# From Subtype Polymorphism To Typeclass-based Ad hoc Polymorphism An Example

Scala

slides by  @philip_schwarz FP Illuminated https://fpilluminated.com/

@philip_schwarz

Here is a function that orders a list of integers using quicksort.

```
def order(ns: List[Int]): List[Int] = ns match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension (ns: List[Int])
  def ordered: List[Int] = order(ns)
```

```
assert( List(4, 1, 3, 5, 2).ordered == List(1, 2, 3, 4, 5) )
```

What if we want to order people, e.g. by their age?

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

Let's define a type to be **Orderable** if it provides a function for comparing its values so as to determine their relative order.

```
trait Orderable[A]:
  def compare(other: A): Int

object Orderable:
  extension [A](l: Orderable[A])
    def <(r: A): Boolean = l.compare(r) < 0
```

Returns a negative integer, zero, or a positive integer as **this** is less than, equal to, or greater than **other**.

And now let's make **Person** an **Orderable**

```
case class Person(name: String, age: Int) extends Orderable[Person]:
  override def compare(other: Person): Int = this.age - other.age
```

Getting **Person** to implement (mix-in) the **Orderable** trait.

```
assert( Person("John", 25) < Person("Jane", 30) )
```

Younger people come first

Now let's use **subtype polymorphism** and modify our **order** function so that rather than sorting a list of integers, it sorts a list of any type that is an **Orderable**.

`[A <: Orderable[A]]` means that the **order** function operates on lists whose element type `A` is a **subtype** of upper bound **Orderable**.

```
def order(ns: List[Int]): List[Int] = ns match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension (ns: List[Int])
  def ordered: List[Int] = order(ns)
```

```
def order[A <: Orderable[A]](as: List[A]): List[A] = as match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension [A <: Orderable[A]](as: List[A])
  def ordered: List[A] = order(as)
```

| | | |
|---|---|---|
| `def order(ns: List[Int]): List[Int] = ns match`<br>`  case Nil => Nil`<br>`  case head :: tail =>`<br>`  val (smaller, larger) = tail.partition(_ < head)`<br>`  order(smaller) ++ List(head) ++ order(larger)` | = | `def order[A <: Orderable[A]](as: List[A]): List[A] = as match`<br>`  case Nil => Nil`<br>`  case head :: tail =>`<br>`  val (smaller, larger) = tail.partition(_ < head)`<br>`  order(smaller) ++ List(head) ++ order(larger)` |
| `extension (ns: List[Int])`<br>`  def ordered: List[Int] =`<br>`  order(ns)` | <> | `extension [A <: Orderable[A]](as: List[A])`<br>`  def ordered: List[A] =`<br>`  order(as)` |

Let's sort some people.

```
val people = List( Person("Jane", 30), Person("John", 25), Person("Jim", 18) )
val peopleByIncreasingAge = List( Person("Jim", 18), Person("John", 25), Person("Jane", 30) )

assert(people.ordered == peopleByIncreasingAge)
```

Now we are able to sort people, but we are no longer able to sort integers, because **Int** does not implement (mix-in) the **Orderable** trait, and we cannot make it do so, because we don't own the **Int** type.

Enter **typeclasses**, which allow a more general solution that does not have this limitation.

Let's define a **typeclass** called **Order**, whose **instance** for type A provides a function for comparing A values so as to determine their relative order.

`(using order: Order[A])` is a **context parameter**, often simply called a **'given'**.

```
trait Orderable[A]:
  def compare(other: A): Int

object Orderable:
  extension [A](l: Orderable[A])
    def <(r: A): Boolean = l.compare(r) < 0
```

```
trait Order[A]:
  def compare(l: A, r: A): Int

object Order:
  extension [A](l: A)(using order: Order[A])
    def <(r: A): Boolean = order.compare(l, r) < 0
```

| | | |
|---|---|---|
| `trait Orderable[A]:`<br>`  def compare(other: A): Int` | <> | `trait Order[A]:`<br>`  def compare(l: A, r: A): Int` |
| | = | |
| `object Orderable:`<br><br>`  extension [A](l: Orderable[A])`<br>`  def <(r: A): Boolean = l.compare(r) < 0` | <> | `object Order:`<br><br>`  extension [A](l: A)(using order: Order[A])`<br>`  def <(r: A): Boolean = order.compare(l, r) < 0` |

Now let's use **typeclass-based ad hoc polymorphism**, and modify our **order** function so that rather than ordering a list of any type that is **Orderable**, it sorts a list of any type for which there exists a **'given' instance** of the **typeclass** called **Order**.

```
def order[A <: Orderable[A]](as: List[A]): List[A] = as match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension [A<: Orderable[A]](as: List[A])
  def ordered: List[A] = order(as)
```

`[A: Order]` is a **context bound**, which is a shorthand syntax for `(using uvxyz: Order[A])`, so it means that function **order** has a **context parameter** of type `Order[A]`.

```
def order[A: Order](as: List[A]): List[A] = as match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension [A: Order](as: List[A])
  def ordered: List[A] = order(as)
```

| | | |
|---|---|---|
| `def order[A <: Orderable[A]](as: List[A]): List[A] = as match` | <> | `def order[A: Order](as: List[A]): List[A] = as match` |
| `  case Nil => Nil`<br>`  case head :: tail =>`<br>`  val (smaller, larger) = tail.partition(_ < head)`<br>`  order(smaller) ++ List(head) ++ order(larger)` | = | `  case Nil => Nil`<br>`  case head :: tail =>`<br>`  val (smaller, larger) = tail.partition(_ < head)`<br>`  order(smaller) ++ List(head) ++ order(larger)` |
| `extension [A <: Orderable[A]](as: List[A])`<br>`  def ordered: List[A] = order(as)` | <> | `extension [A: Order](as: List[A])`<br>`  def ordered: List[A] = order(as)` |

To sort some integers we need to define a **'given'** instance of **Order** for **Int**.

```
given Order[Int] with
  override def compare(l: Int, r: Int): Int = l - r
```

We decided not to give the **instance** a name. if we need to get hold of the **instance**, we can do so using the **summon** function

```
val order: Order[Int] = summon[Order[Int]]
```

To make **summoning** the **instance** more convenient, let's define the following **apply** function in the **Order** companion object

```
def apply[A](using orderable: Order[A]): Order[A] = orderable
```

```
val ascendingIntOrder: Order[Int] = Order[Int]
```

| | | |
|---|---|---|
| `object Order:` | `=` | `object Order:` |
| | `<>` | `  def apply[A](using orderable: Order[A]): Order[A] = orderable` |
| `  extension [A](l: A)(using order: Order[A])`<br>`    def <(r: A): Boolean = order.compare(l, r) < 0` | `=` | `  extension [A](l: A)(using order: Order[A])`<br>`    def <(r: A): Boolean = order.compare(l, r) < 0` |

Let's sort some integers

```
val ints = List(4, 1, 3, 5, 2)
val ascendingInts = List(1, 2, 3, 4, 5)
```

```
assert(ints.ordered == ascendingInts)
```

Our **'given'** instance of **typeclass** `Order[Int]` is **implicitly** passed as a parameter into the **ordered** and **order** functions.

```
assert(ints.ordered(using ascendingIntOrder) == ascendingInts)
assert(ints.ordered(using Order[Int]) == ascendingInts)
```

If for some reason we want to pass the parameter **explicitly**, here are two ways we can do it.

```
assert(ints.ordered(using _ - _) == ascendingInts)
```

Interestingly, an **anonymous function** that subtracts its two parameters also qualifies as an **explicit** `Order[Int]` parameter (see next slide).

## 8.9 “SAM” types

... as in **Java**, **Scala** will allow a **function type** to be used where an **instance** of a **class** or **trait** declaring a **single abstract method** (**SAM**) is required. This will work with any **SAM**. For example, you might define a **trait**, `Increaser`, with a **single abstract method**, **increase**:

```
trait Increaser:
  def increase(i: Int): Int
```

You could then define a method that takes an `Increaser`:

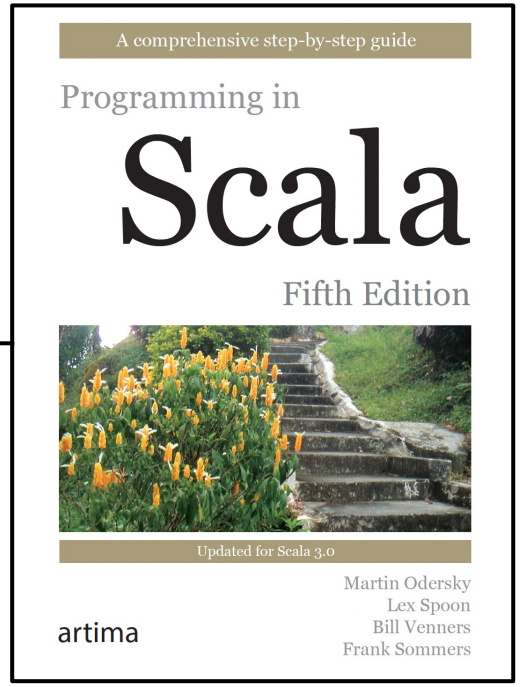
```
def increaseOne(increaser: Increaser): Int =
  increaser.increase(1)
```

To invoke your new method, you could pass in an **anonymous instance** of **trait** `Increaser`, like this:

```
increaseOne(
  new Increaser:
    def increase(i: Int): Int = i + 7
)
```

In **Scala** versions 2.12 and greater, however, you could alternatively just use a **function literal**, because `Increaser` is a **SAM type**:

```
increaseOne(i => i + 7) // Scala
```




```
assert(ints.ordered(using _ - _) == ascendingInts)
```

Because a **SAM** type is expected here, we can pass in a function literal, e.g. _ - _

```
trait Order[A]:
  def compare(l: A, r: A): Int
```

**Order** is a **SAM** type (declares a **SAM**)

**ordered**, and **order** take an **implicit parameter** that is a **SAM** type

```
def order[A: Order](as: List[A]): List[A] = …

extension [A: Order](as: List[A])
  def ordered: List[A] = order(as)
```

Now in the **Order** companion object, let's add a function that given an **Order** which orders elements in a certain way, returns a new **Order** that orders elements in the opposite way.

```
extension [A](order: Order[A])
  def reverse: Order[A] = (l: A, r: A) => order.compare(r, l)
```

Note that we are comparing **r** with **l**, rather than the other way around.

| | | |
|---|---|---|
| `object Order:`<br><br>`  def apply[A](using orderable: Order[A]): Order[A] = orderable`<br><br>`  extension [A](l: A)(using order: Order[A])`<br>`  def <(r: A): Boolean = order.compare(l, r) < 0` | = | `object Order:`<br><br>`  def apply[A](using orderable: Order[A]): Order[A] = orderable`<br><br>`  extension [A](l: A)(using order: Order[A])`<br>`  def <(r: A): Boolean = order.compare(l, r) < 0` |
| | <> | `extension [A](order: Order[A])`<br>`  def reverse: Order[A] = (l: A, r: A) => order.compare(r, l)` |

We can now order integers using the **reverse** of **'given' Order[Int]**, i.e. in descending order.

```
val descendingIntOrder = ascendingIntOrder.reverse

assert(ints.ordered(using descendingIntOrder) == ascendingInts.reverse)

assert(ints.ordered(using Order[Int].reverse) == ascendingInts.reverse)

assert(ints.ordered(using (n1, n2) => n2 - n1) == ascendingInts.reverse)
```

Let's do for **String** what we have done so far for **Int**.

To order **String**s we define a **'given'** instance of **Order** for **String**.

```
given Order[String] with
  override def compare(l: String, r: String): Int = l.compareTo(r)
```

| | | |
|---|---|---|
| given Order[Int] with<br>  override def compare(l: Int, r: Int): Int = l - r | = | given Order[Int] with<br>  override def compare(l: Int, r: Int): Int = l - r |
| | -+ | given Order[String] with<br>  override def compare(l: String, r: String): Int = l.compareTo(r) |

```
val names = List("John", "Jane", "Jim")
val ascendingNames = List("Jane", "Jim", "John")

val ascendingStringOrder = Order[String]

assert(names.ordered == ascendingNames)
assert(names.ordered(using ascendingStringOrder) == ascendingNames)
assert(names.ordered(using Order[String]) == ascendingNames)
assert(names.ordered(using _ compareTo _) == ascendingNames)

val descendingStringOrder = ascendingStringOrder.reverse

assert(names.ordered(using descendingStringOrder) == ascendingNames.reverse)
assert(names.ordered(using Order[String].reverse) == ascendingNames.reverse)
assert(names.ordered(using (s1,s2) => s2 compareTo s1) == ascendingNames.reverse)
```

Now let's move on to ordering people.

We begin by undoing the changes that we made to **Person** in order to use **subtype polymorphism**., i.e. we stop **Person** from implementing (mixing-in) **Orderable**.

```
case class Person(name: String, age: Int) extends Orderable[Person]:
  override def compare(other: Person): Int = this.age - other.age
```

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

| | | |
|---|---|---|
| `case class Person(name: String, age: Int) extends Orderable[Person]:`<br>`  override def compare(other: Person): Int =`<br>`  this.age – other.age` | <> | `case class Person(name: String, age: Int)` |

With **Int**, it was natural for the **'given' Order[Int]** to be ascending order.
Similarly with **String**.

But what should be the default ordering for people? By ascending age?, By ascending name? There isn't a natural order for people, so we are not going to provide a default.

Also, since age is an **Int** and name is a **String**, and since **'givens' Order[Int]** and **Order[String]** already exist, instead of defining an **Order[Person]** from scratch, we are going to define it in terms of one of those (see next slide).

If we have an `Order[B]` for ordering Bs, and we have a function that converts an A to a B, then we can easily create an `Order[A]` for ordering As.

```
extension [B](order: Order[B])
  def on[A](f: A => B): Order[A] = (l: A, r: A) => order.compare(f(l), f(r))
```

| | | |
|---|---|---|
| `object Order:`<br><br>`  def apply[A](using orderable: Order[A]): Order[A] = orderable`<br><br>`  extension [A](l: A)(using order: Order[A])`<br>`  def <(r: A): Boolean = order.compare(l, r) < 0`<br><br>`  extension [A](order: Order[A])`<br>`  def reverse: Order[A] = (l: A, r: A) => order.compare(r, l)` | = | `object Order:`<br><br>`  def apply[A](using orderable: Order[A]): Order[A] = orderable`<br><br>`  extension [A](l: A)(using order: Order[A])`<br>`  def <(r: A): Boolean = order.compare(l, r) < 0`<br><br>`  extension [A](order: Order[A])`<br>`  def reverse: Order[A] = (l: A, r: A) => order.compare(r, l)` |
| | -+ | `  extension [B](order: Order[B])`<br>`  def on[A](f: A => B): Order[A] = (l: A, r: A) => order.compare(f(l), f(r))` |

Let's define some people.

```
val people = List(Person("Jane", 30), Person("John", 25), Person("Jim", 18))
val peopleByAge = List(Person("Jim", 18), Person("John", 25), Person("Jane", 30))
```

Let's say that locally to a certain part of the application, we do want ordering by age to be the default.

```
given ageOrder: Order[Person] = ascendingIntOrder.on(_.age)
```

Create a **Person** order that is an ascending **Int** order **on** a person's age.

```
assert(Person("John", 25) < Person("Jane", 30))
```

Younger people come first

Now let's order the people by ascending age.

```
assert(people.ordered == peopleByAge)
assert(people.ordered(using ageOrder) == peopleByAge)
assert(people.ordered(using Order[Int].on[Person](_.age)) == peopleByAge)
assert(people.ordered(using  (p1, p2) => p1.age - p2.age) == peopleByAge)
```

And now by descending age

```
assert(people.ordered(using ageOrder.reverse) == peopleByAge.reverse)
assert(people.ordered(using Order[Int].on[Person](_.age).reverse) == peopleByAge.reverse)
assert(people.ordered(using (p1, p2) => p2.age - p1.age) == peopleByAge.reverse)
```

Same as on the previous slide, but here we are ordering people by name.

```
val people = List(Person("Jane", 30), Person("John", 25), Person("Jim", 18))
val peopleByName = List(Person("Jane", 30), Person("Jim", 18), Person("John", 25))
```

```
val lexicographicStringOrder = Order[String]
val nameOrder: Order[Person] = lexicographicStringOrder.on(_.name)
```

Create a **Person** order that is an ascending **String** order **on** a person's name.

First let's order the people by ascending name, and then by descending name.

```
assert(people.ordered == peopleByName)
assert(people.ordered(using nameOrder) == peopleByName)
assert(people.ordered(using Order[String].on[Person](_.name)) == peopleByName)
assert(people.ordered(using (p1,p2) => p1.name - p2.name) == peopleByName)
```

```
assert(people.ordered(using nameOrder.reverse) == peopleByName.reverse)
assert(people.ordered(using Order[String].on[Person](_.age).reverse) == peopleByName.reverse)
assert(people.ordered(using (p1, p2) => p2.name - p1.name) == peopleByName.reverse)
```

To conclude the example, let's say that we want to order people first by age, and then by name, i.e. we want to order them by age, but when the age is the same, we want to order them by name.

Rather than defining a **'given'** ordering for this specific case, we are going to do the following:

1. define a generic **'given' `Order[(A, B)]`** for any `A` and `B` for which **'givens' `Order[A]`** and **`Order[B]`** exist.
2. define our desired **`Order[Person]`** applying our existing **`on`** function to **`Order[(Int,String)]`** (the latter exists because **'givens' `Order[Int]`** and **`Order[String]`** exist).

```
given [A, B](using oa: Order[A], ob: Order[B]): Order[(A, B)] with
  override def compare(l: (A, B), r: (A, B)): Int = (l, r) match
    case ((la, lb), (ra, rb)) =>
      val asComparison = oa.compare(la, ra)
      if asComparison == 0
      then ob.compare(lb, rb)
      else asComparison
```

| | | |
|---|---|---|
| `given Order[Int] with`<br>`  override def compare(l: Int, r: Int): Int = l - r`<br><br>`given Order[String] with`<br>`  override def compare(l: String, r: String): Int = l.compareTo(r)` | = | `given Order[Int] with`<br>`  override def compare(l: Int, r: Int): Int = l - r`<br><br>`given Order[String] with`<br>`  override def compare(l: String, r: String): Int = l.compareTo(r)` |
| | -+ | `given [A, B](using oa: Order[A], ob: Order[B]): Order[(A, B)] with`<br>`  override def compare(l: (A, B), r: (A, B)): Int = (l, r) match`<br>`  case ((la, lb), (ra, rb)) =>`<br>`  val asComparison = oa.compare(la, ra)`<br>`  if asComparison == 0 then ob.compare(lb, rb)`<br>`  else asComparison` |

Now we can order people as desired.

```
val morePeople = List(Person("John", 25), Person("Jane", 30), Person("Jack", 25), Person("Jim", 18))
val morePeopleByAgeAndThenName = List(Person("Jim", 18), Person("Jack", 25), Person("John", 25), Person("Jane", 30))

val ageAndThenNameOrder: Order[Person] = Order[(Int,String)].on(p => (p.age, p.name))

assert(morePeople.ordered(using ageAndThenNameOrder) == morePeopleByAgeAndThenName)
```

Note that **John** and **Jack** are the same age.

# Ordering integers

```scala
def order(ns: List[Int]): List[Int] = ns match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension (ns: List[Int])
  def ordered: List[Int] = order(ns)
```

```scala
assert( List(4, 1, 3, 5, 2).ordered == List(1, 2, 3, 4, 5) )
```

# Ordering people using **subtype polymorphism**

```scala
trait Orderable[A]:
  def compare(other: A): Int

object Orderable:
  extension [A](l: Orderable[A])
    def <(r: A): Boolean = l.compare(r) < 0
```

```scala
def order[A <: Orderable[A]](as: List[A]): List[A] = as match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension [A <: Orderable[A]](as: List[A])
  def ordered: List[A] = order(as)
```

```scala
case class Person(name: String, age: Int) extends Orderable[Person]:
  override def compare(other: Person): Int = this.age - other.age
```

```scala
val people = List( Person("Jane", 30), Person("John", 25), Person("Jim", 18) )
val peopleByIncreasingAge = List( Person("Jim", 18), Person("John", 25), Person("Jane", 30) )

assert(people.ordered == peopleByIncreasingAge)
```

Ordering integers, strings and people using **typeclass-based ad hoc polymorphism**

```scala
trait Order[A]:
  def compare(l: A, r: A): Int

object Order:

  def apply[A](using orderable: Order[A]): Order[A] =
    orderable

  extension [A](l: A)(using order: Order[A])
    def <(r: A): Boolean = order.compare(l, r) < 0

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

  extension [B](order: Order[B])
    def on[A](f: A => B): Order[A] = (l: A, r: A) =>
      order.compare(f(l), f(r))
```

```scala
given Order[Int] with
  override def compare(l: Int, r: Int): Int = l - r

given Order[String] with
  override def compare(l: String, r: String): Int =
    l.compareTo(r)

given [A,B](using oa:Order[A], ob:Order[B]):Order[(A,B)] with
  override def compare(l: (A, B), r: (A, B)): Int =
    (l, r) match
      case ((la, lb), (ra, rb)) =>
        val asComparison = oa.compare(la, ra)
        if asComparison == 0 then ob.compare(lb, rb)
        else asComparison
```

```scala
def order[A: Order](as: List[A]): List[A] = as match
  case Nil => Nil
  case head :: tail =>
    val (smaller, larger) = tail.partition(_ < head)
    order(smaller) ++ List(head) ++ order(larger)

extension [A: Order](as: List[A])
  def ordered: List[A] = order(as)
```

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

```scala
val order: Order[Int] = summon[Order[Int]]
```

```scala
val ascendingIntOrder: Order[Int] = Order[Int]
```

```scala
val ints = List(4, 1, 3, 5, 2)
val ascendingInts = List(1, 2, 3, 4, 5)
```

```scala
assert(ints.ordered == ascendingInts)
```

```scala
assert(ints.ordered(using ascendingIntOrder) == ascendingInts)
assert(ints.ordered(using Order[Int]) == ascendingInts)
```

```scala
assert(ints.ordered(using _ - _) == ascendingInts)
```

```scala
val descendingIntOrder = ascendingIntOrder.reverse

assert(ints.ordered(using descendingIntOrder) == ascendingInts.reverse)
assert(ints.ordered(using Order[Int].reverse) == ascendingInts.reverse)
assert(ints.ordered(using (n1, n2) => n2 - n1) == ascendingInts.reverse)
```

```scala
val names = List("John", "Jane", "Jim")
val ascendingNames = List("Jane", "Jim", "John")

val ascendingStringOrder = Order[String]

assert(names.ordered == ascendingNames)
assert(names.ordered(using ascendingStringOrder) == ascendingNames)
assert(names.ordered(using Order[String]) == ascendingNames)
assert(names.ordered(using _ compareTo _) == ascendingNames)

val descendingStringOrder = ascendingStringOrder.reverse

assert(names.ordered(using descendingStringOrder) == ascendingNames.reverse)
assert(names.ordered(using Order[String].reverse) == ascendingNames.reverse)
assert(names.ordered(using (s1,s2) => s2 compareTo s1) == ascendingNames.reverse)
```

```scala
val people = List(Person("Jane", 30), Person("John", 25), Person("Jim", 18))
val peopleByAge = List(Person("Jim", 18), Person("John", 25), Person("Jane", 30))
```

```scala
given ageOrder: Order[Person] = ascendingIntOrder.on(_.age)
```

```scala
assert(Person("John", 25) < Person("Jane", 30))
```

```scala
assert(people.ordered == peopleByAge)
assert(people.ordered(using ageOrder) == peopleByAge)
assert(people.ordered(using Order[Int].on[Person](_.age)) == peopleByAge)
assert(people.ordered(using  (p1, p2) => p1.age - p2.age) == peopleByAge)
```

```scala
assert(people.ordered(using ageOrder.reverse) == peopleByAge.reverse)
assert(people.ordered(using Order[Int].on[Person](_.age).reverse) == peopleByAge.reverse)
assert(people.ordered(using (p1, p2) => p2.age - p1.age) == peopleByAge.reverse)
```

```scala
val people = List(Person("Jane", 30), Person("John", 25), Person("Jim", 18))
val peopleByName = List(Person("Jane", 30), Person("Jim", 18), Person("John", 25))
```

```scala
val lexicographicStringOrder = Order[String]
val nameOrder: Order[Person] = lexicographicStringOrder.on(_.name)
```

```scala
assert(people.ordered == peopleByName)
assert(people.ordered(using nameOrder) == peopleByName)
assert(people.ordered(using Order[String].on[Person](_.name)) == peopleByName)
assert(people.ordered(using (p1,p2) => p1.name - p2.name) == peopleByName)
```

```scala
assert(people.ordered(using nameOrder.reverse) == peopleByName.reverse)
assert(people.ordered(using Order[String].on[Person](_.age).reverse) == peopleByName.reverse)
assert(people.ordered(using (p1, p2) => p2.name - p1.name) == peopleByName.reverse)
```

```scala
val morePeople = List(Person("John", 25), Person("Jane", 30), Person("Jack", 25), Person("Jim", 18))
val morePeopleByAgeAndThenName = List(Person("Jim", 18), Person("Jack", 25), Person("John", 25), Person("Jane", 30))

val ageAndThenNameOrder: Order[Person] = Order[(Int,String)].on(p => (p.age, p.name))

assert(morePeople.ordered(using ageAndThenNameOrder) == morePeopleByAgeAndThenName)
```

That's all – I hope you found it useful

inspired by

# 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/


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



slides by  @philip_schwarz

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

FP Illuminated

https://fpilluminated.com/