```

Monty West
https://www.linkedin.com/in/monty-west/

I just want to quickly show, or explain what this has actually done for us. So what we can do is, **we have added functionality to the Person object, without changing it at all, so we are now able to compare Persons**.

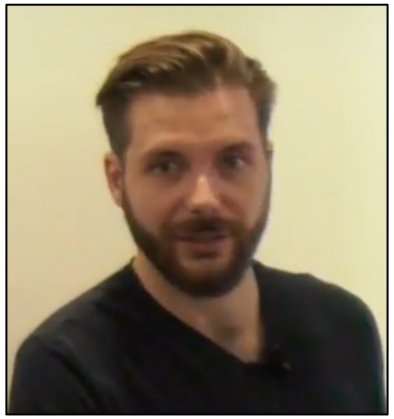
```scala
import cats.Order
import cats.instances.int._

final case class Person(age: Int, name: String)

object Person {

  implicit val personOrder: Order[Person] = Order.by(_.age)

}
```

Obviously this is **quite a powerful approach** and **we could go on like this forever, we could add loads of functionality to our classes without ever needing to change them, without them ever needing to be anything other than value classes or just simple data classes**, and **more than that**, **just a quick taste of the sort of power you can get to, without any actual work**, say we wanted to incorporate the name into the sort, so if we import instances for string and tuple, what we can do is we can get this to **return a tuple**

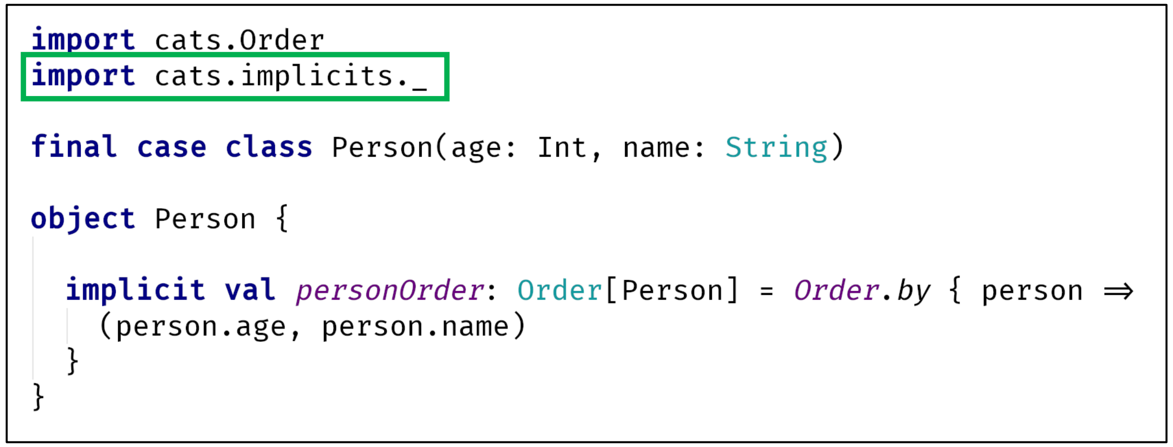
```scala
import cats.Order
import cats.instances.int._
import cats.instances.string._
import cats.instances.tuple._

final case class Person(age: Int, name: String)

object Person {

  implicit val personOrder: Order[Person] = Order.by { person ⇒
    (person.age, person.name)
  }
}
```

So what this will do is it will **sort by age and if they are the same it will sort by name**.

Again, **very minimal code for what is quite a lot of functionality** that we didn't have to define ourselves. Again, **we have added three lines of code**.


Monty West

And **another nice thing about Cats** which I don't think a lot of people really enjoy, is that **if you can't remember where all these instances and syntax are**, **there is just this magic import you can do**, **cats.implicits._**, **just makes everything compile**.



```scala
import cats.Order
import cats.implicits._

final case class Person(age: Int, name: String)

object Person {

  implicit val personOrder: Order[Person] = Order.by { person ⇒
    (person.age, person.name)
  }
}
```

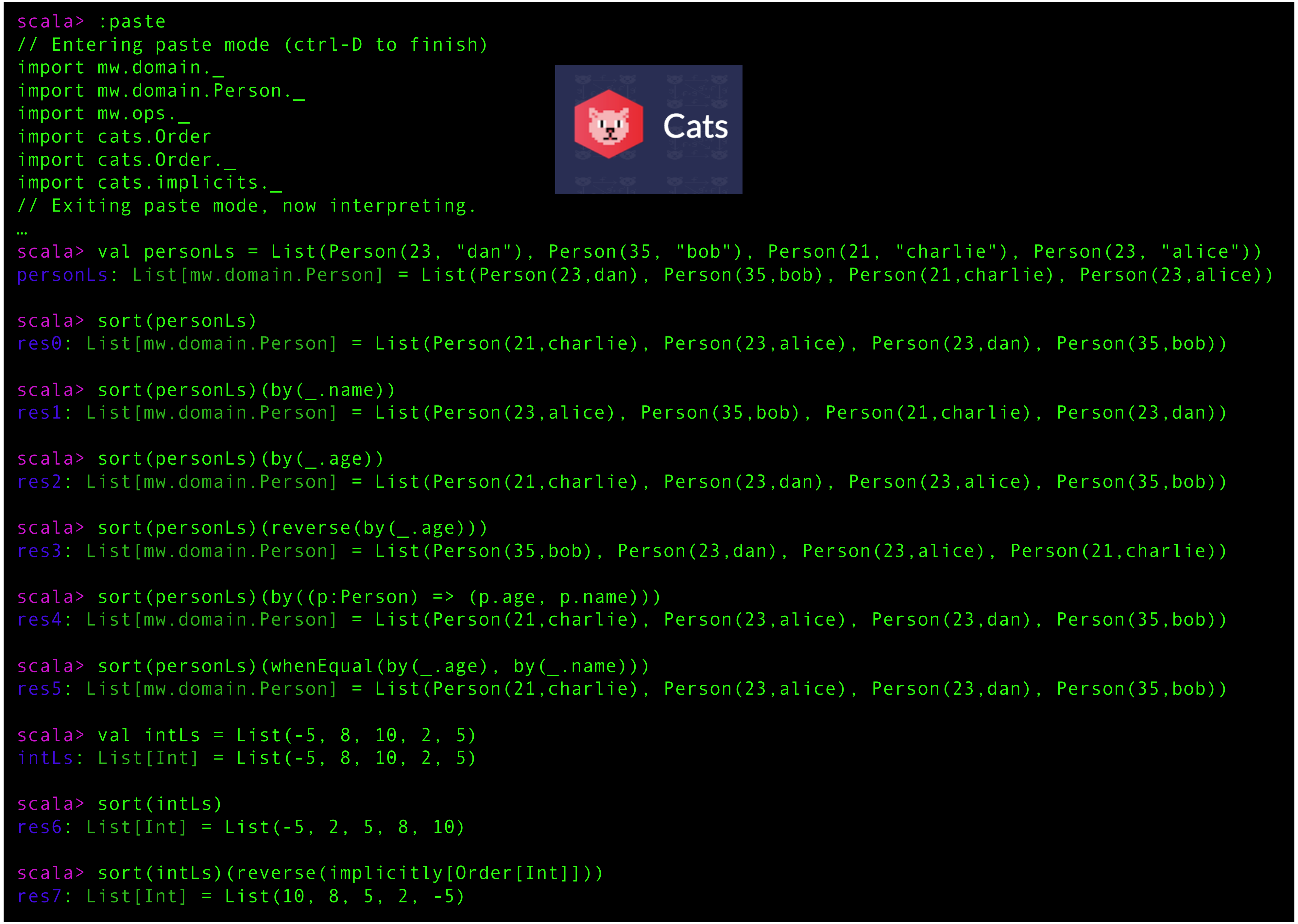
And that's basically the talk. **I have ended up with five extra lines of code from where I started**, so, a very slow live code session, but **unlocked quite a lot of functionality**.

Question from the audience: "do you find that using the **cats.implicits._** instead of the dedicated imports affects the compiler?"

Answer: Yes, so adding this **cats.implicits._** is expensive for you compiler, you will find **the compiler will go slower**, it is good to remember where all these things are, but **if you are really struggling, cats.implicits._ can help, and especially when you get quite deep into all this stuff**. Occasionally you'll find no hit and it just makes everything work, makes everything clean.

Let’s have a quick go at using version 3

@philip_schwarz



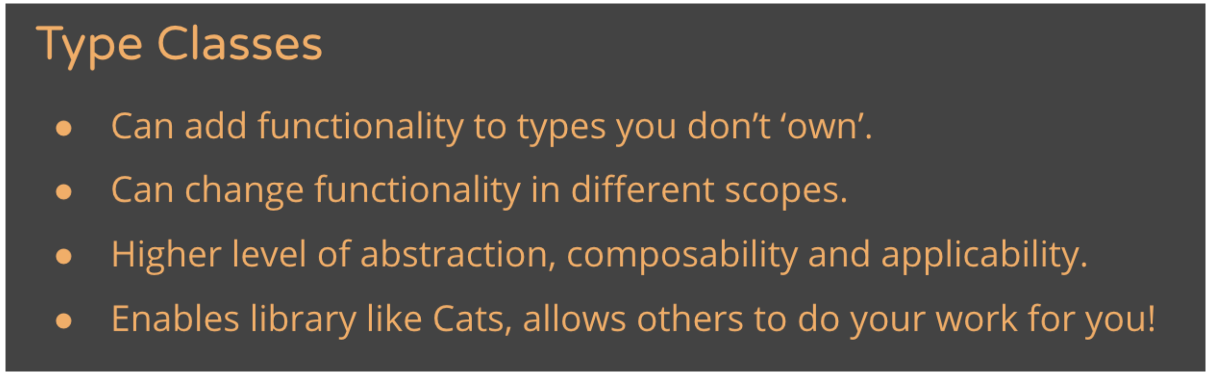
```
scala> :paste
// Entering paste mode (ctrl-D to finish)
import mw.domain._
import mw.domain.Person._
import mw.ops._
import cats.Order
import cats.Order._
import cats.implicits._
// Exiting paste mode, now interpreting.
…
scala> val personLs = List(Person(23, "dan"), Person(35, "bob"), Person(21, "charlie"), Person(23, "alice"))
personLs: List[mw.domain.Person] = List(Person(23,dan), Person(35,bob), Person(21,charlie), Person(23,alice))

scala> sort(personLs)
res0: List[mw.domain.Person] = List(Person(21,charlie), Person(23,alice), Person(23,dan), Person(35,bob))

scala> sort(personLs)(by(_.name))
res1: List[mw.domain.Person] = List(Person(23,alice), Person(35,bob), Person(21,charlie), Person(23,dan))

scala> sort(personLs)(by(_.age))
res2: List[mw.domain.Person] = List(Person(21,charlie), Person(23,dan), Person(23,alice), Person(35,bob))

scala> sort(personLs)(reverse(by(_.age)))
res3: List[mw.domain.Person] = List(Person(35,bob), Person(23,dan), Person(23,alice), Person(21,charlie))

scala> sort(personLs)(by((p:Person) => (p.age, p.name)))
res4: List[mw.domain.Person] = List(Person(21,charlie), Person(23,alice), Person(23,dan), Person(35,bob))

scala> sort(personLs)(whenEqual(by(_.age), by(_.name)))
res5: List[mw.domain.Person] = List(Person(21,charlie), Person(23,alice), Person(23,dan), Person(35,bob))

scala> val intLs = List(-5, 8, 10, 2, 5)
intLs: List[Int] = List(-5, 8, 10, 2, 5)

scala> sort(intLs)
res6: List[Int] = List(-5, 2, 5, 8, 10)

scala> sort(intLs)(reverse(implicitly[Order[Int]]))
res7: List[Int] = List(10, 8, 5, 2, -5)
```

Cats

Monty West
https://www.linkedin.com/in/monty-west/

OK, so that's the end of the live coding portion.

Let's just go back and summarise where we got to.



Type Classes

- Can add functionality to types you don't 'own'.
- Can change functionality in different scopes.
- Higher level of abstraction, composability and applicability.
- Enables library like Cats, allows others to do your work for you!

So over **subtype** **polymorphism**, this sort of **ad hoc** **polymorphism** with **type classes**, **we can add functionality to types we don't own**, which can be very useful, we can change that functionality in different scopes, so if we want to have a different sort function, a different way of doing toString is a similar one, then we can do that in different scopes, again, **a higher level of abstraction** where **we are able to compose these instances, creating instances on the fly very easily**, it makes them more applicable as well.

And I think the good thing here is **it enables things like Cats**. **The Cats library just adds functionality to types it doesn't own**, **`Order`**, that's exactly what it does. **That is not possible with subtype polymorphism obviously**. You'd have to go through and implement everything and what if you wanted to do this for **Java** Time instant for example? The **Cats** library would be useless.

So **this pattern enables stuff like Cats**, which I think a lot of people find very useful. And yes, as we all like, **it does your work for you**.