Updated: 2021-03-01

4.3 Implicits

Northeastern University

Implicits

- Remember from the first lecture: Odersky says that “implicits” are one of the major pillars of Scala
- See my Quora answer to Why should I learn Scala in 2018?

Implicits (1)

- What happens when you pass an *Int* to a method that expects a *Double*?

```
scala> def cToFConverter(c: Double) = 9*c/5+32
cToFConverter: (c: Double)Double

scala> cToFConverter(10)
res1: Double = 50.0
```

  - It just works! If you are coming from a Java background, this will be no big surprise (and no big deal). There's a set of language rules including that *int* will be "widened" to *double* if appropriate. But these rules are <u>arbitrarily</u> defined by the language designers.
  - In Scala, the designers wanted programmers to have more control over this type of thing: Scala has a much more general mechanism called "implicits."
  - Think about why you want to convert the type: because you need to invoke some method that is only available in the converted type.

Implicits (2)

- What about using someone else's date-time library that is written for a world-wide audience but in your application of it, you never have to worry about timezones. It's tedious having to pass in a *tz* parameter to all of the methods. And what if the library is all sealed traits and classes? You can't even add your own non-tz-dependent methods.
  - Scala allows you, as a library designer, to specify certain parameters like this as "implicit".
- **Implicits can be tricky!**

Implicits (3)

- Defining a method that adds two numbers:

```
def add(x: Int, y: Int): Int = x+y
val r = add("1","2")
```

Does not compile

- Defining an implicit converter:

```
scala> implicit def stringToInt(x: String) = x.toInt
stringToInt: (x: String)Int
scala> def add(x: Int, y: Int): Int = x+y
add: (x: Int, y: Int)Int
scala> add("1","2")
res0: Int = 3
```

- Definition must be:
  - a *val*, *def, class,* or a (final) parameter set of a method;
  - marked *implicit;*
  - in scope—scope rules for implicits are different: see Implicits (5);
  - a single identifier (not something like x.y);
  - non-ambiguous (exactly one implicit definition in scope);
  - non-pipelined, i.e. *x+y* can't be replaced by *conv1(conv2(x))+y.*

Implicits (4)

- Where can implicit conversions occur?
  - implicit conversion to an expected type: when compiler sees an *X* but needs a *Y*, it will look for an implicit *X=>Y.*
  - implicit conversion of a receiver: e.g *Y* has a method *value* but *X* does not. So, *X.value* will not compile. Unless you provide an implicit *X=>Y*.
  - implicit parameter sets: a method call *value(x,y)* can be converted to *value(x,y)(z)* if the method is defined thus:

```
def add(x: Int, y: Int)(implicit z: Int): Int = x+y+z
                                                //> add: (x: Int, y: Int)(implicit z: Int)Int
implicit def stringToInt(x: String) = x.toInt;  //> stringToInt: (x: String)Int
implicit val z: Int = 4                         //> z  : Int = 4
val r = add("1","2")                            //> r  : Int = 7
```

  - implicit parameter sets are always:
    - an entire parameter set
    - the last parameter set
    - marked “implicit”

Implicits (5)

- Here’s an example where we define the *locale* implicitly:

```
package edu.neu.coe.scala.scaladate
import java.util.{Date,Locale}
import java.text.DateFormat
import java.text.DateFormat._
trait LocaleDependent {
  def toStringForLocale(implicit locale: Locale): String
}
case class ScalaDate(date: Date) extends LocaleDependent {
  import ScalaDate.locale
  def toStringForLocale(implicit locale: Locale): String = 
getDateInstance(LONG,locale) format date
  override def toString: String = toStringForLocale(locale)
}
object ScalaDate {
  def apply(): ScalaDate = ScalaDate(new Date)
  implicit def locale = Locale.FRANCE
}
```

- In the REPL:

```
scala> ScalaDate()
res1: ScalaDate = 1 octobre 2015
```

Implicits (6)

- Scope rules for implicits:
  - In the *current* scope, an implicit must be declared <u>above</u> the place it is to be used. Important!
    - Basically, it's the same as the rule for *vals*, but it applies to <u>every</u> implicit object, including those defined by *def* or *class*.
    - If the implicit is actually defined somewhere else that is not in scope, then you can get it into scope using *import*.
  - An implicit involving a class *C* may be found in the companion object of *C*.

Implicits (7)

- You can even have implicit classes!
  - Constructor must have exactly one parameter: this is the value that will be “converted” implicitly into an instance of the class.
  - Example: Benchmark class:

```
object Benchmark extends App {
 implicit class Rep(n: Int) {
       /**
     * Method which can be invoked, provided that Benchmark._ has been imported.
     * See for example BenchmarkSpec
     * @param f the function to be invoked
     * @tparam A the result type of f
     * @return the average number of nano-seconds per run
     */
   def times[A](f: => A): Double = {
    // Warmup phase: do at least 20% of repetitions before starting the clock
    1 to (1+n/5) foreach (_ => f)
        val start = System.nanoTime()
        1 to n foreach (_ => f)
        (System.nanoTime() - start) / n.toDouble
      }
    }
   println(s"ave time for 40! is ${10000.times(Factorial.factorial(40))} nanosecs")
}
```

Implicits (8)

- A more common example of this is *StringOps*.
- There are many methods that you would like to have in a *String* but, because it is *final* (as in Java), you cannot add any of your own behavior to the *String* class.
  - For example, suppose you want to create a padding string of exactly n spaces?
  - There's no good way to do this with *String*.
  - But, in Scala, you can just write::

```scala
val n: Int = ???
val padding = " " * n
```

- A new *StringOps(" ")* is constructed and its * method is invoked with parameter n.

Polymorphism

- Polymorphism is perhaps the most important aspect of object-oriented programming:
  - It allows us to refer to an individual instance of something using a label (interface, super-type, whatever) that is less specific, i.e. more generic, than the actual instance.
  - This allows for things like dependency injection and factories.
  - It's also the basis for encapsulation (although that's possibly an orthogonal concept in O-O).

Polymorphism (2)

- In a purely object-oriented language, polymorphism is implemented via inheritance, i.e. by sub-typing:
  - A class extends another class;
  - A class implements an interface;
  - An interface extends another interface.
- There are times when this won’t work very well:
  - The superclass is final;
  - You can extend a class from a third-party library (e.g. open source) but unfortunately, a new version of that library makes nonsense of your sub-class;
  - When it just doesn’t make sense: i.e. it’s better to use composition rather than inheritance.

Polymorphism (3)

- Typeclasses are the functional-programming way of accomplishing this notion of polymorphism:
  - Let's say you have in mind a trait but there's nothing appropriate for it to extend:

```scala
trait Parseable[T] {
    def parse(s: String): Try[T]
}
object Parseable {
    trait ParseableInt extends Parseable[Int] {
        def parse(s: String): Try[Int] = Try(s.toInt)
    }
    implicit object ParseableInt extends ParseableInt
}
object TestParseable {
    def parse[T : Parseable](s: String): Try[T] = implicitly[Parseable[T]].parse(s)
}
```

This form is called a "context bound". But we can also write it as follows:

```scala
    def parse[T](s: String)(implicit ev: Parseable[T]):
Try[T] = ev.parse(s)
```

  - What we are doing here is adding the behavior of *Parseable* to type *T* without requiring *T* to extend anything (without using inheritance).
    - Note that you cannot add a context bound to a trait. Why not?

Polymorphism (3)

- Typeclasses are the functional-programming way of accomplishing this notion of polymorphism:
  - Let's say you have in mind a trait but there's nothing appropriate for it to extend:

```
trait Parseable[T] {
    def parse(s: String): Try[T]
}
object Parseable {
    trait ParseableInt extends Parseable[Int] {
        def parse(s: String): Try[Int] = Try(s.toInt)
    }
    implicit object ParseableInt extends ParseableInt
}
object TestParseable {
    def parse[T : Parseable](s: String): Try[T] = implicitly[Parseable[T]].parse(s)
}
```

This form is called a "context bound". But we can also write it as follows:

```
    def parse[T](s: String)(implicit ev: Parseable[T]):
Try[T] = ev.parse(s)
```

  - What we are doing here is adding the <u>behavior</u> of *Parseable* to type *T* without requiring *T* to <u>extend</u> anything (without using inheritance).
    - Note that you cannot add a context bound to a trait. Why not?

Sorting

- Unlike in Java where we need an explicit *Comparable* (or *Comparator*), ordering in Scala is done implicitly.

```
scala> List(1,3,2).sorted
res2: List[Int] = List(1, 2, 3)
```

Use *Ordering* since 2.8

- but you can provide an explicit ordering method—this works because operator "<" is implemented by the *Ordered* trait:

```
scala> List(1,3,2).sortWith(_ < _)
res3: List[Int] = List(1, 2, 3)
```

- you can mix in *Ordered[A]** with your own trait or class based on *A*, which defines the abstract method `def compare(that: A): Int`

```
case class UniformDouble(x: Double) extends AnyVal with Ordered[UniformDouble] {
    def + (y: Double) = x + y
    def compare(that: UniformDouble): Int = x.compare(that.x)
}

(scalaTest…)
val y = RNG.randoms(new UniformDoubleRNG(0L)) take 10 toList;
y.sorted.head should equal (UniformDouble(0.05298827162996736))
```

*** see: https://github.com/scala/scala/blob/v2.13.5/src/library/scala/math/Ordered.scala**

Ordering

- Since 2.8, Scala has used *Ordering* as the primary mechanism for sorting.
- *Ordering* is a typeclass whereas *Ordered* is a supertype.
  - There are implicit conversions between *Ordered* and *Ordering*, however.
  - For example:
    - `trait Numeric[T] extends Ordering[T]`

Getting IDE help with implicits

- https://confluence.jetbrains.com/display/IntelliJIDEA/Working+with+Scala+Implicit+Conversions
- https://blog.jetbrains.com/scala/2018/07/25/intellij-scala-plugin-2018-2-advanced-implicit-support-improved-patterns-autocompletion-semantic-highlighting-scalafmt-and-more/