Type Classes vs Subtyping

Principles of Functional Programming

Type Classes

In the previous lectures we have seen that type classes could be used to achieve *ad hoc* polymorphism:

```
trait Ordering[A] { def lt(a1: A, a2: A): Boolean }

object Ordering {
  given Int: Ordering[Int]       = (x, y) => x < y
  given String: Ordering[String] = (s, t) => (s compareTo t) < 0
}

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

Alternatively, Using *Subtyping Polymorphism*

```
trait Ordered {
  def lt(other: Ordered): Boolean
}

def sort2(xs: List[Ordered]): List[Ordered] = {
  ...
  ... if x lt y then ...
  ...
}
```

Both `sort` and `sort2` can sort lists containing elements of an arbitrary type `A` as long as there is an implicit instance of `Ordering[A]`, or `A` is a subtype of `Ordered`, respectively.

How does `sort2` compare to `sort`?

Return Type is Less Precise

(Assuming that `Int <: Ordered`)

```
val ints: List[Int] = List(1, 3, 2)
val sortedInts: List[Int] = sort2(ints)
```

Return Type is Less Precise

(Assuming that `Int <: Ordered`)

```
val ints: List[Int] = List(1, 3, 2)
val sortedInts: List[Int] = sort2(ints)

                            ^^^^^^^^^^^
error: type mismatch;
 found   : List[Ordered]
 required: List[Int]
```

Return Type is Less Precise

(Assuming that `Int <: Ordered`)

```
val ints: List[Int] = List(1, 3, 2)
val sortedInts: List[Int] = sort2(ints)

                            ^^^^^^^^^^^
error: type mismatch;
 found   : List[Ordered]
 required: List[Int]

def sort2(xs: List[Ordered]): List[Ordered]
```

First Improvement: Introduce Type Parameter A

```
def sort2[A <: Ordered](xs: List[A]): List[A] = // ... same implementation
```

First Improvement: Introduce Type Parameter A

```
def sort2[A <: Ordered](xs: List[A]): List[A] = // ... same implementation

val sortedInts:   List[Int]    = sort2(ints)    // OK
val sortedString: List[String] = sort2(strings) // OK
```

… assuming `Int <: Ordered` and `String <: Ordered`!

Subtyping Polymorphism Causes Strong Coupling

Subtyping polymorphism imposes a strong coupling between *operations* (`Ordered`) and *data types* (`Int`, `String`) that support them.

By contrast, type classes encourage separating the definition of data types and the operations they support: the `Ordering[Int]` and `Ordering[String]` instances don't have to be defined in the same unit as the `Int` and `String` data types.

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

Implementing an Operation for a Custom Data Type

Here is how we can add the comparison operations to the `Rational` type.

```
case class Rational(num: Int, denom: Int)
```

Implementing an Operation for a Custom Data Type

Here is how we can add the comparison operations to the `Rational` type.

```
case class Rational(num: Int, denom: Int)

object Rational {
  given Ordering[Rational] =
    (x, y) => x.num * y.denom < y.num * x.denom
}

sort(List(Rational(1, 2), Rational(1, 3)))
```

(The `Ordering[Rational]` given definition could be in a different project)

Alternatively, Using Subtyping Polymorphism

Let's see what happens with the subtyping approach, if we make `Rational` extend `Ordered`:

```
case class Rational(num: Int, denom: Int) extends Ordered {
  def lt(other: Ordered) = other match {
    case Rational(otherNum, otherDenom) => num * otherDenom < otherNum * denom
    case _ => ???
  }
}
```

Alternatively, Using Subtyping Polymorphism

Let's see what happens with the subtyping approach, if we make `Rational` extend `Ordered`:

```
case class Rational(num: Int, denom: Int) extends Ordered {
  def lt(other: Ordered) = other match {
    case Rational(otherNum, otherDenom) => num * otherDenom < otherNum * denom
    case _ => ???
  }
}
```

We want to compare rational numbers with other rational numbers only!

Second Improvement: F-Bounded Polymorphism

```
trait Ordered[This <: Ordered[This]] {
  def lt(other: This): Boolean
}

case class Rational(num: Int, denom: Int) extends Ordered[Rational] {
  def lt(other: Rational) = num * other.denom < other.num * denom
}

def sort2[A <: Ordered[A]](xs: List[A]): List[A] = {
  ...
  ... if x lt y then ...
  ...
}
```

Subtyping vs Type Classes

Subtyping alone is less practical than type classes to achieve polymorphism.

Often, you also need to introduce bounded type parameters, or even f-bounded type parameters.

When Are Operation Implementations Resolved?

Another difference between subtyping polymorphism and type classes is **when** the implementation of an operation is resolved.

Type Class Operations Resolution

Remind the `sort` definition:

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

The implicit `Ordering[A]` parameter is resolved by the compiler at **compilation time**

Subtype Operations Resolution

Remind the (simplest) `sort2` definition:

```
def sort2(xs: List[Ordered]): List[Ordered] = {
  ...
  ... if x lt y then ...
  ...
}
```

The implementation of the `lt` operation is resolved at **run time** by selecting the implementation of the actual type of `x`

When Are Operation Implementations Resolved?

- With type classes, the implicit parameter is resolved at **compilation time**,
- With subtyping, the actual implementation of a method is resolved at **run time**.

Summary

In this lecture we have seen that type classes make it possible to decouple data type definitions from the operations they support. In particular, it is possible to retroactively add new operations to a data type.

Conversely, it is possible to write polymorphic programs that work with any data type supporting these operations.

The operation implementations are resolved at compile-time with type classes, whereas they are resolved at run-time with subtyping.