Implicits

Part 2

Implicit Constraints and Conversions

Agenda

1. Beyond One Type Parameter
2. =:= and <:<
3. Safe Casting
4. Implicit Conversions
5. Debugging Implicits
6. Recommendations and Best Practices

Beyond One Type Parameter

- Remember that type parameters become a part of the overall type
- An implicit parameter simply matches and fills in a unique type
- Therefore, availability of an implicit for multiple types can enforce rules
- These rules are orthogonal to the Scala type system and may be used in combination
- Note that implicit parameter availability is enforced, but does not itself affect a type it applies to!

Sophisticated Eating Rules

- We have Foods and Eaters:

```
trait Food { def name: String }

case class Fruit(name: String) extends Food
case class Cereal(name: String) extends Food
case class Meat(name: String) extends Food

trait Eater { def name: String }

case class Vegan(name: String) extends Eater
case class Vegetarian(name: String) extends Eater
case class Paleo(name: String) extends Eater
```

- We want to ensure only correct foods may be fed to each of the eaters: two type parameters now, `EATER` and `FOOD`

```
import scala.annotation.implicitNotFound

@implicitNotFound(msg="Illegal Feeding: No Eats rule from ${EATER} to ${FOOD}")
trait Eats[EATER <: Eater, FOOD <: Food] {
    def feed(food: FOOD, eater: EATER) = s"${eater.name} eats ${food.name}"
}
```

Feed Following the Rules

- Let's write a restrictive `feedTo` method:

```
def feedTo[FOOD <: Food, EATER <: Eater](food: FOOD, eater: EATER)
  (implicit ev: Eats[EATER, FOOD]) = {
    ev.feed(food, eater)
}
```

- May only use this method for an eater and food for which evidence (`ev`) exists that `Eater` eats `Food`

- Some foods and eaters ready to test:

```
val apple = Fruit("Apple")
val alpen = Cereal("Alpen")
val beef = Meat("Beef")

val alice = Vegan("Alice")
val bob = Vegetarian("Bob")
val charlie = Paleo("Charlie")
```

Vegan Rules

- Can we feed an apple to alice?

```
feedTo(apple, alice)

Main.scala:29: Illegal Feeding: No Eats rule from Vegan to Fruit
feedTo(apple, alice)
      ^
```

- Not yet, let's make a rule:

```
object VeganRules {
    implicit object veganEatsFruit extends Eats[Vegan, Fruit]
}
```

- Remember to import the rules, and try again

```
import VeganRules._
feedTo(apple, alice)

import VeganRules._
res6_1: String = "Alice eats Apple"
```

Vegan Rules

- This is now compile-time enforced behavior
- Vegan still can't eat other foods like Meat or Cereal:

```
feedTo(alpen, alice)

Main.scala:31: Illegal Feeding: No Eats rule from Vegan to Cereal
feedTo(alpen, alice)
      ^
```

- Unless we define one of course
- Note that the implicit rules must be imported into scope (implicits must be implicit *in scope*)
- Note also that different implicit rules may be imported to change behavior!

Infix Type Notation

- Recap: Scala has infix notation for method calls with a single parameter:

```
1 + 2
// is equivalent to
1.+(2)
```

- Scala has similar syntactic sugar for types with two type parameters:

```
Eats[Vegan, Fruit]
// is equivalent to
Vegan Eats Fruit
```

- Sometimes you will need to put the infix form in parens to disambiguate for the compiler

- We could now write the method `feedTo` like this:

```
def feedTo[FOOD <: Food, EATER <: Eater](food: FOOD, eater: EATER)
  (implicit ev: EATER Eats FOOD) = {
    ev.feed(food, eater)
}
```

Vegetarian and Paleo Rules

- We can now use the infix to define rules as well, e.g. Vegetarian

```
object VegetarianRules {
    implicit object vegEatsFruit extends (Vegetarian Eats Fruit)
    implicit object vegEatsCereal extends (Vegetarian Eats Cereal)
}
```

- Parens needed here or compiler gets confused
- May also define `val`s instead of `object`s:

```
object PaleoRules {
    implicit val paleoEatsFruit = new (Paleo Eats Fruit) {}
    implicit val paleoEatsMeat = new (Paleo Eats Meat) {}
}
```

- Both work, but overall I think `object extends` looks cleaner

Core Library Type Classes and Constraints

- Scala provides several type classes and constraints in `Predef`, e.g. `Numeric`

```
val intNumT = implicitly[Numeric[Int]]
intNumT.times(5, 8)

res1_1: Int = 40

val doubleNumT = implicitly[Numeric[Double]]
doubleNumT.times(5.0, 8.0)

res3_1: Double = 40.0

val stringNumT = implicitly[Numeric[String]]

Main.scala:24: could not find implicit value for parameter e: Numeric[String]
implicitly[Numeric[String]]
          ^
```

- Numeric types only have a `Numeric` type class which provides arithmetic operations
- There are also implicit constraints like `=:=` and `<:<` which we will look at next

Method Imposed Type Constraints

- Let's say we create a simple `Cell` container to hold generic items:

```
case class Cell[T](item: T) {
    def *(other: Cell[T]): Cell[T] = this.item * other.item
}

Main.scala:24: value * is not a member of type parameter T
    def *(other: Cell[T]): Cell[T] = this.item * other.item
                                               ^
```

- T is generic, so we don't know there will be a * method
- Could make `Numeric` a context bound, but what if we want to still store other types and only allow multiply on `Numerics`?

=:= Type Equivalence

- Let's make a new type context bound by `Numeric` and ask Scala to prove that this new type is the same as T!

```
case class Cell[T](item: T) {
    def *[U: Numeric](other: Cell[U])(implicit ev: T =:= U): Cell[U] = {
      val numClass = implicitly[Numeric[U]]
      Cell(numClass.times(this.item, other.item))
    }
}
```

- We can now make Cells of any type, but only multiply Numerics:

```
val stringCell = Cell("hello")
val intCell = Cell(6)

intCell * Cell(7)
res7: Cell[Int] = Cell(42)

stringCell * Cell("there")
// could not find implicit value for evidence parameter of type Numeric[String]
// stringCell * Cell("there")
```

How It Works

Scala provides implicit proof that for all `T`, `T =:= T` is always true (in Predef)

```
@implicitNotFound(msg = "Cannot prove that ${From} =:= ${To}.")
sealed abstract class =:=[From, To] extends (From => To) with Serializable
private[this] final val singleton_=:= =
   new =:=[Any,Any] { def apply(x: Any): Any = x }
object =:= {
   implicit def tpEquals[A]: A =:= A = singleton_=:=.asInstanceOf[A =:= A]
}
```

- Note that `=:=` also extends `Function1[From, To]` which means that if `T =:= U` is available, `T` can be implicitly converted to `U`
- This makes

```
numClass.times(this.item, other.item)
```

work, even though `this.item` is `T` and `numClass` is expecting a `U`, the implicit converts automatically

- Spiffy

Safe Casting

- Let's consider this on a conceptual `Future`:

```
def flatten[F](implicit ev: T =:= Future[F]): Future[F] =
    FlattenFuture(this.asInstanceOf[Future[Future[F]]])
```

- Where `FlattenFuture` is able to collapse (flatten) the `Future[F]` type to `F` so instead of a `Future[Future[F]]` we will end up with a `Future[F]`
- We need to use a cast in this case, because the implicit is from `T => Future[F]` but we need to convert this instance to a `Future[Future[F]]` here
- This is known as a **safe** cast because the compiler has provided evidence that it will not fail

<:< Sub-Type Implicit Bounding

- One flaw with the above implementation is that it requires the type parameter to be provided in use

```
val flattened: Future[String] = futureFutureString.await[String]
```

```
val flattened: Future[String] = futureFutureString.await
```

will not work since the type parameter will be inferred as `Nothing` before the implicit evidence is found, and hence the implicit evidence will not be found!

- Using `<:<` fixes this problem:

```
def flatten[F](implicit ev: T <:< Future[F]): Future[F] =
    FlattenFuture(this.asInstanceOf[Future[Future[F]])
```

Variance in `<:<` allows the implicit to be found, then Scala infers the more appropriate type from there.

Implicit Conversions

```scala
case class Complex(real: Double, imaginary: Double = 0.0) {
    override def toString = s"$real $sign ${imaginary.abs}i"
    private def sign = if (imaginary >= 0.0) "+" else "-"

    def +(other: Complex) =
      Complex(real + other.real, imaginary + other.imaginary)
}
```

```scala
val c1 = Complex(5, 6)
val c2 = Complex(-3, -6)

c1 + c2
res2: Complex = 2.0 + 0.0i
```

- What about:

```scala
c1 + 6
Main.scala:25: type mismatch;
 found   : Int(6)
 required: Complex
c1 + 6
```

Overloading

```scala
case class Complex(real: Double, imaginary: Double = 0.0) {
  override def toString = s"$real $sign ${imaginary.abs}i"
  private def sign = if (imaginary >= 0.0) "+" else "-"
  def +(other: Complex) = Complex(real + other.real, imaginary + other.imaginary)
  def +(other: Int) = Complex(real + other, imaginary)
}
```

```scala
val c1 = Complex(5, 6)
c1 + 10
res6: Complex = 15.0 + 6.0i
```

```scala
10 + c1
Main.scala:25: overloaded method value + with alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int <and>
  (x: String)String
 cannot be applied to (Complex)
```

Implicit Conversions

```scala
case class Complex(real: Double, imaginary: Double = 0.0) {
    override def toString = s"$real $sign ${imaginary.abs}i"
    private def sign = if (imaginary >= 0.0) "+" else "-"
    def +(other: Complex) =
      Complex(real + other.real, imaginary + other.imaginary)
}
object Complex {
    implicit def intToComplex(i: Int): Complex = Complex(i)
}
```

```scala
val c1 = Complex(5, 6)
c1 + 10
res9_0: Complex = 15.0 + 6.0i
10 + c1
res9_1: Complex = 15.0 + 6.0i
Complex.intToComplex(10) + c1
res9_1: Complex = 15.0 + 6.0i
```

- Note, no longer need the overloaded + either
- When Scala has a type problem between two types, it looks for an implicit conversions to/from either type that will solve it

Extension Methods

```
class TimesInt(i: Int) {
    def times(fn: => Unit): Unit = {
        var x = 0
        while (x < i) {
            x += 1
            fn
        }
    }
}

implicit def intToTimesInt(i: Int): TimesInt = new TimesInt(i)
```

```
5 times { println("hello") }
```

```
intToTimesInt(5).times { println("hello") }
```

```
import scala.language.implicitConversions
```

```
def intToTimesInt(i: Int) = ???  // to disable an implicit in the same scope
```

Implicit Classes

- Convention: for implicit conversions, you should *own* one of the classes
- Implicit classes make this easy (and require no language import)

```
implicit class TimesInt(i: Int) {
    def times(fn: => Unit): Unit = {
        var x = 0
        while (x < i) {
            x += 1
            fn
        }
    }
}
```

- Effectively, Scala generates the implicit method for you automatically

```
implicit def TimesInt(i: Int): TimesInt = new TimesInt(i)
```

- For this reason, `implicit class` definitions must be inside other objects or classes

Implicit Value Classes

- Implicit extension classes are very useful and easy
- However they still require a *wrapping* at runtime into the target class
- Implicit value classes avoid this overhead:

```
implicit class TimesInt(val i: Int) extends AnyVal {
    def times(fn: => Unit): Unit = {
        var x = 0
        while (x < i) {
            x += 1
            fn
        }
    }
}
```

- Must have just **one public** parametric field and no other state, and extend `AnyVal`
- Methods (like `times`) are now generated as static methods, no runtime wrapping required

How to Debug Implicits

1. Don't overuse implicits and you won't need to debug them as much
2. Try applying the implicit method (or parameter) explicitly and see what the compiler says
3. Check for conflicting implicits (Scala won't choose - no ambiguity allowed)
4. `scalac -Xprint:typer` or `scalac -Ybrowse:typer` to see the desugaring
5. Also `desugar` in ammonite can help

Where to Put Implicits

- Just define it in the class you need it?
- A utility object you can import from, e.g. `import some.pkg.here.complex.ComplexNumberImplicits._`
- A package object, e.g.

```
package some.pkg.here

package object complex {
    implicit def intToComplex(i: Int): Complex = new Complex(i)
}

import some.pkg.here.complex.Complex  // will not bring it in
import some.pkg.here.complex._        // will bring it in
```

- The class companion object - always searched last, can't un-invite!

```
object Complex {
    implicit def intToComplex(i: Int): Complex = new Complex(i)
}
```

- Same rules/recommendations for `implicit class` definitions

Best Practices

- Never define implicits between two classes you don't own at least one of
- Only use companion object implicits when it is always desired/safe
- Don't over-use implicits, use them where they make sense
- Also don't avoid implicits, they are the first, safest tool for extending the Scala type system
- Smart use of implicits can often negate the need for macros (particularly type-classes)
- Know and apply the rules of implicits

Rules of Implicits (recap)

1. **Non-ambiguity**, Scala will not choose between applicable implicits
2. **One at a time**, implicits will not be automatically composed or chained
3. **Explicits First**, implicits will not be used if the code will compile without
4. **Implicits Only**, only parameters, classes or methods marked as `implicit` will be used
5. **Simple Naming**, implicits must be imported into namespace so that no `.` is required to reference them (exception is that companion object is also searched for implicits)
6. **Only the types matter**, the implicit should be named for code readers, but Scala only cares about the types