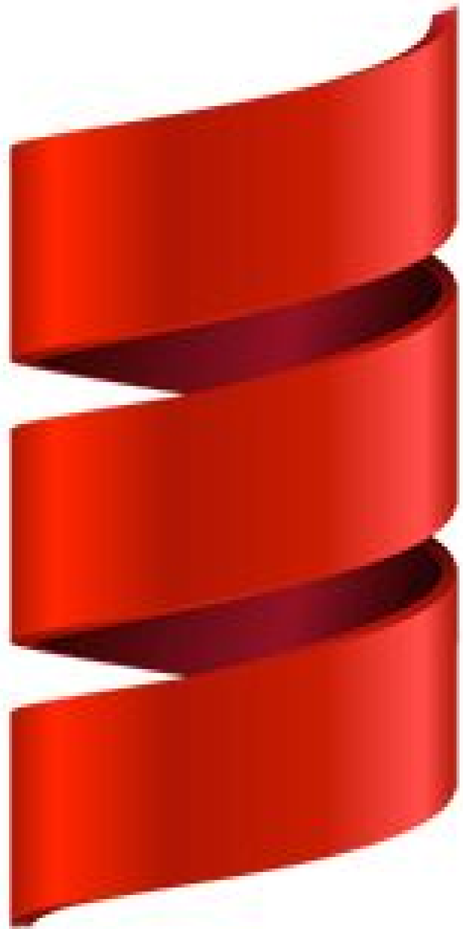


Scala

Implicits and YOU

Agenda

- When are implicits invoked ?
- Where are implicits looked for, where are they available from?
- Why, how and when should I use them?
- Type Classes

Q:When does the compiler look for an implicit value?

- Whenever there is a type error, before giving up, the compiler will try to resolve the issue by finding an implicit conversion or value

- When a parameter is marked implicit, and is not explicitly passed

- When an implicit is explicitly requested

Note about these examples:

- In this section, I outline the rules governing when the compiler looks for implicit values, and where they are found. But these simple examples are extremely bad practice.

- Only the later sections of this presentation should be considered for examples of real-world usage

Compile Error: Wrong Type

```
class Person(name: String)

implicit def stringToPerson(name: String) = new Person(name)

val p: Person = "Jacob"
```

-> String isn't a Person

-> Type Error

Before Giving up:

-> Look for implicit String => Person

-> replace:

val p: Person = stringToPerson("Jacob")

Compile Error: Wrong type 2

```
val keyValPair = ("Alice", 3)

implicit def keyValPairToMap[K,V]( kvp: (K,V) ): Map[K,V] = Map(kvp._1 -> kvp._2 )

val map: Map[String, Int] = keyValPair
```

Compiler Error: Non-existent member

```
class Person(name: String) {
  def sayHello() = println(s"Hello, I'm $name")
}

implicit def stringToPerson(name: String) = new Person(name)

val string = "Jacob"

string.sayHello()
```

-> sayHello() not a method of String

-> Type Error

Before giving up:

-> Look for implicit conversion to something that does

-> finds implicit stringToPerson: String => Person

replace:

```
stringToPerson(string).sayHello()
```

Implicit Parameters

```
implicit val d = 1.23456

def printADouble(implicit double: Double) = println(double)


printADouble(0.0) //0.0
printADouble // GO FIND AN IMPLICIT DOUBLE -> 1.23456
```

Called method taking an implicit parameter, without overriding with explicit parameter

Implicit Parameter Set

```
implicit val i = 2
implicit val s = "cool"

def printIntAndString( implicit i: Int, s: String) = println(i + s)

printIntAndString
printIntAndString(4, "school")
```

Only Entire parameter sets can be marked implicit

i, s -> Both implicit

If you want additional parameters, curry:

```
def printIntAndString(spacer: String)( implicit i: Int, s: String) =
println(i + spacer +  s)
```

Explicitly asking for an implicit

```
implicit val i = 2
implicit val s = "cool"

val a = implicitly[Int] // 2
```

Q: Where are implicits available from

Remember these rules:

- Marking Rule: definitions must be marked implicit
- Scope Rule: definitions must be in scope (1 exception)
- Single Identifier Rule
- One-at-a-time Rule: Implicits are NEVER chained/nested
- Explicits-First Rule: Explicits ALWAYS override implicits

Marking Rule: Easy peasy

```scala
// ERROR
def intToString(i: Int) = i.toString
val s: String = 4

// OK
implicit def intToString(i: Int) = i.toString
val s: String = 4
```

Scope Rule: Value must be in Scope

```
//// ERROR
object One {
  implicit val i = 1
}
val theInt = implicitly[Int]

//// OK
object One {
  implicit val i = 1
  val theInt = implicitly[Int]
}

// Also OK
object One {
  implicit val i = 1
  object Inner {
    val theInt = implicitly[Int]
  }
}
```

```
//Be Careful! Ambiguous implicit value error!
implicit val uhOh = 5
object One {
  implicit val i = 1
  object Inner {
    val theInt = implicitly[Int]
  }
}
```

Single Identifier Rule

```
// ERROR One.i is not available, that's two identifiers
implicit object One {
  val i = 1
}
val theInt = implicitly[Int]
```

```
// OK
trait Num {
  def i: Int
}
implicit object One extends Num {
  override val i = 1
}
val theInt = implicitly[Num].i
```

```
// OK
object One {
  implicit val i = 1
}

import One._
val theInt = implicitly[Int]
```

Exception to Single Identifier: Compiler will look in target type's companion object

```
case class User( id: Long, name: String)

object User {
  implicit def toPerson(user: User) = new Person(user.name)
}

class Person(name: String)

val person: Person = User(1, "Jacob")
```

Note: Could have put the conversion in object Person

But does not look in the class! #FAIL

```
case class User( id: Long, name: String) {
  implicit def toPerson(user: User) = new Person(user.name)
}

class Person(name: String)
val person: Person = User(1, "Jacob") // ERROR
```

One-at-a-time Rule: Implicits are never chained

```
implicit def stringToInt(s: String) = s.length
implicit def intToBoolean(i: Int) =  i != 0

val int: Int = "Hello" // 5
val boolean: Boolean = 4 // true

val doesNotCompile: Boolean = ":-("
```

Compiler will not attempt to perform:

val doesNotCompile = intToBoolean (stringToInt(“:-(“))

Why?

- Too confusing
- Would cause huge compiler performance issues

Explicits-First Rule: If code type-checks, no implicits are used

```
implicit val int = 4
val explicitInt: Int = 5


def printAnInt(implicit i: Int) = println(i)

printAnInt // 4
printAnInt(explicitInt) // 5
```

```
implicit val int = 1

val implicitlyFoundInt = implicitly[Int] // 1

val theInt = implicitly[Int](4) // 4
```

Q: Why, how, and when should I use them?

“There’s a fundamental difference between your own code and libraries of other people: you can change or extend your own code as you wish, but if you want to use someone else’s libraries, you usually have to take them as they are. A number of constructs have sprung up in programming languages to alleviate this problem. Ruby has modules, and Smalltalk lets packages add to each other’s classes. These are very powerful, but also dangerous, in that you modify the behavior of a class for an entire application, some parts of which you might not know… Scala’s answer is implicit conversions and parameters. These can make existing libraries much more pleasant to deal with by letting you leave out tedious, obvious details that obscure the interesting parts of your code. Used tastefully, this results in code that is focused on the interesting, non-trivial parts of your program.”

- Martin Odersky

Q: Why, how, and when should I use them?

Part A: To provide conversions between your code and somebody else's

Example

```
case class Complex(real: Int, imaginary: Int = 0) {
  def +(that: Complex) = copy(real + that.real, imaginary + that.imaginary)
  def -(that: Complex) = copy(real - that.real, imaginary - that.imaginary)
  def unary_- = copy(-real, -imaginary)
}
```

We create our Complex numbers class, so now let's try to use this together with the Standard Library's Int

Example

```
// Whoopsies! We never defined a +(that: Int) on our class. Error!
Complex(1, 5) + 2



// Let's add that in...

case class Complex(real: Int, imaginary: Int = 0) {
  def +(that: Complex) = copy(real + that.real, imaginary + that.imaginary)
  def -(that: Complex) = copy(real - that.real, imaginary - that.imaginary)

  def +(that: Int) = copy(real = real + that)
  def -(that: Int) = copy(real = real - that)

  def unary_+ = this
  def unary_- = copy(-real, -imaginary)
}

Complex(1, 5) + 2 // Complex(3, 5)
```

Example

```
// But without implicits, we'll never get to this:
40 - Complex(2,4)

// We would have to either:

//        fork the standard library to add the method Int.-(Complex): Complex etc

//        extend Int with Complex operations

//        Wrap Int in an explicit adapter class: :-( yuck! …………….
```

Example

```
case class IntOps(i: Int) {
  def toComplex = Complex(i)
}

Complex(1) + IntOps(5).toComplex
IntOps(5).toComplex + Complex(1,-344)
```

Example: Instead, use implicit conversions!

```
case class Complex(real: Int, imaginary: Int = 0) {
  def +(that: Complex) = copy(real + that.real, imaginary + that.imaginary)
  def -(that: Complex) = copy(real - that.real, imaginary - that.imaginary)

  def unary_+ = this
  def unary_- = copy(-real, -imaginary)
}

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


//now we can use Complexes/Ints with each other
Complex(1) + 5
5 + Complex(3, 15)
- Complex(1) - 3
```

Another approach, use implicit class!

```
case class Complex(real: Int, imaginary: Int = 0) {
  def +(that: Complex) = copy(real + that.real, imaginary + that.imaginary)
  def -(that: Complex) = copy(real - that.real, imaginary - that.imaginary)

  def unary_+ = this
  def unary_- = copy(-real, -imaginary)
}

implicit class IntOps(i: Int) {
  def toComplex = Complex(i)
}


Complex(1) + 5.toComplex
5.toComplex + Complex(1,-344)
```

Q: Why, how, and when should I use them?

Part B: To simulate new Syntax

Example: Implementing a Tree with cleaner Syntax

```
object MyTreeModule {
  case class Tree[T](value: T, children: Seq[Tree[T]] = Seq())
}
```

Iteration 1 / 4

Plain Old Scala Case Classes

```
import MyTreeModule._

val myTree = Tree(1, Seq(
  Tree(2),
  Tree(3, Seq(
    Tree(5))),
  Tree(4)))

val x: Tree[Int] = Tree(5)
```

```
Tree("Vancouver", Seq(
    Tree("Burnaby"),
    Tree("North Shore", Seq (
        Tree("West Vancouver"),
        Tree("North Vancouver")
        )),
    Tree("Vancouver Proper", Seq(
        Tree("Downtown", Seq(Tree("West End", Seq(Tree("Jervis Street"))))),
        Tree("West Side")
        ))
 )
```

Example: Implementing a Tree with cleaner Syntax

```
object MyTreeModule {
  case class Tree[T](value: T, children: Seq[Tree[T]] = Seq()) {
    def ~< (children: Seq[Tree[T]]) = Tree(value, children = children)
  }
}
```

Iteration 2 / 4

Use ~< (kinda looks like a tree)

```
import MyTreeModule._

val myTree = Tree(1) ~< Seq(
  Tree(2),
  Tree(3) ~< Seq(
    Tree(5)),
  Tree(4))
)

val x: Tree[Int] = Tree(5)
```

```
Tree("Vancouver") ~< Seq(
     Tree("Burnaby"),
     Tree("North Shore") ~< Seq (
          Tree("West Vancouver"),
          Tree("North Vancouver")
           ),
     Tree("Vancouver Proper") ~< Seq(
          Tree("Downtown") ~< Seq(Tree("West End") ~< Seq(Tree("Jervis Street"))),
          Tree("West Side")
           )
  )
```

Example: Implementing a Tree with cleaner Syntax

```
object MyTreeModule {
  case class Tree[T](value: T, children: Seq[Tree[T]] = Seq()) {
    def ~< (children: Seq[Tree[T]]) = Tree(value, children = children)
  }
 implicit def toTree[T](value: T): Tree[T] = Tree(value)

}
```

Iteration 3 / 4

Use implicit conversions from T => Tree[T]

```
import MyTreeModule._

val myTree3 =
  1 ~< Seq(
          2,
          3 ~< Seq(
                  4
                )
        )
```

```
val x: Tree[Int] = 5

"Vancouver" ~< Seq(
      "Burnaby",
      "North Shore" ~< Seq (
            "West Vancouver",
            "North Vancouver"
             ),
      "Vancouver Proper" ~< Seq(
            "Downtown" ~< Seq("West End" ~< Seq("Jervis Street")),
            "West Side"
             )
)
```

Example: Implementing a Tree with cleaner Syntax

```
object MyTreeModule {
  case class Tree[T](value: T, children: Seq[Tree[T]] = Seq()) {
    def ~< (children: Tree[T]* ) = Tree(value, children = children)
  }
 implicit def toTree[T](value: T): Tree[T] = Tree(value)

}
```

Iteration 4 / 4

Take in arbitrary Tree params, not a Seq[Tree [T]], no more Seq's

```
import MyTreeModule._

val myTree3 =
  1 ~< (
          2,
          3 ~<   4
        )
```

```
val x: Tree[Int] = 5

"Vancouver" ~<(
     "Burnaby",
     "North Shore" ~<(
          "West Vancouver",
          "North Vancouver"
           ),
     "City of Vancouver" ~<(
          "Downtown" ~< “West End” ~< “Jervis Street”,
          "West Side"
           )
)
```

Quick ideas.. might be good, might not be

Complex Numbers:

```
object Complex {
  implicit def intToComplex(i: Int) = Complex(i)
  implicit def doubleToComplex(d: Double) = Complex(d.toInt)
  val i = Complex(0,1)
  implicit class IntOps(int: Int) {
    def toComplex = Complex(int)
    def i = Complex(0,int)
  }
}

import Complex._

i
3 i

5 + 8.i
(3 - i) + (1 + 4.i)
```

Quick ideas.. might be good, might not be

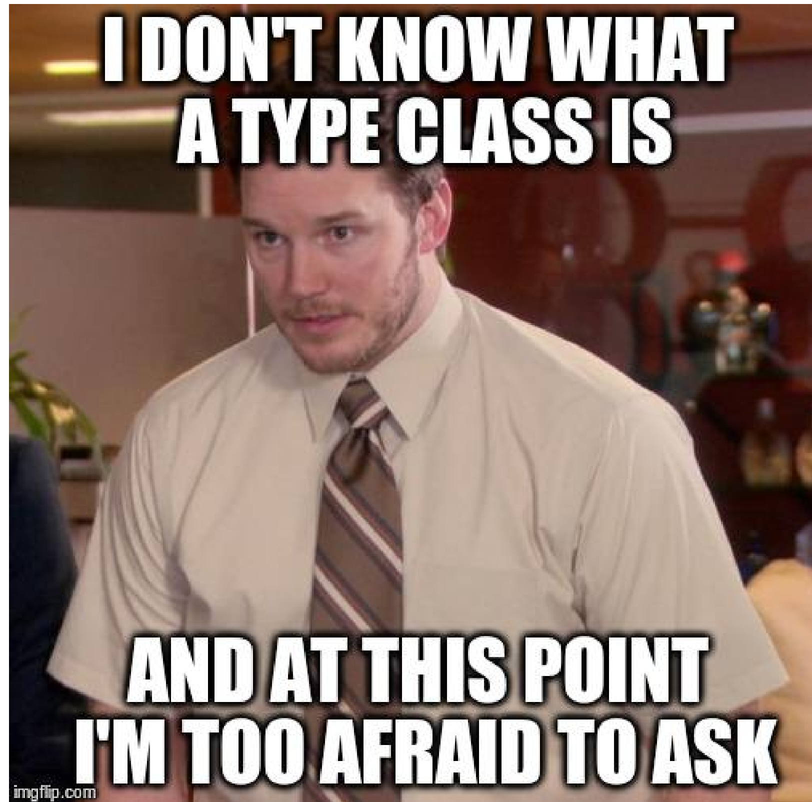
Time:

```
object TimeLibrary {
  //.... Not implemented..
}
import TimeLibrary._

val twentyThreeHours: TimeDuration = 1.d - 1.h
val lunchHourToday: TimeInterval = (today at noon) to (today at 13.oClock)
val twiceAMinute: Frequency = 2.times / 1.m
```

Type Classes

Type Classes



Q: Wtf is a Type Class

Quick Answer: Ad hoc polymorphism

Long Answer: Polymorphism over types, when you can’t use a trait, because:

- it isn’t your code

-or-

- you want to impose your domain-specific polymorphism to it without jamming your silly `extends LikesToEatIceCream` all over the code base

Q: Wtf is a Type Class

-or-

- There are multiple ways in which a type can achieve the polymorphism

Eg: There’s more than one way in which String’s can be summed

There’s more than one way to define a Vector Space on Doubles

etc..

Example: It isn’t your code

Obviously been here before with complex numbers, but now we want to provide an interface and abstract over Ints and Complexes (and Doubles,Longs,Shorts, Floats…)

Example: It isn't your code: Numbery things

```scala
object Math {
  trait Numbery[A] {
    def plus(x: A, y: A): A
    def zero: A
  }
  implicit object NumberyComplex extends Numbery[Complex] {
    override def plus(x: Complex, y: Complex) = x + y
    override def zero = 0.toComplex
  }
  implicit object NumberyInt extends Numbery[Int] {
    override def plus(x: Int, y: Int) = x + y
    override def zero = 0
  }
  implicit object NumberyDouble extends Numbery[Double] {
    override def plus(x: Double, y: Double) = x + y
    override def zero = 0.0
  }
}
```

Numbery[A] is the Type Class

Looking familiar Spray Json users??

Abstracting over Ints,Complexes, Doubles by default

this is 100% extensible though! Not limited to the three provided here!

Example: It isn't your code: Numbery things

```
object Math {
  trait Numbery[A] {
    def plus(x: A, y: A): A
    def zero: A
  }
  implicit object NumberyComplex extends Numbery[Complex] {
    override def plus(x: Complex, y: Complex) = x + y
    override def zero = 0.toComplex
  }
  implicit object NumberyInt extends Numbery[Int] {
    override def plus(x: Int, y: Int) = x + y
    override def zero = 0
  }
  implicit object NumberyDouble extends Numbery[Double] {
    override def plus(x: Double, y: Double) = x + y
    override def zero = 0.0
  }
}
```

Use it to implement a sum function over numbery things:

```
import Math._
object MyApplication {
  def sum[T](seq: Seq[T])(implicit n: Numbery[T]): T =
                                   seq.foldLeft(n.zero)(n.plus(_,_))

  val intSum = sum(List(1,2,3))
  val doubleSum = sum(Vector(1.0, 2.222, 3.1233))
  val complexSum = sum( i, 3, 3 + 5.i)

}
```

Requirements Changed! Strings gotta be Numbery!

Not a problem, add it with the others or just make an implicit object in your own code..

```
import Math._
object MyApplication {

  implicit object NumberyString extends Numbery[String] {
    override def sum(x: String, y: String) = s"$x$y"
    override def zero = ""
  }


  def sum[T](seq: Seq[T])(implicit n: Numbery[T]): T =
                    seq.foldLeft(n.zero)(n.plus(_,_))

  val stringSum = sum(List(“abcd”, “efg”, “hijk”, “...”))
}
```

Numeric

I implemented Numbery for fun, but the standard library comes with its own `Numeric` Type Class, you just have to implement these things:

1. **abstract def compare(x: T, y: T): Int**
2. Returns an integer whose sign communicates how x compares to y.
3. **abstract def fromInt(x: Int): T**
4. **abstract def minus(x: T, y: T): T**
5. **abstract def negate(x: T): T**
6. **abstract def plus(x: T, y: T): T**
7. **abstract def times(x: T, y: T): T**
8.
9. **abstract def toDouble(x: T): Double**
10. **abstract def toFloat(x: T): Float**
11. **abstract def toInt(x: T): Int**
12. **abstract def toLong(x: T): Long**

Example: Multiple Polymorphic implementations

```
object Math {
  trait Numbery[A] {
    def plus(x: A, y: A): A
    def zero: A
  }
}
```

```
import Math._
object MyApplication {
  def sum[T](seq: Seq[T])(implicit n: Numbery[T]): T =
                                                  seq.foldLeft(n.zero)(n.plus(_,_))


  object NumberyStrings extends Numbery[String] {
    override def plus(x: String, y: String) = x + y
    override def zero = “”
  }
  object NumberyStringsWithSpaces extends Numbery[String] {
    override def plus(x: Int, y: Int) = x +” “+ y
    override def zero = “”
  }
  implicit val defaultNumberyStrings = NumberyStringsWithSpaces
}
```

That’s it