Type Classes

Principles of Functional Programming

Martin Odersky and Julien Richard-Foy

Type Classes

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

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

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

Type Classes

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

In Scala, a type class is a generic trait that comes with given instances for type instances of that trait.

E.g., in the `Ordering` example, we have given instances for `Ordering[Int]` and `Ordering[String]`

Type Classes

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

In Scala, a type class is a generic trait that comes with given instances for type instances of that trait.

E.g., in the `Ordering` example, we have given instances for `Ordering[Int]` and `Ordering[String]`

Type classes provide yet another form of polymorphism:

The `sort` method can be called with lists containing elements of any type `A` for which there is a given instance of type `Ordering[A]`.

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

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

Exercise

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

```
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 were 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.

Conditional Instances

Question: How do we define an `Ordering` instance for lists?

Observation: This can be done only if the list elements have an ordering.

```
given listOrdering[A](using ord: Ordering[A]): Ordering[List[A]] with
```

Conditional Instances

Question: How do we define an `Ordering` instance for lists?

Observation: This can be done only if the list elements have an ordering.

```
given listOrdering[A](using ord: Ordering[A]): Ordering[List[A]] with

  def compare(xs: List[A], ys: List[A]) = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _)   => -1
    case (_, Nil)   => 1
    case (x :: xs1, y :: ys1) =>
      val c = ord.compare(x, y)
      if c != 0 then c else compare(xs1, ys1)
```

The given instance `listOrdering` takes type parameters and implicit parameters.

Conditional Instances

Given instances such as `listOrdering` that take implicit parameters are *conditional*:

- An ordering for lists with elements of type `T` exists only if there is an ordering for `T`.

This sort of conditional behavior is best implemented with type classes.

- Normal subtyping and inheritance cannot express this: a class either inherits a trait or doesn't.

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

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

sort(xss)
```

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

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

sort[List[Int]](xss)
```

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

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

sort[List[Int]](xss)(using listOrdering)
```

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

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

sort[List[Int]](xss)(using listOrdering(using Ordering.Int))
```

Exercise

Implement an instance of the `Ordering` typeclass for pairs of type `(A, B)`, where `A`, `B` have `Ordering` instances defined on them.

Example use case: Consider a program for managing an address book. We would like to sort the addresses by zip codes first and then by street name. Two addresses with different zip codes are ordered according to their zip code, otherwise (when the zip codes are the same) the addresses are sorted by street name. E.g.

```
type Address = (Int, String)  // Zipcode, Street Name
val xs: List[Address] = ...
sort(xs)
```

Exercise

Implement an instance of the `Ordering` typeclass for pairs of type `(A, B)`, where `A`, `B` have `Ordering` instances defined on them.

```
given pairOrdering[A, B](using orda: Ordering[A], ordb: Ordering[B])
 : Ordering[(A, B)] with
  def compare(x: (A, B), y: (A, B)) =
    val c = orda.compare(x._1, y._1)
    if c != 0 then c else ordb.compare(x._2, y._2)
```

Type Classes and Extension Methods

Like any trait, a type class trait may define extension methods.

For, instance, the `Ordering` trait would usually contain comparison methods like this:

```
trait Ordering[A]:

  def compare(x: A, y: A): Int

  extension (x: A)
    def <  (y: A): Boolean = compare(x, y) < 0
    def <= (y: A): Boolean = compare(x, y) <= 0
    def >  (y: A): Boolean = compare(x, y) > 0
    def >= (y: A): Boolean = compare(x, y) >= 0
```

Visibility of Extension Methods

Extension methods on a type class trait are visible whenever a given instance for the trait is available.

For instance one can write:

```
def merge[T: Ordering](xs: List[T], ys: List[T]): Boolean = (xs, ys) match
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

- There's no need to name and import the `Ordering` instance to get access to the extension method < on operands of type `T`.
- We have an `Ordering[T]` instance in scope, that's where the extension method comes from.

Summary

Type classes provide a way to turn types into values.

Unlike class extension, type classes

- can be defined at any time without changing existing code,
- can be conditional.

In Scala, type classes are constructed from parameterized traits and given instances.

Type classes give rise to a new kind of polymorphism, which is sometimes called *ad-hoc* polymorphism.

This means that the a type `TC[A]` has different implementations for different types `A`.