



香港科技大學

THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 12)

COMP 3031, Fall 2025

Lionel Parreaux

Contextual Abstraction

*what comes with the text,
but is not in the text*

Context Takes Many Forms

- ▶ the current configuration
- ▶ the current scope
- ▶ the meaning of “<” on this type
- ▶ the user on behalf of which the operation is performed
- ▶ the security level in effect
- ▶ the program’s current state
- ▶ ...

Code becomes more modular if it can *abstract* over context.

That is, functions and classes can be written without knowing in detail the context in which they will be called or instantiated.

How Is Context Represented?

So far:

- ▶ global values
- ▶ global mutable variables
- ▶ “Monkey Patching”
- ▶ dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- ▶ global values *i.e., no abstraction - this is often too rigid*
- ▶ global mutable variables
- ▶ “Monkey Patching”
- ▶ dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- ▶ global values *i.e., no abstraction - this is often too rigid*
- ▶ global mutable variables *what if different modules need different settings? interference can be dangerous!*
- ▶ “Monkey Patching”
- ▶ dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- ▶ global values *i.e., no abstraction - this is often too rigid*
- ▶ global mutable variables *what if different modules need different settings? interference can be dangerous!*
- ▶ “Monkey Patching” - *more powerful ways to shoot yourself in the foot...*
- ▶ dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- ▶ global values *i.e., no abstraction - this is often too rigid*
- ▶ global mutable variables *what if different modules need different settings? interference can be dangerous!*
- ▶ “Monkey Patching” - *more powerful ways to shoot yourself in the foot...*
- ▶ dependency injection frameworks (e.g. Spring, Guice) - *outside the language, rely on bytecode rewriting → harder to understand and debug.*

Functional Context Representation

In functional programming, the natural way to abstract over context is with function, class, or trait parameters.

- + flexible
- + types are checked
- + not relying on side effects

But sometimes this is too much of a good thing! It can lead to

- many function arguments
- which hardly ever change
- repetitive, errors are hard to spot

Example: Sorting

We have seen sort functions. For instance, here's an outline of 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 T for the type of elements:

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

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 T for the type of elements:

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

But this does not work, because there's not a single comparison method < that works for all types.

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 T for the type of elements:

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

But this does not work, because there's not a single comparison method < that works for all types.

In other words, we need to ask the question: What is the meaning of < on type T *at the call site*?

This means querying the call-site context.

Parameterization of sort

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

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

Calling Parameterized sort

We can now call sort as follows:

```
val ints = List(-5, 6, 3, 2, 7)
```

```
val strings = List("apple", "pear", "orange", "pineapple")
```

```
sort(ints)((x, y) => x < y)
```

```
sort(strings)((s1, s2) => s1.compareTo(s2) < 0)
```


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[T](xs: List[T])(ord: Ordering[T]): List[T] =  
  ...  
  ... if ord.compare(x, y) < 0 then ...  
  ...
```

Ordering Instances

Calling the new sort can be done like this:

```
import scala.math.Ordering

sort(ints)(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) =
      if x < y then -1 else if x > y then 1 else 0
```

Reducing Boilerplate

Problem: Passing around Ordering arguments is cumbersome.

```
sort(ints)(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 *implicit parameter*.

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

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

```
sort(ints)  
sort(strings)
```

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

Type Inference

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

That is, the previous calls to `sort` are augmented as follows:

```
sort(ints)
sort(strings)
```

Type Inference

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

That is, the previous calls to `sort` are augmented as follows:

```
sort[Int](ints)
sort[String](strings)
```

Term Inference

The Scala compiler is also able to do the opposite, namely to *infer expressions* (aka terms) from *types*.

When there is exactly one “obvious” value for a type in a using clause, the compiler can provide that value to us.

```
sort[Int](ints)  
sort[String](strings)
```

Term Inference

The Scala compiler is also able to do the opposite, namely to *infer expressions* (aka terms) from *types*.

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

```
sort[Int](ints)(using Ordering.Int)
sort[String](strings)(using Ordering.String)
```


Using Clauses

Using Clauses

An implicit parameter is introduced by a using parameter clause:

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

A matching explicit argument can be passed in a using argument clause:

```
sort(strings)(using Ordering.String)
```

But the argument can also be left out (and it usually is).

If the argument is missing, the compiler will infer one from the parameter type.

```
sort(strings)
```

Using Clauses Syntax Reference

Multiple parameters can be in a using clause:

```
def f(x: Int)(using a: A, b: B) = ...  
f(x)(using a, b)
```

Or, there can be several using clauses in a row:

```
def f(x: Int)(using a: A)(using b: B) = ...
```

using clauses can also be freely mixed with regular parameters:

```
def f(x: Int)(using a: A)(y: Boolean)(using b: B) = ...  
f(x)(using a)(y)(using b)
```

Anonymous Using Clauses

Parameters of a using clause can be anonymous:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] =  
  ... merge(sort(fst), sort(snd)) ...
```

```
def merge[T](xs: List[T])(using Ordering[T]): List[T] = ...
```

This is useful if the body of `sort` does not mention `ord` at all, but simply passes it on as an implicit argument to further methods.

Anonymous Using Clauses

Parameters of a using clause can be anonymous:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] =  
  ... merge(sort(fst), sort(snd)) ...
```

```
def merge[T](xs: List[T])(using Ordering[T]): List[T] = ...
```

This is useful if the body of sort does not mention ord at all, but simply passes it on as an implicit argument to further methods.

Equivalent to:

```
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] =  
  ... merge(sort(fst)(using ord), sort(snd)(using ord))(using ord) ...
```

```
def merge[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

Instead of:

```
def printSorted[T](xs: List[T])(using Ordering[T]) =  
  println(sort(xs))
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

With a context bound:

```
def printSorted[T: Ordering](xs: List[T]) =  
  println(sort(xs))
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

With a context bound:

```
def printSorted[T: Ordering](xs: List[T]) =  
  println(sort(xs))
```

More generally, a method definition such as:

$$\text{def } f[T : U_1 \dots : U_n](ps) : R = \dots$$

is expanded to:

$$\text{def } f[T](ps)(\text{using } U_1[T], \dots, U_n[T]) : R = \dots$$

Given Instances

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

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

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

Anonymous Given Instances

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

```
given Ordering[Double] with  
  def compare(x: Double, y: Double): Int = ...
```

The compiler will synthesize a name for an anonymous instance:

```
given given_Ordering_Double: Ordering[Double] with  
  def compare(x: Double, y: Double): Int = ...
```

Summoning an Instance

One can refer to a (named or anonymous) instance by its type:

```
summon[Ordering[Int]]  
summon[Ordering[Double]]
```

These expand to the equivalent of:

```
Ordering.Int  
Ordering.given_Ordering_Double
```

Summoning an Instance

One can refer to a (named or anonymous) instance by its type:

```
summon[Ordering[Int]]  
summon[Ordering[Double]]
```

These expand to the equivalent of:

```
Ordering.Int  
Ordering.given_Ordering_Double
```

summon is a predefined method. It can be defined like this:

```
def summon[T](using x: T) = x
```

Implicit Parameter Resolution

Say, a function takes an implicit 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 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 found in a companion object *associated* with T .

The definition of *associated* is quite general. Besides the companion object of a class itself, the compiler will also consider

- ▶ companion objects associated with any of T 's inherited types
- ▶ companion objects associated with any type argument in T
- ▶ if T is an inner class, the outer objects in which it is embedded.

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

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

Importing Given Instances

Since given instances can be anonymous, how can they be imported?

In fact, there are three ways to import a given instance.

1. By-name:

```
import scala.math.Ordering.Int
```

2. By-type:

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

3. With a wildcard:

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

Since the names of givens don't really matter, the second form of import is preferred since it is most informative.

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(using 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 instance is eligible, an *ambiguity* is reported:

```
trait C:  
  val x: Int  
given c1: C with  
  val x = 1  
given c2: C with  
  val x = 2
```

```
f(using c: C) = ()
```

```
f
```

```
^
```

error: ambiguous `implicit` arguments:

both value c1 and value c2

`match type C` of parameter c 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 definition

```
given a: A
```

is more specific than a definition

```
given b: B
```

if:

- ▶ a is in a closer lexical scope than b, or
- ▶ a is defined in a class or object which is a subclass of the class defining b, or
- ▶ type A is a generic instance of type B, or
- ▶ type A is a subtype of type B.

Priorities: Example (1)

Which given instance is summoned here?

```
class A[T](x: T)
given universal[T](using x: T): A[T](x)
given specific: A[Int](2)
```

```
summon[A[Int]]
```

Priorities: Example (1)

Which given instance is summoned here?

```
class A[T](x: T)
given universal[T](using x: T): A[T](x)
given specific: A[Int](2)
```

```
summon[A[Int]]
```

Answer: the most *specific* one, namely, 'specific'.

Priorities: Example (2)

Which given instance is summoned here?

```
trait A:  
  given ac: C  
trait B extends A:  
  given bc: C  
object O extends B:  
  val x = summon[C]
```


Priorities: Example (2)

Which given instance is summoned here?

```
trait A:  
  given ac: C  
trait B extends A:  
  given bc: C  
object O extends B:  
  val x = summon[C]
```

Answer: the *most recently defined* one, namely, 'this.bc'.

Priorities: Example (3)

Which given instance is summoned here?

```
given ac: C
def f() =
  given b: C
  def g(using c: C) = ()
  g
```

Priorities: Example (3)

Which given instance is summoned here?

```
given ac: C
def f() =
  given b: C
  def g(using c: C) = ()
  g
```

Answer: the *innermost* one, namely, b.

Summary so far

We have introduced a way to do *type-directed programming*, with the help of 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 companion objects associated with the queried type.

Type Classes

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:

$$\text{let } q = \frac{\text{num}_q}{\text{denom}_q}, r = \frac{\text{num}_r}{\text{denom}_r},$$

$$q < r \Leftrightarrow \frac{\text{num}_q}{\text{denom}_q} < \frac{\text{num}_r}{\text{denom}_r} \Leftrightarrow \text{num}_q \times \text{denom}_r < \text{num}_r \times \text{denom}_q$$

Exercise (Solution)

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

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

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

given RationalOrdering: Ordering[Rational] with
  def compare(x: Rational, y: Rational) =
    val xn = x.numer * y.denom
    val yn = y.numer * x.denom
    if xn < yn then -1 else if xn > yn then 1 else 0
```

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[Int])
```

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[Int](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 (Solution)

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 a type `TC[A]` has different implementations for different types `A`.

How Other Languages Do It

How Other Languages Do It

Haskell

In Haskell, *type classes* are a built-in concept. It is less general than Scala's contextual parameters but also easier to reason about.

Example (simplified):

```
data Ordering = LT | EQ | GT
class Ord a where compare :: a -> a -> Int
```

How Other Languages Do It

Haskell

In Haskell, *type classes* are a built-in concept. It is less general than Scala's contextual parameters but also easier to reason about.

Example (simplified):

```
data Ordering = LT | EQ | GT
class Ord a where compare :: a -> a -> Int
```

Successor Languages

Modern languages like Rust copied Haskell by supporting very similar language facilities: Rust “traits” are similar to Haskell type classes.

```
pub trait Ord: Eq + PartialOrd {
    fn cmp(&self, other: &Self) -> Ordering; }
```

Other languages, like Agda, Lean and (soon) OCaml, implement general contextual parameters, which can be used to similarly encode type classes.