Type-Directed Programming — Motivating Example

Principles of Functional Programming

Type-Directed Programming: Intuition

We have seen that the compiler is able to *infer types* from *values*:

```
val x = 42        > x: Int = 42
val y = x + 1     > y: Int = 43
```

The Scala compiler is also able to do the opposite, namely to *infer values* from *types*.

Why is this useful?

Type-Directed Programming: Intuition

We have seen that the compiler is able to *infer types* from *values*:

```
val x = 42        > x: Int = 42
val y = x + 1     > y: Int = 43
```

The Scala compiler is also able to do the opposite, namely to *infer values* from *types*.

Why is this useful?

When there is exactly one "obvious" value for a type, the compiler can provide that value to us.

Let's look at a motivating example

Sorting Lists of Numbers

Consider a method `sort` that takes as parameter a `List[Int]` and returns another `List[Int]` containing the same elements, but sorted:

```
def sort(xs: List[Int]): List[Int] = {
  ...
  ... if x < y then ...
  ...
}
```

At some point, this method has to compare two elements `x` and `y` of the given list.

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `A` for the type of elements:

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

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `A` for the type of elements:

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

But this does not work, because the comparison < is not defined for all arbitrary types `A`.

Parameterization of `sort`

The most flexible design is to pass the comparison operation as an additional parameter:

```
def sort[A](xs: List[A])(lessThan: (A, A) => Boolean): List[A] = {
  ...
  ... if lessThan(x, y) then ...
  ...
}
```

Calling Parameterized sort

We can now call sort as follows:

```
scala> val xs = List(-5, 6, 3, 2, 7)
scala> val strings = List("apple", "pear", "orange", "pineapple")

scala> sort(xs)((x, y) => x < y)
res0: List[Int] = List(-5, 2, 3, 6, 7)

scala> sort(strings)((s1, s2) => s1.compareTo(s2) < 0)
res1: List[String] = List(apple, orange, pear, pineapple)
```

Parameterization with Ordering

There is already a class in the standard library that represents orderings:

```
scala.math.Ordering[A]
```

Provides ways to compare elements of type `A`. So, instead of parameterizing with the `lessThan` function, we could parameterize with `Ordering` instead:

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

Ordering Instances

Calling the new `sort` can be done like this:

```
import scala.math.Ordering

sort(xs)(Ordering.Int)
sort(strings)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

```
object Ordering {
  val Int = new Ordering[Int] {
    def compare(x: Int, y: Int) = x - y
  }
}
```

Reducing Boilerplate

Problem: Passing around `Ordering` arguments is cumbersome.

```
sort(xs)(Ordering.Int)
sort(ys)(Ordering.Int)
sort(strings)(Ordering.String)
```

Sorting a `List[Int]` value always uses the same `Ordering.Int` argument, sorting a `List[String]` value always uses the same `Ordering.String` argument, and so on…

Type-Directed Programming

Principles of Functional Programming

Reminder: General `sort` Operation

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

Problem: passing around `Ordering` arguments is cumbersome.

```
sort(xs)(Ordering.Int)
sort(ys)(Ordering.Int)
sort(strings)(Ordering.String)
```

Sorting a `List[Int]` value always uses the same `Ordering.Int` argument, sorting a `List[String]` value always uses the same `Ordering.String` argument, and so on…

Implicit Parameters

We can reduce the boilerplate by making `ord` an **inferred parameter**.

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

Then, calls to `sort` can omit the `ord` parameter:

```
sort(xs)
sort(ys)
sort(strings)
```

The compiler infers the argument value based on its expected type.

Implicit Parameters (2)

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

val xs: List[Int] = ...
```

Implicit Parameters (2)

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

val xs: List[Int] = ...

sort(xs)
```

Implicit Parameters (2)

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

val xs: List[Int] = ...

sort(xs)

sort[Int](xs)
```

Implicit Parameters (2)

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

val xs: List[Int] = ...

sort(xs)

sort[Int](xs)
```

In this case, the type of `ord` is `Ordering[Int]`.

Implicit Parameters (2)

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

val xs: List[Int] = ...

sort(xs)

sort[Int](xs)
```

In this case, the type of `ord` is `Ordering[Int]`.

```
sort[Int](xs)(given Ordering.Int)
```

(assuming there exists a **given instance** of type `Ordering[Int]` named `Ordering.Int`)

Given Clauses Syntax Reference (1)

There can be several `given` parameter clauses in a definition and `given` parameter clauses can be freely mixed with normal ones.

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

At call site, the arguments of the given clause are usually left out, although it is possible to explicitly pass them:

```
// Argument inferred by the compiler
sort(xs)

// Explicit argument
sort(xs)(given Ordering.Int.reverse)
```

Given Clauses Syntax Reference (2)

Multiple parameters can be in a `given` clause:

```
def f(x: Int)(given foo: Foo, bar: Bar) = ...
```

Or, there can be several `given` clauses in a row:

```
def f(x: Int)(given foo: Foo)(given bar: Bar) = ...
```

Given Clauses Syntax Reference (3)

Parameters of a given clause can be anonymous:

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

This is useful if the body of `sort` does not mention `ord` at all, but simply relies on the fact that there is an available given instance of type `Ordering[A]`.

Implicit Parameters Resolution

Say, a function takes an inferred parameter of type `T`.

The compiler will search a **given instance** that:

- has a type compatible with `T`,
- is visible at the point of the function call, or is defined in a companion object *associated* with `T`.

If there is a single (most specific) instance, it will be taken as actual arguments for the inferred parameter.

Otherwise it's an error.

Given Instances

For the previous example to work, the `Ordering.Int` definition must be a `given` instance:

```
object Ordering {

  given Int: Ordering[Int] {
    def compare(x: Int, y: Int): Int = ...
  }

}
```

This code defines a given instance of type `Ordering[Int]`, named `Int`.

Given Instances Syntax Reference

Given instances can be anonymous. Just omit the instance name:

```
given Ordering[Int] { ... }
```

Given instances can take type parameters and implicit parameters:

```
given [A, B](given Ordering[A], Ordering[B]): Ordering[(A, B)] { ... }
```

An alias can be used to define a given instance:

```
given Ordering[Int] = ...
```

Given Instances Search Scope

The search for a given instance of type `T` includes:

- all the given instances that are visible (inherited, imported, or defined in an enclosing scope),
- the *given instances search scope* of type `T`, made of given instances found in a companion object *associated* with `T`. In essence, the types associated with a type `T` are:
  - if `T` is a compound type $T_1 with T_2 ... with T_n$, the union of the parts of T_1, ... 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_n,
  - otherwise, just `T` itself.

Import Givens

Because given arguments are not visible to the programmer it might be hard to see where they come from.

To alleviate this problem, wildcard imports don't bring given instances to the lexical scope:

```
import scala.math.Ordering._
```

This line does *not* import the given instances of the `Ordering` object.

Instead, programmers have to explicitly import given instances:

```
import scala.math.Ordering.given
```

This line imports *only* the given instances of the `Ordering` object.

Import Givens (2)

You can be more specific in which type of given instances you want to import:

```
import scala.math.Ordering.{given Ordering[?]}
```

This line imports only given instances of type `Ordering[?]`.

Companion Objects Associated With a Queried Type

If the compiler does not find a given instance matching the queried type `T` in the lexical scope, it continues searching in the companion objects associated with `T`.

Consider the following hierarchy:

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

If a given instance of type `Bar[Y]` is required, the compiler will look into the companion objects `Bar`, `Y`, `Foo`, and `X` (but not `Baz`).

Exercise

```
val xs = List(3, 1, 2)
sort(xs)
```

In the above example of the `sort` method call, where does the compiler find the given instance of type `Ordering[Int]`?

o In the enclosing scope

o Via a given import

o In a companion object associated with the type `Ordering[Int]`

Exercise

```
val xs = List(3, 1, 2)
sort(xs)
```

In the above example of the `sort` method call, where does the compiler find the given instance of type `Ordering[Int]`?

o In the enclosing scope

o Via a given import

x In a companion object associated with the type `Ordering[Int]`

- The given instance is found in the `Ordering` companion object

No Given Instance Found

If there is no available given instance matching the queried type, an error is reported:

```
scala> def f(given n: Int) = ()
scala> f
       ^
error: no implicit argument of type Int was found for parameter n of method f
```

Ambiguous Given Instances

If more than one given instances are eligible, an **ambiguity** is reported:

```
scala> given x: Int = 0
scala> given y: Int = 1
scala> def f(given n: Int) = ()
scala> f
       ^
error: ambiguous implicit arguments:
 both value x and value y
 match type Int of parameter n of method f
```

Priorities

Actually, several given instances matching the same type don't generate an ambiguity if one is **more specific** than the other.

In essence, a `given a: A` definition is more specific than a `given b: B` definition if:

- type `A` is a subtype of type `B`,
- type `A` has more "fixed" parts,
- `a` is defined in a class or object which is a subclass of the class defining `b`,
- `a` is in a closer lexical scope than `b`.

Priorities: Example (1)

Which given instance matches the `Int` implicit parameter when the `f` method is called?

```
given universal[A]: A = ???
given int: Int = ???

def f(given n: Int) = ()

f
```

Priorities: Example (2)

Which given instance matches the `Int` implicit parameter when the `f` method is called?

```
trait A {
  given x: Int = 0
}
trait B extends A {
  given y: Int = 1

  def f(given n: Int) = ()

  f
}
```

Priorities: Example (3)

Which implied instance matches the queried `Int` implicit parameter when the `f` method is called?

```
given x: Int = 0
locally {
  given y: Int = 1

  def f(given n: Int) = ()

  f
}
```

Context Bounds

A syntactic sugar allows the omission of the given clause:

```
def printSorted[A: Ordering](as: List[A]): Unit = {
  println(sort(as))
}
```

Type parameter `A` has one **context bound**: `Ordering`. There must be a given instance of type `Ordering[A]` at the point of application.

More generally, a method definition such as:

$def\ f[A : U_1 ... : U_n](ps) : R = ...$

Is expanded to:

$def\ f[A](ps)(given U_1[A], ..., U_n[A]) : R = ...$

Implicit Query

At any point in a program, one can **query** a given instance of a specific type by calling the `summon` operation:

```
scala> summon[Ordering[Int]]
res0: Ordering[Int] = scala.math.Ordering$Int$@73564ab0
```

`summon` is not a special keyword, it is defined as a library operation:

```
def summon[A](given value: A): value.type = value
```

Summary

In this lecture 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) given instance matching the queried type for it to be used by the compiler.

Given instances are searched in the enclosing **lexical scope** (imports, parameters, inherited members) as well as in the **given instances search scope** made of given instances defined in companion objects of types associated with the queried type.

Type Classes

Principles of Functional Programming

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] { def compare(x: Int, y: Int) = x - y }
  given String: 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**.

Polymorphism

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]`.

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.

```
/** 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 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.

Exercise

Implement an instance of the `Ordering` typeclass for pairs of type `(A, B)`.

Example of use: 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.

Using Type Classes

So far we have shown how to add new instances of a type class, for a specific type.

Now, let's focus on 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 problem to happen, type classes are often accompanied by **laws**, which describe properties that instances must satisfy, and that users of type classes can rely on.

Example: Laws of the `Ordering` Type Class

Which properties do we want instances of the `Ordering` type class to satisfy so that we can implement the `sort` function?

Example: Laws of the `Ordering` Type Class

Which properties do we want instances of the `Ordering` type class to satisfy so that we can implement the `sort` function?

$x < y \wedge y < z \implies x < z$ (transitivity)

Example of Type Class: Ring

According to Wikipedia:

> *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. Through this generalization, theorems from arithmetic are extended to non-numerical objects such as polynomials, series, matrices and functions.*

By abstracting over the ring structure, developers could write programs that could then be applied to various domains (arithmetic, polynomials, series, matrices and functions).

Ring Laws

A ring is a set equipped with two binary operations, $+$ and $*$, satisfying the following laws (called the ring axioms):

$(a+b)+c=a+(b+c)$	($+$ is associative)
$a+b=b+a$	($+$ is commutative)
$a+0=a$	(0 is the additive identity)
$a+-a=0$	($-a$ is the additive inverse of a)
$(a*b)*c=a*(b*c)$	($*$ is associative)
$a*1=a$	(1 is the multiplicative identity)
$a*(b+c)=a*b+a*c$	(left distributivity)
$(b+c)*a=b*a+c*a$	(right distributivity)

Exercise

Define a `Ring` Type Class.

Implement a function that checks that the + associativity law is satisfied by a given `Ring` instance.

`Numeric` Type Class

In practice, the standard library already provides a type class `Numeric`, which models a ring structure.

EPFL

Higher-Order Givens

Principles of Functional Programming

Higher-Order Given Instances (1)

Consider how we order two `String` values:

- ”abc” < ”abd”?

Higher-Order Given Instances (1)

Consider how we order two `String` values:

- ”abc” < ”abd”?

We compare the characters of each string, element-wise.

Higher-Order Given Instances (1)

Consider how we order two `String` values:

- "abc" < "abd"?

We compare the characters of each string, element-wise.

Problem: How to generalize this process to sequences of any element type `A` for which there is a given `Ordering[A]` instance?

Higher-Order Given Instances (2)

```
given listOrdering[A](given Ordering[A]): Ordering[List[A]] { ... }
```

Higher-Order Given Instances (3)

```
given listOrdering[A](given ord: Ordering[A]): Ordering[List[A]] {
  def compare(xs0: List[A], ys0: List[A]) = {
    def loop(xs: List[A], ys: List[A]): Int = (xs, ys) match {
      case (x :: xsTail, y :: ysTail) =>
        val c = ord.compare(x, y)
        if c != 0 then c else loop(xsTail, ysTail)
      case (xs, ys) =>
        if xs.isEmpty then (if ys.nonEmpty then -1 else 0) else 1
    }
    loop(xs0, ys0)
  }
}

scala> sort(List(List(1, 2, 3), List(1), List(1, 1, 3)))
res0: List[List[Int]] = List(List(1), List(1, 1, 3), List(1, 2, 3))
```

Higher-Order Given Instances (4)

```
def sort[A](xs: List[A])(given Ordering[A]): List[A]
given listOrdering[A](given Ordering[A]): Ordering[List[A]]

val xss: List[List[Int]] = ...
sort(xss)
```

Higher-Order Given Instances (4)

```
def sort[A](xs: List[A])(given Ordering[A]): List[A]
given listOrdering[A](given Ordering[A]): Ordering[List[A]]

val xss: List[List[Int]] = ...
sort(xss)

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

Higher-Order Given Instances (4)

```
def sort[A](xs: List[A])(given Ordering[A]): List[A]
given listOrdering[A](given Ordering[A]): Ordering[List[A]]

val xss: List[List[Int]] = ...
sort(xss)

sort[List[Int]](xss)

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

Higher-Order Given Instances (4)

```
def sort[A](xs: List[A])(given Ordering[A]): List[A]
given listOrdering[A](given Ordering[A]): Ordering[List[A]]

val xss: List[List[Int]] = ...
sort(xss)

sort[List[Int]](xss)

sort[List[Int]](xss)(given listOrdering)

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

Higher-Order Given Instances (5)

An arbitrary number of given instances can be combined until the search hits a “terminal” instance:

```
given a: A = ...
given aToB(given A): B = ...
given bToC(given B): C = ...
given cToD(given C): D = ...

summon[D]
```

Recursive Given Instances

```
trait A
given loop(given a: A): A = a

summon[A]
```

Recursive Given Instances

```
trait A
given loop(given a: A): A = a

summon[A]

       ^
```

error: no implicit argument of type A was found for parameter x of method summon. I found:

```
 loop(/* missing */summon[A])
```

But method loop produces a diverging implicit search when trying to match type A.

Summary

In this lecture, we have seen:

- given instance definitions can also have given clauses
- an arbitrary number of given instances can be chained until a terminal instance is reached

Extension Methods

Principles of Functional Programming

Extension Methods: Motivation

In the previous lectures, we have seen that type classes could be used to retroactively add operations to existing data types.

However, when the operations are defined outside of the data types, they can't be called like methods on these data type instances.

```
case class Rational(num: Int, denom: Int)
given Ordering[Rational] = ...
val x: Rational = ...
val y: Rational = ...

x < y
//   ^
// value '<' is not a member of 'Rational'
```

Extension Methods (1)

Extension methods make it possible to add methods to a type after the type is defined.

Extension Methods (1)

Extension methods make it possible to add methods to a type after the type is defined.

```
def (lhs: Rational) < (rhs: Rational): Boolean =
  lhs.num * rhs.denom < rhs.num * lhs.denom

val x: Rational = ...
val y: Rational = ...

x < y // It works!
```

Extension Methods (2)

```
def (lhs: Rational) < (rhs: Rational): Boolean = ...
```

- An extension method definition is a method definition with a parameter clause **before** the method name,
- The leading parameter clause must have exactly one parameter,
- Extension methods are applicable if they are visible (by being defined, inherited, or imported) in a scope enclosing the point of the application.

Given Extension Methods (1)

How can we add the < operation to any type `A` for which there is a given `Ordering[A]` instance?

Given Extension Methods (1)

How can we add the < operation to any type `A` for which there is a given `Ordering[A]` instance?

```
def [A](lhs: A) < (rhs: A)(given ord: Ordering[A]): Boolean =
  ord.lt(lhs, rhs)
```

Given Extension Methods (1)

How can we add the < operation to any type `A` for which there is a given `Ordering[A]` instance?

```
def [A](lhs: A) < (rhs: A)(given ord: Ordering[A]): Boolean =
  ord.lt(lhs, rhs)
```

At application site, the compiler will infer the `ord` argument according to the operands type.

Given Extension Methods (2)

Alternatively, the < extension method could be directly defined in the `Ordering` type class:

```
trait Ordering[A] {
  def (lhs: A) < (rhs: A): Boolean
}
```

Given Extension Methods (2)

Alternatively, the < extension method could be directly defined in the `Ordering` type class:

```
trait Ordering[A] {
  def (lhs: A) < (rhs: A): Boolean
}
```

Such extension methods are applicable if there is a given instance visible at the point of the application.

Complete Example (1)

```
trait Ordering[A] {
  def (lhs: A) < (rhs: A): Boolean
}
```

Complete Example (2)

```
object Ordering {
  given Ordering[Int] {
    def (lhs: Int) < (rhs: Int) = lhs < rhs
  }
  given [A: Ordering]: Ordering[List[A]] {
    def (lhs: List[A]) < (rhs: List[A]) = {
      def loop(xs: List[A], ys: List[A]): Boolean = (xs, ys) match {
        case (x :: xsT, y :: ysT) =>
          if x < y then true
          else if y < x then false
          else loop(xsT, ysT)
        case (xs, ys) => xs.isEmpty && ys.nonEmpty
      }
      loop(lhs, rhs)
    }
```

Complete Example (3)

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

object Rational {
  given Ordering[Rational] {
    def (lhs: Rational) < (rhs: Rational) =
      lhs.num * rhs.denom < rhs.num * lhs.denom
  }
}
```

Complete Example (4)

```
import Ordering.given

val i  = 1
val j  = 2
val p  = Rational(1, 2)
val q  = Rational(1, 3)
val xs = List(1, 2, 3)
val ys = List(1, 2, 4)

i < j   // true
p < q   // false
xs < ys // true
```

Summary

In this lecture, we have seen:

- how to define extension methods outside of a type definition,
- how to define extension methods for type class instances.

Implicit Conversions

Principles of Functional Programming

Implicit Conversions

Implicit conversions make it possible to convert an expression to a different type.

This mechanism is usually used to provide more ergonomic APIs.

Implicit Conversions

Implicit conversions make it possible to convert an expression to a different type.

This mechanism is usually used to provide more ergonomic APIs.

Example: API for defining JSON documents.

```
// { "name": "Paul", "age": 42 }
Json.obj("name" -> "Paul", "age" -> 42)
```

Type Coercion: Motivation (1)

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json
```

Type Coercion: Motivation (1)

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json

// { "name": "Paul", "age": 42 }
JObject("name" -> JString("Paul"), "age" -> JNumber(42))
```

Problem: Constructing JSON objects is too verbose.

Type Coercion: Motivation (2)

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json

// { "name": "Paul", "age": 42 }
Json.obj("name" -> "Paul", "age" -> 42)
```

How could we support the above user-facing syntax?

Type Coercion: Motivation (3)

```
// { "name": "Paul", "age": 42 }
Json.obj("name" -> "Paul", "age" -> 42)
```

What could be the type signature of the `obj` constructor?

Type Coercion: Motivation (3)

```
// { "name": "Paul", "age": 42 }
Json.obj("name" -> "Paul", "age" -> 42)
```

What could be the type signature of the `obj` constructor?

```
def obj(fields: (String, Any)*): Json
```

Type Coercion: Motivation (3)

```
// { "name": "Paul", "age": 42 }
Json.obj("name" -> "Paul", "age" -> 42)
```

What could be the type signature of the `obj` constructor?

```
def obj(fields: (String, Any)*): Json
```

Allows invalid JSON objects to be constructed!

```
Json.obj("name" -> ((x: Int) => x + 1))
```

We want invalid code to be signaled to the programmer with a compilation error.

Type Coercion (1)

```
object Json {

  def obj(fields: (String, JsonField)*): Json =
    JObject(fields.map(_.json): _*)

  class JsonField(val json: Json)

}
```

Type Coercion (2)

```
class JsonField(val json: Json)

object JsonField {
  given stringToJsonField: Conversion[String, JsonField] {
    def apply(s: String) = new JsonField(JsString(s))
  }
  given intToJsonField: Conversion[Int, JsonField] {
    def apply(n: Int) = new JsonField(JNumber(n))
  }
  ...
  given jsonToJsonField: Conversion[JSon, JsonField] {
    def apply(j: Json) = new JsonField(j)
  }
}
```

Type Coercion: Usage

```
Json.obj("name" -> "Paul", "age" -> 42)
```

Type Coercion: Usage

```
Json.obj("name" -> "Paul", "age" -> 42)
```

The compiler implicitly inserts the following conversions:

```
Json.obj(
  "name" -> Json.JsonField.stringToJsonField.apply("Paul"),
  "age" -> Json.JsonField.intToJsonField.apply(42)
)
```

Implicit Conversions

The compiler looks for implicit conversions on an expression `e` of type `T` if `T` does not conform to the expression's expected type `S`.

In such a case, the compiler looks in the implicit scope for a given instance of type `Conversion[T, S]`.

Note: at most one implicit conversion can be applied to a given expression.

Summary

- Implicit conversions can improve the ergonomics of an API

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.