Oleksii Avramenko

Follow 706 Followers About

Implicits in Scala (2.12.2)

 Oleksii Avramenko Feb 20, 2017 · 4 min read

Scala's implicits have multiple applicable use cases which can serve different purposes. In this article we will go over some examples and try to understand how they can be useful. We will cover:

- Implicit parameters
- Type conversions (implicit functions)
- "Pimp my library" (implicit classes)
- Type classes (implicit objects)

Case 1: implicit parameters

Lets take a look at simplest example

```scala
implicit val bob = "Bob"

def greet(implicit name: String) = {
  println(s"Hello, $name!")
}

// usage
greet

// outputs "Hello, Bob!"
```

implicitvals.scala hosted with ♥ by GitHub view raw

Here `bob` will be implicitly passed into function `greet`. Missing parameters to the function call are looked up **by type in the current scope** meaning that code will not compile if there is no implicit variable of type `String` in the scope or there are multiple variables of the same type which will cause ambiguity:

```scala
implicit val bob = "Bob"
implicit val alice = "Alice"

def greet(implicit name: String) = {
  println(s"Hello, $name!")
}

// usage
greet

// ambiguous implicit values:
// both value bob in object Main of type => String
// and value alice in object Main of type => String
// match expected type String
//  greet
//  ^
```

implicitsambiguity.scala hosted with ♥ by GitHub view raw

Case 2: Type conversions with implicit functions

Implicit functions allow us to define conversions between types:

```
implicit def intToStr(num: Int): String = s"The value is $num"

42.toUpperCase() // evaluates to "THE VALUE IS 42"

def functionTakingString(str: String) = str

// note that we're passing int
functionTakingString(42) // evaluates to "The value is 42"
```

implicitfunctions.scala hosted with ♥ by GitHub view raw

When a compiler sees a type that is not expected in the evaluation context then it will try to find an implicit function in the current scope that can produce the expected type. In our example such contexts are expression `42.toUpperCase()` and a function call `functionTakingString(42)`. Function `toUpperCase()` is not a defined on integers so `intToStr` is considered as a conversion and code compiles. The implicit function name is not that important — only the function type signature, in our case its `(Int) => (String)`.

Case 3: “Pimp my library”

So, as we saw above, implicit function can convert some type `A` into type `B`. There is actually no constraints on the type `B`, it doesn’t have to be a primitive type, like in the example. Let’s say we have a simple class working on string:

```
case class StringOps(str: String) {
  def yell = str.toUpperCase() + "!"
  def isQuestion = str.endsWith("?")
```

We can write an implicit function that converts `String` into our `StringOps`.

```
case class StringOps(str: String) {
  def yell = str.toUpperCase() + "!"
  def isQuestion = str.endsWith("?")
}

implicit def stringToStringOps(str: String): StringOps = StringOps(str)

"Hello world".yell // evaluates to "HELLO WORLD!"
"How are you?".isQuestion // evaluates to 'true'
```

stringopswithimplicitfunction.scala hosted with ♥ by GitHub view raw

That allows us to call our functions on `String` as if they were part of `String` class.

Scala 2.10 introduced *implicit classes* that can help us reduce the boilerplate of writing implicit function for conversion.

```
object Helpers {
  implicit class StringOps(str: String) {
    def yell = str.toUpperCase() + "!"
    def isQuestion = str.endsWith("?")
  }
}

"Hello world".yell // evaluates to "HELLO WORLD!"
"How are you?".isQuestion // evaluates to 'true'
```

implicitclasses.scala hosted with ♥ by GitHub view raw

Note, that there are requirements for the class to be implicit:

- It has to be inside another trait, class or object
- It has to have exactly one parameter (but it can have multiple implicit parameters on its own)
- There may not be any method, member or object in scope with the same name

Case 4: Type classes

With *implicit objects* it is possible to implement *type classes* — a type system construct that supports ad hoc polymorphism. (To get more understanding about type classes and their purpose check this link).

Type class is somewhat similar to an interface which can have multiple implementations. In OOP languages those implementations are usually classes that extend the interface and are instantiated where needed. With type classes they have to be instantiated once and be 'globally' available. Singleton is a usual name for this pattern which scala natively supports with *object* declarations.

Typical example of type classes application is a Monoid implementation.

```scala
// Our interface
trait Monoid[A] {
  def zero: A
  def plus(a: A, b: A): A
}

// Implementation for integers
implicit object IntegerMonoid extends Monoid[Int] {
  override def zero: Int = 0
  override def plus(a: Int, b: Int): Int = a + b
}

// Implementation for strings
implicit object StringMonoid extends Monoid[String] {
  override def zero: String = ""
  override def plus(a: String, b: String): String = a.concat(b)
}

// Could be implementation for custom classes, etc..

// Our generic function that knows which implementation to use based on type parameter 'A'
def sum[A](values: Seq[A])(implicit ev: Monoid[A]): A = values.foldLeft(ev.zero)(ev.plus)
```

typeclasses.scala hosted with ♥ by GitHub view raw

and produces their sum but the “sum” can mean different things based on value types. If it’s integers then it’s just an addition, if strings — string concatenation, lists — lists concatenation. Information about which implementation to use comes in implicit parameter that is usually called “*ev*”. *ev* stands for *evidence* — an evidence that provided type `A` implements interface `Monoid`. It might be easier to think about evidence as a functional analogy for strategy pattern where we pass desired implementation into the function. We also doing it implicitly meaning that compiler will do all the work for you. If you don’t have an implementation for some type and you try to use it — the code won’t compile.

There is an alternative syntax for specifying implement parameters list:

Both definitions are equivalent but in the second case notation is a bit shorter. But we lost the name of an *evidence (implementation)* which we are referencing. There is a syntactic sugar to retrieve it — `implicitly`:

No magic here — `implicitly` is just a regular function in Predef.scala that basically takes a single implicit parameter, gives it a name and returns it. Looks like this:

Scala implicits are powerful features of the language which can be used in different context to achieve different goals. Comes without saying that because of it’s non explicit nature its easy to get things wrong so use it carefully.