Type Classes

Effective Programming in Scala

Type Classes

In the previous lectures we have seen a particular pattern of code:

```
trait Ordering[A]:
  def compare(a1: A, a2: A): Int

object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int) =
      if x < y then -1 else if x > y then 1 else 0
  given String: Ordering[String] with
    def compare(s: String, t: String) = s.compareTo(t)

def sort[A](xs: List[A])(using Ordering[A]): List[A] = ...
```

We say that `Ordering` is a **type class**.

Exercise

Implement an instance of the `Ordering` type class for the `Rational` type.

```
/** A rational number
  * @param num   Numerator
  * @param denom Denominator
  */
case class Rational(num: Int, denom: Int)
```

Reminder:

let $q = \frac{num_q}{denom_q}$, $r = \frac{num_r}{denom_r}$,

$$q < r \Leftrightarrow \frac{num_q}{denom_q} < \frac{num_r}{denom_r} \Leftrightarrow num_q \times denom_r < num_r \times denom_q$$

Digression: Retroactive Extension

It is worth noting that we are able to implement the `Ordering[Rational]` instance without changing the `Rational` class definition.

Type classes support *retroactive* extension: the ability to extend a data type with new operations without changing the original definition of the data type.

In this example, we have added the capability of comparing `Rational` numbers.

Polymorphism

Type classes provide yet another form of polymorphism: the behavior of `sort` varies according to the type of elements in the list.

At compilation-time, the compiler resolves the specific `Ordering` implementation that matches the type of the list elements.

Type Classes vs Subtyping (1)

Subtyping also provides a way to specialize the behavior of a method according to the type of value it is applied to.

```
trait Comparable:
  def compareTo(that: Comparable): Int

case class Rational(num: Int, denom: Int) extends Comparable:
  def compareTo(that: Comparable): Int = ???

def sort(List[Comparable]): List[Comparable] = ???
```

In Scala, type classes are often preferred over subtyping to achieve polymorphism.

Type Classes vs Subtyping (2)

```
trait Ordering[A]:
  def compare(x: A, y: A): Int

trait Comparable:
  def compareTo(that: Comparable): Int
```

- In the first case, we compare `x` to `y`, whereas in the second case we compare `this` to `that`,
- A type class only defines operations about another type, but it does not define this type itself.

Summary

Type classes classify types by the operations they support.

In Scala, this mechanism is often preferred over subtyping to achieve polymorphism.

One reason for this is that type classes support retroactive extension.