Scala Implicits

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

4 марта 2021

Scala function parameters

Parameter lists

Scala allows function definitions with multiple parameter lists

```
1 def multiply(a: Int, b: Int): Int
```

Parameter lists

Scala allows function definitions with multiple parameter lists

```
1 def multiply(a: Int, b: Int): Int
2 def multiply(a: Int)(b: Int): Int
```

Parameter lists

Scala allows function definitions with multiple parameter lists

```
1 def multiply(a: Int, b: Int): Int
2 def multiply(a: Int)(b: Int): Int
3
4 multiply(2,3)
5 multiply(2)(3)
```

Omitting parenthesis

If parameters to a function are not required at call-site, the parenthesis can be omited

```
1 def foo(x: Int = 123)()
2
3 foo() // OK(with a warning)
```

Implicit parameters

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 compiler will look if it can get an implicit value of the correct type, and if it can, pass it automatically.

Implicit parameters

Example

```
1 ???
2
3 def findInt(implicit int: Int): Int = int
4
5 findInt // res0 = 123
```

Implicit parameters

Example

```
1 implicit val i: Int = 123
2
3 def findInt(implicit int: Int): Int = int
4
5 findInt // res0 = 123
```

Trivia

- all parameters in a clause will be implicit
- only one implicit clause allowed per function
- implicit clause must be the last one
- all parameters must be found to omit the clause
- implicit parameters can be passed explicitly

Application rules

Say, a function takes an implicit parameter of type `T`.
The compiler will search an implicit definition that:

- is marked `implicit`
- 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) definition, it will be taken as actual argument for the implicit parameter. Otherwise it's an error.

Passing implicit parameters

implicit parameters keep their *implicitness* inside a function definition

```
1 def foo(i: Int)(implicit ctx: Context)    = bar(i.toString) // ctx is passed to bar
2 def bar(s: String)(implicit ctx: Context) = println(s)
3
4 implicit def c: Context = ???
5 foo(42)
```

Implicit definitions

Implicit definitions

Implicit parameter is substisuted as a value. Value definitions in Scala can be marked as implicit

```
1 trait Context
2
3 def foo(i: Int)(implicit ctx: Context)     = bar(i.toString)
4 def bar(s: String)(implicit ctx: Context) = println(s)
5
6 implicit val c: Context = new Context {}
7 foo(42)
```

Implicit definitions

Implicit parameter is substisuted as a value. Value definitions in Scala can be marked as implicit

```
1 trait Context
2
3 def foo(i: Int)(implicit ctx: Context)     = bar(i.toString)
4 def bar(s: String)(implicit ctx: Context) = println(s)
5
6 implicit def c: Context = new Context {}
7 foo(42)
```

Implicit definitions

Implicit parameter is substisuted as a value. Value definitions in Scala can be marked as implicit

```
1 trait Context
2
3 def foo(i: Int)(implicit ctx: Context)     = bar(i.toString)
4 def bar(s: String)(implicit ctx: Context) = println(s)
5
6 implicit object c extends Context {}
7 foo(42)
```

Definition restriction

`implicit` definitions cannot be top-level

```
1 //file Foo.scala
2 package com.foo.bar
3
4 implicit object Foo {
5  // ^--- error: `implicit' modifier cannot be used for top-level objects
6 }
```

Implicit Conversions

Implicit conversion

When a the provided type doesn't match the expected type, the Scala compiler looks for any method in scope marked implicit that takes the provided type as parameter and returns the expected type as a result. If found, it inserts the call to the method in between.

```
1 implicit def int2String(i: Int): String = i.toString()
2
3 123.startsWith("12")
```

Implicit conversion

When a the provided type doesn't match the expected type, the Scala compiler looks for any method in scope marked implicit that takes the provided type as parameter and returns the expected type as a result. If found, it inserts the call to the method in between.

```
1 implicit def int2String(i: Int): String = i.toString()
2
3 int2string(123).startsWith("12")
```

Implicit conversions

Limitations

Implicit conversions **cannot** be nested at *application-site*

but **can** be nested during *resolve*

```
1 case class A(i: Int)
2 case class B(i: Int)
3 case class C(i: Int)
4
5 implicit def aToB(a: A): B = B(a.i)
6 implicit def bToC(b: B): C = C(b.i)
7
8 val a = A(1)
9 val c: C = a // error: type mismatch, expected C got A
```

Implicit conversions

Limitations

Implicit conversions **cannot** be nested at *application-site*

but **can** be nested during *resolve*

```
1  case class A(i: Int)
2  case class B(i: Int)
3  case class C(i: Int)
4
5  implicit def aToB(a: A): B = B(a.i)
6  implicit def bToC[T](t: T)(implicit tToB: T => B): C = C(t.i)
7
8  val a = A(1)
9  val c: C = a // OK
```

Implicits class

An implicit class is a class marked with the `implicit` keyword. This keyword makes the class's primary constructor available for implicit conversions when the class is in scope

```
1 implicit class RichString(val str: String) {
2   def toUrl: URL = new URL(str)
3 }
4
5 "https://foo.com".toUrl
```

Implicits class

An implicit class is a class marked with the `implicit` keyword. This keyword makes the class's primary constructor available for implicit conversions when the class is in scope

```
1 implicit class RichString(val str: String) extends AnyVal {
2   def toUrl: URL = new URL(str)
3 }
4
5 "https://foo.com".toUrl
```

Implicits class

An implicit class is a class marked with the `implicit` keyword. This keyword makes the class's primary constructor available for implicit conversions when the class is in scope

```
1 implicit class RichString(val str: String) extends AnyVal {
2   def toUrl: URL = new URL(str)
3 }
4
5 new RichString("https://foo.com").toUrl
```

Implicit class

Limitations

1. They must be defined inside of another trait/class/object

```
1 object Helpers {
2    implicit class RichInt(x: Int) // OK!
3 }
4
5 implicit class RichDouble(x: Double) // BAD!
```

2. They may only take one non-implicit argument in their constructor

```
1 implicit class RichDate(date: java.util.Date) // OK!
2 implicit class Indexer[T](collection: Seq[T], index: Int) // BAD!
3 implicit class Indexer[T](collection: Seq[T])(implicit index: Index) // OK!
```

3. There may not be any method, member or object in scope with the same name as the implicit class

NB! *This means an implicit class cannot be a case class*

```
1 object Bar
2 implicit class Bar(x: Int) // BAD!
3
4 val x = 5
5 implicit class x(y: Int) // BAD!
6
7 implicit case class Baz(x: Int) // BAD!
```

Implicit Search

1 - Local scope

```scala
trait Show[A] { def show(a: A): String }

implicit val showInt: Show[Int] = new Show[Int] {
  def show(a: Int): String = a.toString
}

def stringify[A](a: A)(implicit s: Show[A]): String = s.show(a)
stringify(1)
```

2 - Explicit imports

```scala
trait Show[A] { def show(a: A): String }

def stringify[A](a: A)(implicit s: Show[A]): String = s.show(a)

import Show.instances.intShow
stringify(1)
```

3 - Wildcard imports

```scala
trait Show[A] { def show(a: A): String }

def stringify[A](a: A)(implicit s: Show[A]): String = s.show(a)

import Show.instances._
stringify(1)
```

4 - Companion object of type

or Implicit scope of an argument's type

```scala
trait Show[A] { def show(a: A): String }
case class Foo(a: Int)
object Foo {
  implicit val fooShow: Show[Foo] = {f: Foo => f.toString}
}

def stringify[A](a: A)(implicit s: Show[A]): String = s.show(a)
stringify(Foo(1))
```

5 - Implicit scope of type arguments

or Companion object of a Type Class

```scala
case class Foo(i: Int)
object Foo {
  implicit def showFoo: Show[Foo] = {f: Foo => f.i.toString}
}
trait Show[A] {def show(a: A): String}
object Show {
  implicit def showOpt[A](implicit s: Show[A]): Show[Option[A]] = ...
}
def stringify[A](a: A)(implicit s: Show[A]): String = s.show(a)
stringify(Option(Foo(1)))
```

Implicit's Application Patterns Intro

Context bounds are basically syntactic sugar for less verbose typeclass applications.

The following lines are identical:

```
1 def g[A : B](a: A) = implicitly[B[A]].h(a)
2 def g[A](a: A)(implicit ev: B[A]) = ev.h(a)
```

Library syntax

Split typeclasses and syntax enrichments via implicit conversions and context bounds

```
1  object Show {
2    object syntax {
3      implicit class ShowSyntax[A](val a: A) extends AnyVal {
4        def stringify(implicit s: Show[A]): String = s.show(a)
5  } } }
6
7  import Show.syntax._
8  def stringify[A: Show](a: A): String = a.stringify
9
10 stringify(1)
11 1.stringify
```

Context pattern

The most basic use of implicits is the Implicit Context pattern: using them to pass in some "context"object into all your methods.
This is something you could pass in manually, but is common enough that simply "not having to pass it everywhere"is itself a valuable goal.

Context pattern

Example

```
1 class Context
2 def foo(i: Int)(implicit context: Context): String = ???
3 def bar(s: String)(implicit context: Context): String = ???
4 def baz(i: Int)(implicit context: Context): String = bar(foo(i))
5
6 implicit def context = new Context
7 baz(1)
```

Implicit derivation

One neat thing about using Type-class Implicits is that you can perform "deep"checks.

```
1  case class Foo(i: Int)
2  trait Show[A] {def show(a: A): String}
3  object Show {
4    implicit def showFoo: Show[Foo] =
5      {f: Foo => f.i.toString}
6    implicit def showSeq[A: Show]: Show[Seq[A]] =
7      (a: Seq[A]) => a.map(x => implicitly[Show[A]].show(x)).mkString
8  }
9  def stringify[A](a: A)(implicit s: Show[A]): String = s.show(a)
10 stringify(Seq(Seq(Foo(1))))
```