Scala Clinic
Intermediate Scala

Jim Powers

Patch.com

23 February 2012

Intermediate Scala

- Implicits

Intermediate Scala

- Implicits
  - Implicit Conversions

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds
- Views (collections)

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds
- Views (collections)
- View Bounds

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds
- Views (collections)
- View Bounds
- Context Bounds

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds
- Views (collections)
- View Bounds
- Context Bounds
- *Type Classes*

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds
- Views (collections)
- View Bounds
- Context Bounds
- *Type Classes*
- Manifests and Type Erasure

Intermediate Scala

- Implicits
  - Implicit Conversions
  - Implicit Arguments
- Type Bounds
- Views (collections)
- View Bounds
- Context Bounds
- *Type Classes*
- Manifests and Type Erasure
- Higher-Kinded Types

Implicits

Implicits are a way to tell the compiler of a way to prove (*provide evidence*) that there is a way to get from *here* (desired functionality) to *there* (actual functionality) with the compiler actually *connecting the dots* for you.

Detour: *Companion Objects*

Companion Objects are objects that share the same name as a *class* in the same compilation unit.

Companion Objects can be used to house implicit definitions to be used whenever the associated class is in scope.

Companion Objects

Companion objects can hold stuff for instances

Example:

```
object Test {
  class Foo(val value:Int)
  object Foo {
    implicit def fooToInt(f:Foo):Int = f.value
  }

  val v1:Foo = new Foo(10)
  val v2:Int = v1
}
```

Implicit Scope

Implicit Scope is the *search space* used by the compiler to resolve requests for *implicit evidence*.

Implicit Scope

- First look in current scope

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)
  - Same scope in other files (4)

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)
  - Same scope in other files (4)
- Parts of the type of implicit value being looked up and their companion objects

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)
  - Same scope in other files (4)
- Parts of the type of implicit value being looked up and their companion objects
  - Companion objects of the type

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)
  - Same scope in other files (4)
- Parts of the type of implicit value being looked up and their companion objects
  - Companion objects of the type
  - Companion objects of type arguments of types

Taken from Josh Suereth's NE Scala Symposium presentation:
`https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8`

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)
  - Same scope in other files (4)
- Parts of the type of implicit value being looked up and their companion objects
  - Companion objects of the type
  - Companion objects of type arguments of types
  - Outer objects for nested types

Taken from Josh Suereth's NE Scala Symposium presentation:
`https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8`

Implicit Scope

- First look in current scope
  - Implicits defined in current scope (1)
  - Explicit imports (2)
  - wildcard imports (3)
  - Same scope in other files (4)
- Parts of the type of implicit value being looked up and their companion objects
  - Companion objects of the type
  - Companion objects of type arguments of types
  - Outer objects for nested types
  - Other dimensions

Taken from Josh Suereth's NE Scala Symposium presentation:
https://docs.google.com/present/view?id=dfqn4jb_106hq4mvbd8

Implicit Conversions

Implicit Conversions provide a way to make *ad-hoc* conversions between values of some type. There are two primary use-cases for *implicit conversions*:

Implicit Conversions

Implicit Conversions provide a way to make *ad-hoc* conversions between values of some type. There are two primary use-cases for *implicit conversions*:

- To “pimp” or extend existing types with new functionality

Implicit Conversions

Implicit Conversions provide a way to make *ad-hoc* conversions between values of some type. There are two primary use-cases for *implicit conversions*:

- To “pimp” or extend existing types with new functionality
  - One of the ways to “lift” values into a *DSL*

Implicit Conversions

Implicit Conversions provide a way to make *ad-hoc* conversions between values of some type. There are two primary use-cases for *implicit conversions*:

- To "pimp" or extend existing types with new functionality
  - One of the ways to "lift" values into a *DSL*
- To remove conversion boilerplate when a value one type can unambiguously be used *as* value of another type.

Implicits

“Pimping”

Example:

```
object Test {
  implicit def secondsTo(x:Double) = new {
    def millis = x * 1000.0
    def tenths = x * 10.0
    def minutes = x/60.0
    def hours = x/3600.0
    def days = x/(3600.0*24.0)
  }

  def test:String = {
    val t = 12345.0
    "%f days, %f hours, %f minutes, %f seconds, %f tenths, %f ms".format(
      t.days,
      t.hours,
      t.minutes,
      t,
      t.tenths,
      t.millis)
  }
}

Test.test // ->> "0.142882 days, 3.429167 hours, 205.750000 minutes,
12345.000000 seconds, 123450.000000 tenths, 12345000.000000 ms"
```

Implicits

Converting

Example:

```
object Test {
  implicit def intToString(i:Int):String = i.toString
  def ish(s:String) = s+"-ish"
  def test = ish(5)
}
```

Implicit Arguments

Implicit Arguments enable the compiler to supply arguments based on type alone (although you can supply your own arguments as well). The feature effectively allows the compiler to infer *proofs*. The implicit argument mechanism enables the *typeclass pattern*.

Implicits

Implicit Arguments

Example:

```
object Test {
  class Greeter(name:String) {
    def greet(implicit prefix:String) =
      "%s, %s".format(prefix,name)
  }
  object Greeter {
    implicit val prefix = "Howdy!"
  }
  def run {
    import Greeter._
    val g = new Greeter("jim")
    println(g.greet)
  }
}
```

Type Bounds

Type Bounds provide a way to constrain the types supplied as type parameters.

Detour: Variance

Variance describes the acceptable *variations* of a type parameter when relating two values. The options are:

Detour: Variance

Variance describes the acceptable *variations* of a type parameter when relating two values. The options are:

- *Invariant* - type parameters must match *exactly*

Detour: Variance

Variance describes the acceptable *variations* of a type parameter when relating two values. The options are:

- *Invariant* - type parameters must match *exactly*
- *Covariant* - acceptable values can be of the same type or a *subtype*

Detour: Variance

Variance describes the acceptable *variations* of a type parameter when relating two values. The options are:

- *Invariant* - type parameters must match *exactly*
- *Covariant* - acceptable values can be of the same type or a *subtype*
- *Contravariant* - acceptable values can be of the same type or a *supertype*

Detour: Varaince

Invariance

Example:

```
object Test {
  class Foo[T](val x:T)

  val x:Foo[Object] = new Foo[Object](new Object)
  // Won't compile
  val y:Foo[Object] = new Foo[String]("Won't compile!")
//   error: type mismatch;
//   found    : Test.Foo[String]
//   required: Test.Foo[java.lang.Object]
// Note: String <: java.lang.Object, but class Foo is invariant in type T.
// You may wish to define T as +T instead. (SLS 4.5)
//                val y:Foo[Object] = new Foo[String]("Won't compile!")
}
```

Detour: Variance

Covaraince

Example:

```
object Test {
  class Foo[+T](val x:T)
  val x:Foo[Object] = new Foo[Object](new Object)
  // Compiles!
  val y:Foo[Object] = new Foo[String]("Compiles!")
  // Won't compile
  val z:Foo[String] = new Foo[Object]("Won't compile!")
}
```

Detour: Variance

Contravaraince

Example:

```
object Test {
  abstract class Foo[-T,+V] {
    def apply(x:T):V
  }

  val x = new Foo[String,Int] {
    def apply(x:String):Int = x.length
  }

  val y = new Foo[Object,Int] {
    def apply(x:Object):Int = -1
  }

  // Compiles!
  val z:Foo[String,Int] = y

  // Won't compile
  val w:Foo[Object,Int] = x
}
```

Type Bounds

Type Bounds are just ways to further constrain *variance*!

Type Bounds

Upper Bound

Example:

```scala
object Test {
  trait Baz {
    val tehBaz = "Whoa yeah!"
  }

  class Foo[T <: Baz] {
    def apply(x:T) = x.tehBaz
  }

  class Bad extends Baz
  class Goo extends Bad {
    override val tehBaz = "Hell's yeah!"
  }
  class Garg {
    val tehBaz = "Wha wha!"
  }

  // all good!
  val f1 = (new Foo[Bad])(new Bad)
  val f2 = (new Foo[Goo])(new Goo)
  // Won't compile
  val f3 = (new Foo[Garg])(new Garg)
}
```

Type Bounds

Lower Bound

Example:

```
object Test {
  trait Baz {
    val tehBaz = "Whoa yeah!"
  }

  class Bad extends Baz
  class Goo extends Bad {
    override val tehBaz = "Hell's yeah!"
  }
  class Foo[T >: Bad] {
    def apply(x:T) = x.toString
  }

  // all good!
  val f1 = (new Foo[Bad])(new Bad)
  // Won't compile
  val f2 = (new Foo[Goo])(new Goo)
}
```

Type Bounds

Upper and Lower Bound

Example:

```
object Test {
  trait Top {
    val tehBaz = "I'm the tops!"
  }
  trait Baz extends Top {
    override val tehBaz = "Whoa yeah!"
  }

  class Topsy extends Top
  class Bad extends Baz
  class Goo extends Bad {
    override val tehBaz = "Hell's yeah!"
  }
  class Foo[T >: Goo <: Baz] {
    def apply(x:T) = x.toString
  }

  // all good!
  val f1 = (new Foo[Bad])(new Bad)
  // Won't compile
  val f2 = (new Foo[Goo])(new Goo)
  // Won't compile
  val f3 = (new Foo[Topsy])(new Topsy)
}
```

Views (collections)

Views on collections is a way to *fuse* transformer operations such as `map, flatMap, filter, etc.` into a single operation and apply that operation once to a collection avoiding the creation of intermediate collections. Intelligent use of collection views can significantly improve performance while not losing the expressiveness of Scala's collections library.

Views (collections)

Bad pattern

Example:

```
object Test {
  val v = Vector(1,2,3,4,5,6,7,8,9,10)
  def sillyAdd(x1:Int,x2:Int) =
    v map (_ + x1) map (_ + x2)
  // The above creates 2 copies
}
```

Views (collections)

Better pattern

Example:

```
object Test {
  val v = Vector(1,2,3,4,5,6,7,8,9,10)
  def betterSillyAdd(x1:Int,x2:Int) =
    (v.view map (_ + x1) map (_ + x2)).force
  // The above creates 1 copy
}
```

View Bounds (or just plain Views)

View Bounds are a way to express to the compiler a requirement that a type T be treated as a type S in some scope. This requirement can be fulfilled by the compiler if an *implicit conversion* can be found. The type of the implicit conversion has to be $T \Rightarrow S$.

View Bounds

Sugared

Example:

```
object Test {
  class Height[S](v:S,f:S => Int) {
    def height:Int = f(v)
  }
  implicit def stringToHeight(s:String):Height[String] =
    new Height(s,_.length)
  def sillyLength[A <% Height[A]](x:A) =
    x.height
}
```

View Bounds

De-sugared

Example:

```
object Test {
  class Height[S](v:S,f:S => Int) {
    def height:Int = f(v)
  }
  implicit def stringToHeight(s:String):Height[String] =
    new Height(s,_.length)
  def sillyLength[A](x:A)(implicit f:A => Height[A]) =
    f(x).height
}
```

Context Bounds

Context Bounds is a way to tell the compiler to look for a *correspondence* to a given type in implicit scope and make that correspondence implicit in a newly introduced scope.

Context Bounds: More technical explanation

Context Bounds defines a constraint on a type T that says that a corresponding value of type `C`$[T]$ must be visible in implicit scope to successfully type-check. If such a correspondence exists then introduce the corresponding value of type `C`$[T]$ into a newly defined scope.

Context Bounds

Sugared

Example:

```
object Test {
  implicit val inTail:List[Int] =
    List(-1,-2,-3)
  implicit val stringTail:List[String] =
    List("A","B","C")
  def sillyLists[T](x:T)(implicit t:List[T]) =
    x::t
  def makeSillyLists[T : List](x:T) =
    sillyLists(x)
  val v1 = makeSillyLists(1)
  val v2 = makeSillyLists("Better")
  // Won't compile
  val v3 = makeSillyLists(1.0)
// error: could not find implicit value for evidence parameter of type List[Double]
//              val v3 = makeSillyLists(1.0)
}
```

Context Bounds

De-sugared

Example:

```
object Test {
  implicit val inTail:List[Int] =
    List(-1,-2,-3)
  implicit val stringTail:List[String] =
    List("A","B","C")
  def sillyLists[T](x:T)(implicit t:List[T]) =
    x::t
  def makeSillyLists[T](x:T)(implicit t:List[T]) =
    sillyLists(x)
  val v1 = makeSillyLists(1)(inTail)
  val v2 = makeSillyLists("Better")(stringTail)
}
```

Context Bounds

Defining “Zeros”

Example:

```
object Test {
  class Zero[T](val value:T)

  def zero[T](implicit z:Zero[T]):T = z.value

  implicit def intZero:Zero[Int] = new Zero(0)
  implicit def stringZero:Zero[String] = new Zero("")

  def accept[T : Zero](v:T,f:T => Boolean):T =
    if (f(v)) v else zero

  def stringTest(s:String):Boolean = s.length < 3
  def intTest(i:Int):Boolean =  (i % 5) == 0

  // ->   List("", 12, "")
  val v1 = List("12345","12","123").map(x => accept(x,stringTest _))
  // -> Vector(0, 0, 0, 0, 5, 0, 0, 0, 0, 10)
  val v2 = (1 to 10).map(x => accept(x,intTest _))
}
```

Type Classes

The term *Type Classes* comes from research into *typed lambda calculus* and has been made popular by the programming language *Haskell*. Even though the word "classes" appears in the term, it has *nothing, whatsoever, to do with OO or Scala classes*. *Type Classes* are a way to address the need for *ad-hoc polymorphism* in a language.

Type Classes

Type Classes are a considerably more powerful way to construct software systems than by using structural inheritance.

Type Classes

Appending stuff

Example: *Lots of duplication*

```
object Test {
  def appendList[T](x:T,xs:List[T]):List[T] = xs :+ x
  def appendVector[T](x:T,xs:Vector[T]):Vector[T] = xs :+ x
  // Repeat for all things you are interested in ...

  // Can't we just do:
  def append[T,M[_]](x:T,xs:M[T]):M[T] = xs :+ x
  // Won't compile :-(
}
```

Type Classes

But I just want a generic append function! After some research you discover the notion of a *semigroup*.

Type Classes

Semigroup

Example:

```
abstract class Semigroup[T,M[_]] {
  def append(x:M[T],xs:M[T]):M[T]
}
```

Type Classes

Semigroups are close to what we want, but we need to get T into $M[T]$ in some generic fashion. More research turns up the *monoid*.

Type Classes

Monoid

Example:

```
abstract class Monoid[T,M[_]] {
  def identity(x:T):M[T]
  def append(x:M[T],xs:M[T]):M[T]
}
```

Type Classes

Monoid Typeclass

Example:

```
object Test {
  abstract class Monoid[T,M[_]] {
    def identity(x:T):M[T]
    def append(x:M[T],xs:M[T]):M[T]
  }

  implicit def listMonoid[T]:Monoid[T,List] = new Monoid[T,List] {
    def identity(x:T):List[T] = List(x)
    def append(x:List[T],xs:List[T]):List[T] = xs ++ x
  }
  implicit def vectorMonoid[T]:Monoid[T,Vector] = new Monoid[T,Vector] {
    def identity(x:T):Vector[T] = Vector(x)
    def append(x:Vector[T],xs:Vector[T]):Vector[T] = xs ++ x
  }

  def append[T,M[_]](x:T,xs:M[T])(implicit m:Monoid[T,M]):M[T] =
    m.append(m.identity(x),xs)

  val v1:List[Int] = append(1,List(-1,-2,-3))
  val v2:Vector[Int] = append(1,Vector(-1,-2,-3))
}
```

Type Classes

Generalized Monoids

Example:

```
object Test {
  abstract class Monoid[T,M[_]] {
    def identity(x:T):M[T]
    def append(x:M[T],xs:M[T]):M[T]
  }
  case class Id[T](value:T)
  implicit def toId[T](x:T):Id[T] = Id(x)
  implicit def fromId[T](x:Id[T]):T = x.value

  implicit def intPlusMonoid:Monoid[Int,Id] = new Monoid[Int,Id] {
    def identity(x:Int):Id[Int] = Id(x)
    def append(x:Id[Int],xs:Id[Int]):Id[Int] = Id(xs.value + x.value)
  }

  def append[T,M[_]](x:T,xs:M[T])(implicit m:Monoid[T,M]):M[T] =
    m.append(m.identity(x),xs)

  val v1:Int = append[Int,Id](1,2)
}
```

Generalized Monoids

Example: *Choose your monoid*

```
object Test {
  abstract class Monoid[T,M[_]] {
    def identity(x:T):M[T]
    def append(x:M[T],xs:M[T]):M[T]
  }
  case class Id[T](value:T)
  implicit def toId[T](x:T):Id[T] = Id(x)
  implicit def fromId[T](x:Id[T]):T = x.value

  implicit def intPlusMonoid:Monoid[Int,Id] = new Monoid[Int,Id] {
    def identity(x:Int):Id[Int] = Id(x)
    def append(x:Id[Int],xs:Id[Int]):Id[Int] = Id(xs.value + x.value)
  }

  def intMultMonoid:Monoid[Int,Id] = new Monoid[Int,Id] {
    def identity(x:Int):Id[Int] = Id(x)
    def append(x:Id[Int],xs:Id[Int]):Id[Int] = Id(xs.value * x.value)
  }

  def append[T,M[_]](x:T,xs:M[T])(implicit m:Monoid[T,M]):M[T] =
    m.append(m.identity(x),xs)

  // Uses plus
  val v1:Int = append[Int,Id](1,2)
  // Uses mult
  val v2:Int = append[Int,Id](1,2)(intMultMonoid)
```

Manifests and Type Erasure

Sadly, not all type information available at compile time is not available at run time. In particular, type parameter information is lost at run time.

Manifests and Type Erasure

Type information lost

Example:

```
object Test {
  def whatDoIHave(l:List[_]):String = l match {
    case _:List[Int] => "List of int"
    case _:List[Double] => "List of double"
    case _:List[String] => "List of String"
    case _:List[Byte] => "List of byte"
    case _ => "I don't know"
  }

  val v1 = whatDoIHave(List(1,2,3)) // => List of int
  val v2 = whatDoIHave(List(1.0,2.0,3.0)) // => List of int :-(
}
```

Manifests and Type Erasure

Recovering type information

Example:

```
object Test {
  class TypeMatch[T](implicit m:Manifest[T]) {
    def unapply(x:Manifest[_]):Option[Boolean] =
      if (x <:< m) Some(true)
      else None
  }
  def typeMatch[T : Manifest]:TypeMatch[T] = new TypeMatch[T]
  val listInt = typeMatch[List[Int]]
  val listDouble = typeMatch[List[Double]]
  val listString = typeMatch[List[String]]
  val listByte = typeMatch[List[Byte]]
  def whatDoIHave[T](l:List[T])(implicit m:Manifest[List[T]]):String = m
match {
    case listInt(_) => "List of int"
    case listDouble(_) => "List of double"
    case listString(_) => "List of String"
    case listByte(_) => "List of byte"
    case _ => "I don't know"
  }
  val v1 = whatDoIHave(List(1,2,3)) // => List of int
  val v2 = whatDoIHave(List(1.0,2.0,3.0)) // => List of int :-(
}
```

Higher-Kinded Types

Types classify values, *kinds* classify types.

Higher-Kinded Types

Kind: $*$

- Int, Byte, String, ...

Higher-Kinded Types

Kind: $*$

- Int, Byte, String, ...
- Classes with no type parameters

Higher-Kinded Types

Kind: $*$

- Int, Byte, String, ...
- Classes with no type parameters
- List[Int], Stream[String], ...

Higher-Kinded Types

Kind: $*$

- Int, Byte, String, ...
- Classes with no type parameters
- List[Int], Stream[String], ...
- Types with parameters where all parameters are filled in

Higher-Kinded Types

Kind: $* \rightarrow *$

- List,Stream,Vector,set

Higher-Kinded Types

Kind: $* \to *$

- List,Stream,Vector,set
- Any type with a single type-parameter argument

Higher-Kinded Types

Kind: $* \to * \to *$

- Function1, Map, PartialFunction

Higher-Kinded Types

Kind: $* \to * \to *$

- Function1, Map, PartialFunction
- Any type with a two type-parameters