Scala 3 Reference / Contextual Abstractions / Using Clauses

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

Using Clauses

Edit this page on GitHub

Functional programming tends to express most dependencies as simple function parameterization. This is clean and powerful, but it sometimes leads to functions that take many parameters where the same value is passed over and over again in long call chains to many functions. Context parameters can help here since they enable the compiler to synthesize repetitive arguments instead of the programmer having to write them explicitly.

For example, with the given instances defined previously, a `max` function that works for any arguments for which an ordering exists can be defined as follows:

```scala
def max[T](x: T, y: T)(using ord: Ord[T]): T =
  if ord.compare(x, y) < 0 then y else x
```

Here, `ord` is a *context parameter* introduced with a `using` clause. The `max` function can be applied as follows:

```scala
max(2, 3)(using intOrd)
```

The `(using intOrd)` part passes `intOrd` as an argument for the `ord` parameter. But the point of context parameters is that this argument can also be left out (and it usually is). So the following applications are equally valid:

```scala
max(2, 3)
max(List(1, 2, 3), Nil)
```

Anonymous Context Parameters

In many situations, the name of a context parameter need not be mentioned explicitly at all, since it is used only in synthesized arguments for other context parameters. In that case one can avoid defining a parameter name and just provide its type. Example:

```
def maximum[T](xs: List[T])(using Ord[T]): T =
  xs.reduceLeft(max)
```

`maximum` takes a context parameter of type `Ord[T]` only to pass it on as an inferred argument to `max`. The name of the parameter is left out.

Generally, context parameters may be defined either as a full parameter list `(p_1: T_1, ..., p_n: T_n)` or just as a sequence of types `T_1, ..., T_n`. Vararg parameters are not supported in `using` clauses.

Class Context Parameters

If a class context parameter is made a member by adding a `val` or `var` modifier, then that member is available as a given instance.

Compare the following examples, where the attempt to supply an explicit `given` member induces an ambiguity:

```
class GivenIntBox(using val givenInt: Int):
  def n = summon[Int]

class GivenIntBox2(using givenInt: Int):
  given Int = givenInt
  //def n = summon[Int] // ambiguous
```

The `given` member is importable as explained in the section on importing `given`s:

```
val b = GivenIntBox(using 23)
import b.given
summon[Int]  // 23

import b.*
//givenInt // Not found
```

Inferring Complex Arguments

Here are two other methods that have a context parameter of type `Ord[T]`:

```
def descending[T](using asc: Ord[T]): Ord[T] = new Ord[T]:
  def compare(x: T, y: T) = asc.compare(y, x)

def minimum[T](xs: List[T])(using Ord[T]) =
  maximum(xs)(using descending)
```

The `minimum` method's right-hand side passes `descending` as an explicit argument to `maximum(xs)`. With this setup, the following calls are all well-formed, and they all normalize to the last one:

```
minimum(xs)
maximum(xs)(using descending)
maximum(xs)(using descending(using listOrd))
maximum(xs)(using descending(using listOrd(using intOrd)))
```

Multiple `using` Clauses

There can be several `using` clauses in a definition and `using` clauses can be freely mixed with normal parameter clauses. Example:

```
def f(u: Universe)(using ctx: u.Context)(using s: ctx.Symbol, k: ctx.Kind) = ..
```

Multiple `using` clauses are matched left-to-right in applications. Example:

```
object global extends Universe { type Context = ... }
given ctx : global.Context with { type Symbol = ...; type Kind = ... }
given sym : ctx.Symbol
given kind: ctx.Kind
```

Then the following calls are all valid (and normalize to the last one)

```
f(global)
f(global)(using ctx)
f(global)(using ctx)(using sym, kind)
```

But `f(global)(using sym, kind)` would give a type error.

Summoning Instances

The method `summon` in `Predef` returns the given of a specific type. For example, the given instance for `Ord[List[Int]]` is produced by

```
summon[Ord[List[Int]]]  // reduces to listOrd(using intOrd)
```

The `summon` method is simply defined as the (non-widening) identity function over a context parameter.

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

Syntax

Here is the new syntax of parameters and arguments seen as a delta from the standard context free syntax of Scala 3. `using` is a soft keyword, recognized only at the start of a parameter or argument list. It can be used as a normal identifier everywhere else.

```
ClsParamClause      ::=  ... | UsingClsParamClause
DefParamClauses     ::=  ... | UsingParamClause
UsingClsParamClause ::=  ‘(’ ‘using’ (ClsParams | Types) ‘)’
UsingParamClause    ::=  ‘(’ ‘using’ (DefParams | Types) ‘)’
ParArgumentExprs    ::=  ... | ‘(’ ‘using’ ExprsInParens ‘)’
```

‹ Given I...

Contex... ›

Contributors to this page

odersky pikinier20 BarkingBad julienrf b-studios som-snytt michelou