Motivating Example

- We have previously seen that the compiler is able to infer types from values
- For instance, when we write the definition: ***val x = 42***, the compiler infers that the type of x is Int, because the type of 42 is Int
- This also works with more complex expressions, for instance if we write: ***val y = x + 1***, the compiler is again able to infer that y has type Int because the + operation between two Int values returns as Int value
- The compiler is also able to do the opposite, namely to infer values from types
- The remainder of this lesson introduces the motivation for such a mechanism

Motivating Example – Sorting A List

- Consider a method sort that takes as parameter a List[Int] and returns another List[Int] containing the same elements, but sorted. The implementation of this method would look like the following:

```
def sort(xs: List[Int]): List[Int] = {
  …
  … if (x < y) …
  …
}
```

- The actual implementation of the sorting algorithm does not matter in this example, the important part is that at some point the method has to compare two elements x and y of the list
- The signature of the method sort, as shown above, only works with collections of type List[Int]. It is possible to generalize the method so that it can sort collections of type List[Double] or List[String]?
- A straightforward approach would be to use a polymorphic type A for the type of elements: ***def sort[A](xs: List[A]): List[A] = …***
- But this is not enough, because the comparison operation < is not defined for arbitrary types A. The operation sort has to take as a parameter the comparison operation:

```
def sort[A](xs: List[A])(lessThan: (A, A) => Boolean): List[A] = {
  …
  … if (lessThan(x, y)) …
  …
}
```

- We can call sort as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val strings = List("apple", "pear", "orange", "pineapple")
sort(xs)((x, y) => x < y)                    // result: List[Int](-5, 2, 3, 6, 7)
sort(strings)((s1, s2) => s1.compareTo(s2) < 0)
// result: List[String]("apple", "orange", "pear", "pineapple")
```

Refactoring With Ordering

- The comparison function introduced above is a way to implement an ordering relation. In fact, there is already a type in the standard library, in the package ***scala.math***, that represents orderings:

```
trait Ordering[A] {
    def compare(a1: A, a2: A): Int
    def lt(a1: A, a2: A): Boolean = compare(a1, a2) <= 0
    …
}
```

- It has a single abstract method compare, which takes two values and returns a positive number if the first value is higher than the second, a negative number if the first value is lower than the second, or 0 if the two values are equal
- It also provides more convenient operations such as lt, which returns a Boolean indicating whether the first value is lower than the second value
- So, instead of parameterizing the method sort with the function lessThan, it can be refactored to take a parameter of type Ordering:

```
def sort[A](xs: List[A])(ord: Ordering[A]): List[A] = {
    …
    … if (ord.lt(x, y)) …
    …
}
```

- With this change, the sort method can be called like this:

```
sort(xs)(Ordering.Int)
sort(strings)(Ordering.String)
```

- This makes use of the values Int and String defined in the ***scala.math.Ordering***, which produces the right orderings on integers and strings
- Note that the symbols Int and String refer to values here, not types. In Scala, it is possible to use the same symbol for both types and values. Depending on the compiler deduces whether a symbol refers to a type or a value
- For the sake of completeness, here is how the Ordering.Int value is defined:

```
object Ordering {
    val Int = new Ordering[Int]{
        def compare(x: Int, y: Int) = if (x > y) 1 else if (x < y) -1 else 0
    }
}
```

Reducing Boilerplate

- The last version of the method sort works fine but is a bit cumbersome to use
- Indeed, sorting a collection of type List[Int] always uses the same Ordering.Int argument, sorting a collection of type List[String] always uses the same Ordering.String argument
- Passing the ordering argument is not just verbose, the problem is that the ordering argument can be systematically inferred from the type of the elements to sort
- Because it is so systematic, could it be automated instead?