Implicits

Scala Training, 2016
Contents

Intro

Implicit Conversions

Implicit Resolution

Implicit Classes

Implicit Parameters

Implicit Values
Teaser

Ever wondered how the following examples compile?

```scala
// Building a map entry
val entry = 2 -> "ABC"

// Building regexes
val uppercaseRegex = "[A-Z]".r

// String concatenation
val string = "Hello" + "world!"
```

- Lookup the String & Int classes, they don’t define the methods we use here
- Some voodoo magic however fits everything to compile
- Implicits in Scala:
  - Values that are passed transparently
  - Conversions between two types made automatically
Implicit Conversions

- The compiler fights type errors
- How does it fight? Accessing an unexisting member (i.e. ‘->’) causes the compiler to search for an implicit conversion to the receiver (RHS)
- The search is made in the implicit scope
What’s an implicit conversion?

- A method taking an argument of type S (source) and returning a type T (target)
- Implicit methods are prefixed with the `implicit` keyword

```
implicit def fromString2Int(source: String): Int =
  Integer parseInt source

val two = "4" / 2
```

- The compiler wraps the required code into a call to the implicit conversion
The Implicit Resolution Scopes

- **Current Scope**
  - Implicits defined in the current scope
  - Explicit imports
  - Wildcard imports

- **Implicit Scope**
  - Companion objects of a type
  - Implicit scope of an argument’s type
  - Implicit scope of type arguments
  - Outer objects for nested types
What if there are more implicits in scope?

- As a general rule, the current scope beats the implicit scope.
  - Local identifiers (in the enclosing code block)
  - Members of the enclosing scope (e.g. class members)
  - Imported modifiers (wildcard < explicit)
  - The most specific member from the implicit scope.

- Keep your implicits in your companions! Benefits:
  - No need to import them (they’re picked up from the implicit scope)
  - If needed you can easily override them locally
Precedence exemplified!

// User.scala
case class User(fName: String, lName: String, age: Int)
object User {
    implicit def toJson(user: User): String =
    s""""{
        "firstName":${user.fName},
        "lastName":${user.lName},
        "age":${user.age}
    }"""".stripMargin
}
// MyApp.scala
object MyApp extends App {
  def writeJson(json: String): Unit = println(json)

  implicit def toJson(user: User): String = 
  s"""""""""""""""""""""""""""""""
  |{"first_name": ${user.fName},
  | "last_name": ${user.lName},
  | "age": ${user.age}
  |}
  """".stripMargin

  // What’s the output of?
  writeJson(User("John", "Doe", 20))
}
What if we have multiple implicits defined?

THERE CAN BE ONLY ONE
Inheritance (kind of) without subclassing

- Create a wrapper around the desired class
- The extension methods should be defined in the wrapper
- Define an implicit conversion from the item to the wrapper
- Make the wrapper a value class for performance reasons

```scala
class RichStr(val s: String) extends AnyVal {
  def trimWhiteSpaces: String =
    s.replaceAll("\s+", "")
}
implicit def string2rich(s: String): RichStr =
  new RichStr(s)

// What will be printed?
println("boo ya".trimWhiteSpaces)
```
Implicit classes

- Scala 2.10 introduces the implicit classes
- You get the ‘pimp my library‘ stuff out of the box

```scala
implicit class RichStr(val s: String) extends AnyVal {
  def trimWhiteSpaces: String =
    s.replaceAll("\\s+", "")
}
println("boo ya").trimWhiteSpaces)
```
Type Classes

- Type classes from Haskell
- Force some type to conform to some interface
- It’s more powerful than inheritance / less coupling
- Use implicits to implement ad-hoc polymorphism

```scala
trait Show[T] {
  def show(t: T): String
}

object Show {
  // Define defaults in companion object
  implicit val showString = new Show[String] {
    override show(s: String) = s
  }
}
```
Providing operations on types

```scala
object Show {
  implicit class Ops[T](val t: T) {
    def show(implicit instance: Show[T]): String =
      instance.show(t)
  }
}
```
Implicit Parameters

- An implicit parameter list marks all parameters as implicit
- Methods have at most one implicit parameter list
  - It must be the last one
  - This applies to constructors as well

```scala
class HttpService(implicit val settings: HttpSettings) {
  def run(request: HttpRequest)
      (implicit ec: ExecutionContext): Future[_] = ???
}
```
Implicit Values

- The compiler tries to "fix" code using its implicit mechanism when:
  - Method / constructor calls have missing parameters
  - Methods are called with unexpected objects (conversions)
- The implicit resolution lookup is done in the same way for parameters
- Implicit values are usually picked for parameters

```scala
implicit val ec = scala.concurrent.ExecutionContext.Implicits.global
```
Type Constructors

- Higher-Kinded types are called type constructors
- They are used to build types
  - Complex types can be seen as much simpler types:
  - A $F[G[X], Y]$ can look like a $F[X]$

```scala
object HttpService extends App {
  type Callback[A] = PartialFunction[A, Unit]
  def printResult[A]: Callback[A] =
    { case r => println(r) }

  Http()
    .bindAndHandle(reject, "localhost", 8080)
    .onSuccess(printResult)
}
```
Implicit Sugar - Context Bounds

- Syntax sugar for omitting some implicit parameters
- Expresses a "has a" relation to a unary type constructor
- Use ':' to define
- The following are equivalent:

```scala
trait Max {
  def max[A : Ordering](a1: A, a2: A): A
  def max(a1: A, a2: A)(implicit o: Ordering[A]): A
}
```
Implicitly

- Use it to access an implicit value by its type
- It’s used to access the value of a hidden parameter of a context bound

```scala
object Predef {
  def implicitly[T](implicit e: T) = e
  // for summoning implicit values from the nether world
}

object Max {
  def max[A : Ordering](a1: A, a2: A): A =
    implicitly[Ordering[A]].max(a1, a2)
}
```