Scala Course

Implicits

47 Deg

2021-02-15

Implicits

Implicits provide us the ability to **pass arguments** to function **without provide** them **explicitly**.

The simplest way for understanding them is to see a simple example:

```
implicit val name: String = "John"
// name: String = "John"

def printImplicitName(implicit name: String): Unit =
  println("Hello " + name)

printImplicitName
// Hello John
```

Implicits

Any value (vals, functions, objects, etc) can be declared to be implicit by using them, you guessed it, **implicit** keyword.

```
implicit val myImplicitInt = 1
implicit def myImplicitFunction(s: String) = s
implicit class MyImplicitClass(i: Int)
```

Implicits

By itself, **implicit** keyword doesn't change the behavior of the value, so above values can be used as usual.

```
myImplicitInt + 1
// res1: Int = 2
myImplicitFunction("hi")
// res2: String = "hi"
new MyImplicitClass(1)
// res3: MyImplicitClass = repl.MdocSession$App$MyImplic
```

Types of implicits

- Implicit conversions
- Type expansion
- Type class pattern

Implicit conversions

One of the most used features of implicits are **implicit conversions**. Using them, we'll be able to use values of a type as values of other type **if there's an implicit conversion between them**.

We'll declare implicit conversions as `implicit def`.

Implicit conversions

```scala
import java.util.UUID

implicit def uuidAsString(uuid: UUID): String =
  uuid.toString

val id: UUID = UUID.randomUUID
// id: UUID = 698f1f29-4033-4c1e-8e61-19d31415f1ef

def printString(str: String): Unit =
  println(str)

printString(id)
// 698f1f29-4033-4c1e-8e61-19d31415f1ef
```

Type expansion

We'll use **type expansion** when we want to **add new methods to types we don't control**.

As an example... let's copy Ruby's `n.times do...` pattern.

```
3.times {
  println("it worked!")
}
// error: value times is not a member of Int
// 3.times {
// ^^^^^^^
```

Type expansion

We can add new methods to a type we don't control using `implicit classes`:

```
implicit class IntTimes(val x: Int) {
  def times(action: => Unit): Unit =
    (1 to x).foreach(_ => action)
}

3.times {
  println("it worked!")
}
// it worked!
// it worked!
// it worked!
```

Type Class Pattern

We can go further, and not only expanding types but abstract types.

We need to see first what a **Typeclass** is, though.

Resolution order

When finding an implicit parameter, Scala will look for candidates for the given type in:

- Current scope
- Imports
- Companion objects
- Parameters arguments
- Type parameters
- ...

Check this reference in the Scala docs for more details.

Resolution order

Companion objects

```scala
class MyInt(val n: Int)
object MyInt {
  implicit def toStr(mi: MyInt) = s"MyInt: ${mi.n}"
}
val myInt = new MyInt(1)
// myInt: MyInt = repl.MdocSession$App$MyInt@409026c5
val str: String = myInt
// str: String = "MyInt: 1"
```

Resolution order

Companion objects

```
class MyInt(val n: Int)
object MyInt {
  implicit def toStr(mi: MyInt) = s"MyInt: ${mi.n}"
}
val myInt = new MyInt(1)
// myInt: MyInt = repl.MdocSession$App7$MyInt@197a6325
implicit def toNewStr(mi: MyInt) = s"MyNewInt: ${mi.n}"
val str: String = myInt
// str: String = "MyNewInt: 1"
```

Resolving ambiguity

- If there are more than one matching implicits, the most specific one is chosen
- If there is no unique most specific candidate, a compile error is reported

Resolving ambiguity

```scala
import scala.concurrent.Future
import scala.concurrent.{ExecutionContext => EC}
import scala.concurrent.ExecutionContext.Implicits.global
// ^ we import an EC which is implicit
def myFuture(s: String)(implicit ec: EC): Future[String] =
  Future(s) // Future.apply receives an implicit EC
myFuture("hi") // ???
```

Resolving ambiguity

```scala
import scala.concurrent.Future
import scala.concurrent.{ExecutionContext => EC}
import scala.concurrent.ExecutionContext.Implicits.global
// ^ we import an EC which is implicit
def myFuture(s: String)(implicit ec: EC): Future[String] =
  Future(s) // Future.apply receives an implicit EC
myFuture("hi")
// error: ambiguous implicit values:
//  both method global in object Implicits of type scala
//  and value ec of type scala.concurrent.ExecutionConte
//  match expected type scala.concurrent.ExecutionContex
//   Future(s) // Future.apply receives an implicit EC
//   ^^^^^^^^
```

Resolving ambiguity

```scala
import scala.concurrent.Future
import scala.concurrent.{ExecutionContext => EC}
import scala.concurrent.ExecutionContext.Implicits.global

def myFuture(s: String): Future[String] =
  Future(s)

myFuture("hi")
// res10: Future[String] = Future(Success(hi))
```

Resolving ambiguity

```scala
import scala.concurrent.Future
import scala.concurrent.{ExecutionContext => EC}

implicit val ec: EC = EC.global
// ec: concurrent.ExecutionContext = scala.concurrent.im

def myFuture(s: String)(implicit ec: EC): Future[String]
  Future(s)

myFuture("hi")
// res12: Future[String] = Future(Success(hi))
```

Resolving ambiguity

```
import scala.concurrent.Future
import scala.concurrent.{ExecutionContext => EC}

def myFuture(s: String)(implicit ec: EC): Future[String] =
  Future(s)

import scala.concurrent.ExecutionContext.Implicits.global
myFuture("hi")
// res14: Future[String] = Future(Success(hi))
```

Implicits

This is an overview.

Check this reference in the Scala docs for more details.