Scala Implicits

Created By:
Neeraj Goel

Where Implicits are Tried (1) - Implicit conversion to an expected type

Whenever the compiler sees an X, but needs a Y, it will look for an implicit function that converts X's to Y's

Where Implicits are Tried (2) - Converting the receiver

Suppose you write down foo.doit, and foo does not have a method named doit. The compiler will try to insert conversions before giving up. In this case, the conversion needs to apply to the receiver, foo.

Two major applications

- Interoperating with new types
- Simulating new syntax

Where Implicits are Tried (3) - Implicit parameters

The compiler will sometimes replace foo(x) with foo(x)(y), or new Foo(x) with new Foo(x)(y), thus adding a missing parameter list to complete a function call. For this usage, not only must the inserted identifier (y) be marked implicit, but also the formal parameter list in foo’s or Foo's definition be marked as implicit

Use Case - The most common use of these implicit parameters is to provide information about a type

Rules for Implicit Conversions

Rule 1- Marking Rule: Only definitions marked implicit are available

. The implicit keyword is used to mark which declarations the compiler may use as implicits. The compiler will only change x + y to convert(x) + y if convert is marked as implicit.

Rule 2 - Scope Rule: An inserted implicit conversion must be a single identifier or be associated with the source or target type of the conversion.

The compiler will usually not insert a conversion of the form foo.convert. It will not expand x + y to foo.convert(x) + y. Any conversion must be available in the current scope via a single identifier. If you want to make foo.convert available as an implicit, then you need to import it.

There’s one exception to this “single identifier” rule. To pick up an implicit conversion the compiler will also look in the companion modules of the source or expected target types of the conversion. For instance, you could package an implicit conversion from X to Y in the companion module of class

Rule 3 - Non-ambiguity Rule

If the compiler has two options to fix x + y, say using either convert1(x) + y or convert2(x) + y,then it will report an error and refuse to choose between them. It

Rule 4 - One-at-a-time Rule

Only one implicit is tried.

The compiler will never convert x + y to convert1(convert2(x)) + y.

Debugging Help

If you run scalac with Xprint:typer option, then the compiler will show you what your code looks like after all implicit conversions have been added by the type checker.

References

https://booksites.artima.com/programming_in_scala_3ed