Type-Directed Programming

- In the previous lesson we have seen how to write a general method for sorting lists:
  def sort[A](xs: List[A])(ord: Ordering[A]): List[A] = …
- But we noticed that passing around Ordering arguments is cumbersome:
  sort(xs)(Ordering.Int)
  sort(strings)(Ordering.String)
- In this lesson, we will see how we can make the compiler pass the ordering argument for us

Implicit Parameters

- The first step consists in indicating that we want the compiler to supply the ord argument by marking it as **implicit**:
  def sort[A](xs: List[A])(implicit ord: Ordering[A]): List[A] = …
- Then, calls to sort can omit the ord parameter and the compiler will try to infer it for us:
  sort(xs)
  sort(strings)
- The compiler infers the argument value based on its *expected type*
- Let's detail the steps that the compiler goes through, in order to infer the implicit parameter. Consider the following expression: ***sort(xs)***
- Since xs has type List[Int], the compiler fixes the type parameter A of sort to Int:
  sort[Int](xs)
- As a consequence, this also fixes the expected type of the ord parameter to Ordering[Int]
- The compiler looks for *candidate definitions* that match the expected type Ordering[Int]. In our case, the only matching candidate is the Ordering.Int definition. Thus, the compiler passes the value Ordering.Int to the method sort:
  sort[Int](xs)(Ordering.Int)
- Before we explain how candidate values are defined, let's state some facts about implicit parameters:
  - A method can have only one implicit parameter list and it must be the last parameter list given
  - At call site, the arguments of the given clause are usually left out, although it is possible to explicitly pass them:
    sort(xs) *// argument inferred by the compiler*
    sort(xs)(Ordering.Int.reverse) *// explicit argument*

Candidates for Implicit Parameters

- Where does the compiler look for *candidate definitions* when it tries to infer an implicit parameter of type T?
- The compiler searches for definitions that:
  - Have type T
  - Are marked implicit
  - Are visible at the point of the function call or are defined in a companion object *associated* with T

- If there is a single(most specific) definition, it will be taken as the actual argument for the implicit parameter. Otherwise an error is reported

Implicit Definitions

- An implicit definition is a definition qualified with the implicit keyword:

```
object Ordering {
    implicit val Int: Ordering[Int] = …
}
```

- The above code defines an implicit value of type Ordering[Int] named Int
- Any val, lazy val, def or object definition can be marked implicit
- Last but not least, implicit definitions can take type parameters and implicit parameters:

```
implicit def orderingPair[A, B](implicit
    orderingA: Ordering[A],
    orderingB: Ordering[B]
): Ordering[(A, B)] = …
```

Implicit Search Scope

- The search for an implicit value of type T first looks at all the implicit definitions that are *visible* (inherited, imported, or defined in an enclosing scope)
- If the compiler does not find an implicit instance matching the queried type T in the lexical scope, it continues searching in the *companion objects associated* with T
- A *companion* object is an object that has the same name as a type. For instance, the object scala.math.Ordering is the companion of type scala.math.Ordering
- The types associated with a type T are:
    - If T has parent types T_1 with T_2 … with T_n, the union of the parts of T_1, T_2 … T_n as well as T itself
    - If T is a parameterized type S[T_1, T_2 … T_n], the union of the parts of S and T_1, T_2 … T_n
    - Otherwise, just t itself
- As an example, consider the following type hierarchy:

```
trait Foo[A]
trait Bar[A] extends Foo[A]
trait Baz[A] extends Bar[A]
trait X
trait Y extends X
```

- If an implicit value of type **Bar[Y]** is required, the compiler will look for implicit definitions in the following companion objects:
    - **Bar**, because it is part of Bar[Y]
    - **Y**, because it is a part of Bar[Y]
    - **Foo**, because it is a parent type of Bar
    - **X**, because it is a parent type of Y
- However, the Baz companion object will not be visited

Implicit Search Process

- The search process can result into either no candidates found or at least one candidate found

- If there is no available implicit definition matching the queried type, an error is reported:
  def f(implicit n: Int) = ()
  f *// error: could not find implicit value for parameter n:Int*
- If more than one implicit definition is eligible, an **ambiguity** is reported:
  implicit val x: Int = 0
  implicit val y: Int = 1
  def f(implicit n: Int) = ()
  f *// error: ambiguous implicit values: both value x of type => Int and value y*
  // of type => Int match expected type Int
- Actually, several implicit definitions matching the same type don't generate an ambiguity if one is **more specific** than the other
- A definition a: A is more specific than a definition b: B if:
  - Type A has more "fixed" parts
  - Or, a is defined in a class or object which is a subclass of the class defining b

Example*: Which implicit definition matches the Int implicit parameter when the following method f is called?*

implicit def universal[A]: A = ???
implicit def int: Int = ???
def f(implicit n: Int) = ()
f

*In this case, because **universal** takes a type parameter and **int** doesn't, **int** has more fixed parts and is considered to be more specific than **universal**. Thus, there is no ambiguity and the compiler selects **int**.*

Example*: Which implicit definition matches the Int implicit parameter when the following method f is called?*

```
trait A {
  implicit val x: Int = 0
}
trait B extends A {
  implicit def y: Int = 1
  def f(implicit n: Int) = ()
  f
}
```

*Here, because **y** is defined in a trait that extends **A** (which is where **x** is defined), **y** is more specific than **x**. Thus, there is no ambiguity and the compiler selects **y**.*

Context Bounds

- Syntactic sugar allows the omission of the implicit parameter list:
  ```
  def printSorted[A: Ordering](as: List[A]): Unit = {
    println(sort(as))
  }
  ```
- Type parameter A has one **context bound**: Ordering. This is equivalent to writing:
  ```
  def printSorted[A](as: List[A])(implicit ev1: Ordering[A]): Unit = {
    println(sort(as))
  }
  ```

- More generally, a method definition such as: ***def f[A: U_1 … : U_n](ps): R = …***

Is expanded to: ***def f[A](ps)(implicit ev_1: U_1[A], … ev_n: U_n[A]): R = …***

Implicit Query

- At any point in a program, one can query an implicit value of a given type by calling the implicitly operation:

  implicitly[Ordering[Int]] // result(Ordering[Int]): scala.math.OrderingInt@...
- Note that implicitly is not a special keyword, it is defined as a library operation:

  def implicitly[A](implicit value: A): A = value

Summary

- In this lesson we have introduced the concept of **type-directed programming**, a language mechanism that infers **values** from **types**
- There has to be a **unique**(most specific) implicit definition matching the queried type for it to be selected by the compiler
- Implicit values are searched in the enclosing **lexical scope**(imports, parameters, inherited members) as well as in the **implicit scope** of the queried type
- The implicit scope of type is made of implicit values defined in **companion objects** of **types associated** with the queried type