Type Classes

- In the previous lectures we have seen a particular pattern of code combining parameterized types and implicits. We have defined a parameterized type Ordering[A], implicit instances of that type for concrete types A and implicit parameters of type Ordering[A]:

```
trait Ordering[A] {
  def compare(a1: A, a2: A): Int
}
object Ordering {
  implicit val Int: Ordering[Int] = new Ordering[Int] {
      def compare(x: Int, y: Int) = if (x < y) -1 else if (x > y) 1 else 0
  }
  implicit val String: Ordering[String] = new Ordering[String] {
      def compare(s: String, t: String) = s.compareTo(t)
  }
}
def sort[A: Ordering](xs: List[A]): List[A] = …
```

- We say that Ordering is a **type class**
- Type classes provide yet another form of polymorphism: The method sort can be called with lists containing elements of any type A for which there is an implicit value of type Ordering[A]
- At compile-time, the compiler resolves the specific Ordering implementation that matches the type of the list elements

Retroactive Extension

- Type classes let us add new features to data types without changing the original definition of these data types. For instance, consider the following Rational type, modeling a rational number:

```
case class Rational(numerator: Int, denominator: Int)
```

- We can add the capability "to be compared" to the type Rational by defining an implicit instance of type Ordering[Rational]:

```
object RationalOrdering {
  implicit val orderingRational: Ordering[Rational] =
      new Ordering[Rational] {
        def compare(q: Rational, r: Rational): Int =
          q.numerator * r.denominator – r.numerator * q.denominator
      }
}
```

Laws

- So far, we have shown how to implement instances of a type class, for some specific types (Int, String and Rational)
- Now, let's have a look at the other side: how to use (and reason about) type classes

- For example, the sort function is written in terms of the Ordering type class, whose implementation is itself defined by each specific instance, and is therefore unknown at the time the sort function is written. If an Ordering instance implementation is incorrect, then the sort function becomes incorrect too
- To prevent this from happening, type classes are often accompanied by **laws**, which describe properties that instances must satisfy and on which users of type classes can rely on
- Instances of the Ordering[A] type class must satisfy the following properties:
    - **Inverse**: the sign of the result of comparing x and y must be the inverse of the sign of the result of comparing y and x
    - **Transitive**: if a value x is lower than y and if also y is lower than z, then x must be lower than z
    - **Consistent**: if two values x and y are equal, then the sign of the result of comparing x and z should be the same as the sign of the result obtained when comparing y and z
- The authors of a type class should think about such kind of laws and they should provide ways for instance implementers to check that these laws are satisfied

Example of Type Class: Ring

- Let's see how we can define a type class modeling a ring structure. A ring is an algebraic structure defined as follows: *In mathematics, a **ring** is one of the fundamental algebraic structures used in abstract algebra. It consists of a set equipped with two binary operations that generalize the arithmetic operations of addition and multiplication. Though this generalization, theorems from arithmetic are extended to non-numerical objects such as polynomials, matrices and functions.*
- This structure is so common that, by abstracting over the ring structure, developers could write programs that could then be applied to various domains (arithmetic, polynomials, series, matrices and functions)
- A ring is a set equipped with two binary operations: + and *, satisfying the following laws (called the ring axioms):
    - **+ is associative**: *(a + b) + c = a + (b + c)*
    - **+ is commutative**: *a + b = b + a*
    - **0 is the additive identity**: *a + 0 = a*
    - **-a is the additive inverse of a**: *a + -a = 0*
    - *** is associative**: *(a * b) * c = a * (b * c)*
    - **1 is the multiplicative identity**: *a * 1 = a*
    - **left distribution**: *a * (b + c) = a * b + a * c*
    - **right distribution**: *(b + c) * a = b * a + c * a*
- Here is how we can define a ring type class in Scala:

```
trait Ring[A] {
  def plus(x: A, y: A): A
  def mult(x: A, y: A): A
  def inverse(x: A): A
  def zero: A
  def one: A
}
```

- Here is how we define an instance of Ring[Int]:

```
object Ring {
  implicit val ringInt: Ring[Int] = new Ring[Int] {
    def plus(x: Int, y: Int): Int = x + y
    def mult(x: Int, y: Int): Int = x * y
    def inverse(x: Int): Int = -x
    def zero: Int = 0
    def one: Int = 1
  }
}
```

- Finally, this is how we would define a function that checks that the + associativity law is satisfied by a given Ring instance:

```
def plusAssociativity[A](x: A, y: A, z: A)(implicit ring: Ring[A]): Boolean =
  ring.plus(ring.plus(x, y), z) == ring.plus(x, ring.plus(y, z))
```

- Note: in practice, the standard library already provides a type class Numeric, which models a ring structure

Summary

- In this lesson we have identified a new programming pattern: type classes
- Type classes provide a form of polymorphism: they can be used to implement algorithms that can be applied to various types. The compiler selects the type class implementation for a specific type at compile-time
- A type class definition is a trait that takes type parameters and defines operations that apply to these types. Generally, a type class definition is accompanied by laws, checking that implementations of their operations are correct