Conditional Implicit Definitions

- In this lesson, we will see that implicit definitions can themselves take implicit parameters
- Let's start with an example. Consider how we order two String values: is "abc" lexicographically before "abd"? To answer this question, we need to compare all the characters of the String values, element-wise:
    - Is a before a? No
    - Is b before b? No
    - Is c before d? Yes
    - We conclude that "abc" is before "abd"
- So, we compare two sequences of characters with an algorithm that compares the characters of the sequences element-wise. Said otherwise, we can define an ordering relation for a sequence of characters based on the ordering relation for characters
- Can we generalize this process to sequences of any element of type A for which there is an implicit Ordering[A] instance? The signature of such an Ordering[List[A]] definition takes an implicit parameter of type Ordering[A]:

```
implicit def orderingList[A](implicit ord: Ordering[A]): Ordering[List[A]]
```

- For reference, a complete implementation is shown below. You can see that at some point in the algorithm we call the operation compare of the ord parameter:

```
implicit def orderingList[A](implicit ord: Ordering[A]): Ordering[List[A]] =
  new Ordering[List[A]] {
    def compare(xs: List[A], ys: List[A]) = (xs, ys) match {
      case (x :: xsTail, y :: ysTail) => {
        val c = ord.compare(x, y)
        if (c != 0) c else compare(xsTail, ysTail)
      case (Nil, Nil) => 0
      case ( _, Nil) => 1
      case (Nil, _ ) => -1
    }
  }
```

- With this definition, we can sort a list of lists of numbers, for example:

```
val xss = List(List(1, 2, 3), List(1), List(1, 1, 3))
sort(xss)        // List(List(1), List(1, 1, 3), List(1, 2, 3))
```

- But let's take a step back. We haven't defined an instance of Ordering[List[Int]] and yet we have been able to sort a list of List[Int] elements. How did the compiler manage to provide such an instance to us?
- First, we called sort(xss). The compiler fixed the type parameter A of the method to List[Int], based on the type of the argument xss, as if we have written: ***sort[List[Int]](xss)***
- Then, the compiler searched for an implicit definition of type Ordering[List[Int]]. It found that our orderingList definition could be a match under the condition that it could also find definition of type Ordering[Int], which it eventually found. Finally, the compiler inserted the following arguments for us: sort[List[Int]](xss)(orderingList(Ordering.Int))
- In this case, the compiler combined two implicit definitions (orderingList and Ordering.Int) before terminating

- In general, though, an arbitrary number of implicit definitions can be combined until the search hits a “terminal” definition
- Consider for instance these four implicit definitions:

  implicit def a: A = …
  implicit def aToB(implicit a: A): B = …
  implicit def bToC(implicit b: B): C = …
  implicit def cToD(implicit c: C): D = …

  We can then ask the compiler to summon a value of type D: ***implicitly[D]***
  The compiler finds that there is a candidate definition, cToD, that can provide such a D value, under the condition that it can also find an implicit definition of type C. Again, it finds that there is a candidate definition, bToC, that can provide such a C value, under the condition that it can also find an implicit definition of type B. Once again, it finds that there is a candidate definition, aToB, that can provide such a B value, under the condition that it can also find an implicit value of type A. Finally, it finds a candidate definition for type A and the algorithm terminates.
- At the beginning of this lesson, we showed that by using implicit parameters the compiler could infer simple arguments for us. We have now reached a point where we can appreciate that the compiler can infer more complex arguments (by inferring arguments of arguments)
- It not only significantly reduces code verbosity, it also alleviates developers from implementing parts of their programs, which are summoned by the compiler based on their type. In practice, complex fragments of programs such as serializers and deserializers of data types can be summoned by the compiler

Recursive Implicit Definitions

- What happens if we write an implicit definition that depends on itself?

  trait X
  implicit def loop(implicit x: X): X = x
  implicitly[X]
- The compiler detects that it keeps searching for an implicit definition of the same type and returns an error: ***error: diverging implicit expansion for type X starting with method loop***
- Note: it is possible to write recursive implicit definitions by making sure that the search always terminates

Example: Sort by Multiple Criteria

- Consider a situation where we want to compare several movies. Each movie has a title, a rating(in number of stars) and a duration(in minutes):

```
case class Movie(title: String, rating: Int, duration: Int)
val movies = Seq(
   Movie(“Interstellar”, 9, 169),
   Movie(“Inglorious Basterds”, 8, 140),
   Movie(“Fight Club”, 9, 139),
   Movie(“Zodiac”, 8, 157),
)
```
- We want to sort movies by rating first, and then by duration

- To achieve this, a first step is to change our sort function to take as parameter the sort criteria in addition to the elements to sort:

  ```
  def sort[A, B](elements: Seq[A])(criteria: A => B)(implicit ord: Ordering[B]): Seq[A] = ...
  ```
- The sort algorithm remains the same except that instead of comparing the elements together, we compare the criteria applied to each element
- With this function, here is how we can sort movies by title: ***sort(movies)(_.title)***
  And here is how we can sort them by rating: ***sort(movies)(_.rating)***
- Each time the sort function is called, its ordering parameter is inferred by the compiler based on the type of the criteria
- However, our initial problem was to sort the movies by multiple criteria. We would like to sort first by rating and then by duration:

  ```
  sort(movies)(movie => (movie.rating, movie.duration))
  ```
- The type of the criteria is now a tuple type (Int, Int). Unfortunately, the compiler is unable to infer the corresponding ordering parameter. We need to define how simple orderings can be combined together to get an ordering for multiple criteria
- We do so by defining the following implicit ordering:

  ```
  implicit def orderingPair[A, B](implicit
      orderingA: Ordering[A],
      orderingB: Ordering[B]
  ): Ordering[(A, B)] = …
  ```
- This definition provides an ordering for pairs of type (A, B) given orderings for types A and B
- The complete implementation is the following:

  ```
  implicit def orderingPair[A, B](implicit orderingA: Ordering[A],
     orderingB: Ordering[B]): Ordering[(A, B)] = new Ordering[(A, B)] {
        def compare(pair1: (A, B), pair2: (A, B)): Int = {
           val firstCriteria = orderingA.compare(pair1._1, pair2._1)
           if (firstCriteria != 0)
              firstCriteria
           else
              orderingB.compare(pair1._2, pair2._2)
        }
  }
  ```
- We first compare the two values according to the first criteria, and if they are equal we compare them according to the second criteria
- With this definition, the compiler is now able to infer the ordering for the following call:

  ```
  sort(movies)(movie => (movie.rating, movie.duration))
  ```
- Here is the same call where the inferred parameter is explicitly written:

  ```
  sort(movies)(movie => (movie.rating, movie.duration))(
      orderingPair(Ordering.Int, Ordering.Int)
  )
  ```
- Note that in the standard library the sort function that we have defined here is already available as a method sortBy on collections