TOUR OF SCALA

IMPLICIT PARAMETERS

A method can have an *implicit* parameter list, marked by the *implicit* keyword at the start of the parameter list. If the parameters in that parameter list are not passed as usual, Scala will look if it can get an implicit value of the correct type, and if it can, pass it automatically.

The places Scala will look for these parameters fall into two categories:

- Scala will first look for implicit definitions and implicit parameters that can be accessed directly (without a prefix) at the point the method with the implicit parameter block is called.
- Then it looks for members marked implicit in all the companion objects associated with the implicit candidate type.

A more detailed guide to where Scala looks for implicits can be found in the FAQ.

In the following example we define a method `sum` which computes the sum of a list of elements using the monoid's `add` and `unit` operations. Please note that implicit values cannot be top-level.

```scala
abstract class Monoid[A] {
  def add(x: A, y: A): A
  def unit: A
}

object ImplicitTest {
  implicit val stringMonoid: Monoid[String] = new Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }

  implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }

  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))

  def main(args: Array[String]): Unit = {
    println(sum(List(1, 2, 3)))       // uses intMonoid implicitly
    println(sum(List("a", "b", "c"))) // uses stringMonoid implicitly
  }
}
```

`Monoid` defines an operation called `add` here, that combines a pair of `A` s and returns another `A`, together with an operation called `unit` that is able to create some (specific) `A`.

To show how implicit parameters work, we first define monoids `stringMonoid` and `intMonoid` for strings and integers, respectively. The `implicit` keyword indicates that the corresponding object can be used implicitly.

The method `sum` takes a `List[A]` and returns an `A`, which takes the initial `A` from `unit`, and combines each next `A` in the list to that with the `add` method. Making the parameter `m` implicit here means we only have to provide the `xs` parameter when we call the method if Scala can find an implicit `Monoid[A]` to use for the implicit `m` parameter.

In our `main` method we call `sum` twice, and only provide the `xs` parameter. Scala will now look for an implicit in the scope mentioned above. The first call to `sum` passes a `List[Int]` for `xs`, which means that `A` is `Int`. The implicit parameter list with `m` is left out, so Scala will look for an implicit of type `Monoid[Int]`. The first lookup rule reads

> *Scala will first look for implicit definitions and implicit parameters that can be accessed directly (without a prefix) at the point the method with the implicit parameter block is called.*

`intMonoid` is an implicit definition that can be accessed directly in `main`. It is also of the correct type, so it's passed to the `sum` method automatically.

The second call to `sum` passes a `List[String]`, which means that `A` is `String`. Implicit lookup will go the same way as with `Int`, but will this time find `stringMonoid`, and pass that automatically as `m`.

The program will output

```
6
abc
```

← **previous** **next** →

Contributors to this page:

ckipp01 mlachkar gmal1 ashawley Philippus Ivoah kiendang angadgill martijnhoekstra lierdakil heathermiller