Relationship with Scala 2 Implicits

Many, but not all, of the new contextual abstraction features in Scala 3 can be mapped to Scala 2's implicits. This page gives a rundown on the relationships between new and old features.

Simulating Scala 3 Contextual Abstraction Concepts with Scala 2 Implicits

Given Instances

Given instances can be mapped to combinations of implicit objects, classes and implicit methods.

1. Given instances without parameters are mapped to implicit objects. For instance,

   ```
   given intOrd: Ord[Int] with { ... }
   ```

   maps to

   ```
   implicit object intOrd extends Ord[Int] { ... }
   ```

2. Parameterized givens are mapped to combinations of classes and implicit methods. For instance,

   ```
   given listOrd[T](using ord: Ord[T]): Ord[List[T]] with { ... }
   ```

   maps to

   ```
   class listOrd[T](implicit ord: Ord[T]) extends Ord[List[T]] { ... }
   final implicit def listOrd[T](implicit ord: Ord[T]): listOrd[T] =
     new listOrd[T]
   ```

```
new listOrd[T]
```

3. Alias givens map to implicit methods or implicit lazy vals. If an alias has neither type nor context parameters, it is treated as a lazy val, unless the right-hand side is a simple reference, in which case we can use a forwarder to that reference without caching it.

Examples:

```
given global: ExecutionContext = new ForkJoinContext()

val ctx: Context
given Context = ctx
```

would map to

```
final implicit lazy val global: ExecutionContext = new ForkJoinContext()
final implicit def given_Context = ctx
```

Anonymous Given Instances

Anonymous given instances get compiler synthesized names, which are generated in a reproducible way from the implemented type(s). For example, if the names of the `IntOrd` and `ListOrd` givens above were left out, the following names would be synthesized instead:

```
given given_Ord_Int: Ord[Int] with { ... }
given given_Ord_List[T](using ord: Ord[T]): Ord[List[T]] with { ... }
```

The synthesized type names are formed from

1. the prefix `given_`,
2. the simple name(s) of the implemented type(s), leaving out any prefixes,
3. the simple name(s) of the top-level argument type constructors to these types.

Tuples are treated as transparent, i.e. a type `F[(X, Y)]` would get the synthesized name `F_X_Y`. Directly implemented function types `A ⇒ B` are represented as `A_to_B`. Function types used as arguments to other type constructors are represented as `Function`.

Using Clauses

Using clauses correspond largely to Scala 2's implicit parameter clauses. E.g.

Using clauses correspond largely to Scala 2's implicit parameter clauses. E.g.

```
def max[T](x: T, y: T)(using ord: Ord[T]): T
```

would be written

```
def max[T](x: T, y: T)(implicit ord: Ord[T]): T
```

in Scala 2. The main difference concerns applications of such parameters. Explicit arguments to parameters of using clauses *must* be written using `(using ...)`, mirroring the definition syntax. E.g, `max(2, 3)(using IntOrd)`. Scala 2 uses normal applications `max(2, 3)(IntOrd)` instead. The Scala 2 syntax has some inherent ambiguities and restrictions which are overcome by the new syntax. For instance, multiple implicit parameter lists are not available in the old syntax, even though they can be simulated using auxiliary objects in the "Aux" pattern.

The `summon` method corresponds to `implicitly` in Scala 2. It is precisely the same as the `the` method in Shapeless. The difference between `summon` (or `the`) and `implicitly` is that `summon` can return a more precise type than the type that was asked for.

Context Bounds

Context bounds are the same in both language versions. They expand to the respective forms of implicit parameters.

Note: To ease migration, context bounds in Scala 3 map for a limited time to old-style implicit parameters for which arguments can be passed either in a using clause or in a normal argument list. Once old-style implicits are deprecated, context bounds will map to using clauses instead.

Extension Methods

Extension methods have no direct counterpart in Scala 2, but they can be simulated with implicit classes. For instance, the extension method

```
extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2
```

could be simulated to some degree by

```
implicit class CircleDecorator(c: Circle) extends AnyVal {
  def circumference: Double = c.radius * math.Pi * 2
}
```

Abstract extension methods in traits that are implemented in given instances have no direct counterpart in Scala 2. The only way to simulate these is to make implicit classes available through imports. The Simulacrum macro library can automate this process in some cases.

Type Class Derivation

Type class derivation has no direct counterpart in the Scala 2 language. Comparable functionality can be achieved by macro-based libraries such as Shapeless, Magnolia, or scalaz-deriving.

Context Function Types

Context function types have no analogue in Scala 2.

Implicit By-Name Parameters

Implicit by-name parameters are not supported in Scala 2, but can be emulated to some degree by the `Lazy` type in Shapeless.

Simulating Scala 2 Implicits in Scala 3

Implicit Conversions

Implicit conversion methods in Scala 2 can be expressed as given instances of the `scala.Conversion` class in Scala 3. For instance, instead of

```
implicit def stringToToken(str: String): Token = new Keyword(str)
```

one can write

```
given stringToToken: Conversion[String, Token] with
  def apply(str: String): Token = KeyWord(str)
```

or

```
given stringToToken: Conversion[String, Token] = KeyWord(_)
```

Implicit Classes

Implicit classes in Scala 2 are often used to define extension methods, which are

directly supported in Scala 3. Other uses of implicit classes can be simulated by a pair of a regular class and a given `Conversion` instance.

Implicit Values

Implicit `val` definitions in Scala 2 can be expressed in Scala 3 using a regular `val` definition and an alias given. For instance, Scala 2's

```
lazy implicit val pos: Position = tree.sourcePos
```

can be expressed in Scala 3 as

```
lazy val pos: Position = tree.sourcePos
given Position = pos
```

Abstract Implicits

An abstract implicit `val` or `def` in Scala 2 can be expressed in Scala 3 using a regular abstract definition and an alias given. For instance, Scala 2's

```
implicit def symDecorator: SymDecorator
```

can be expressed in Scala 3 as

```
def symDecorator: SymDecorator
given SymDecorator = symDecorator
```

Implementation Status and Timeline

The Scala 3 implementation implements both Scala 2's implicits and the new abstractions. In fact, support for Scala 2's implicits is an essential part of the common language subset between 2.13 and Scala 3. Migration to the new abstractions will be supported by making automatic rewritings available.

Depending on adoption patterns, old style implicits might start to be deprecated in a version following Scala 3.0.

‹ By-Na... Metap... ›

 b-studios  asakaev  ShapelessCat  nicolasstucki  SrTobi 

Scaladoc