




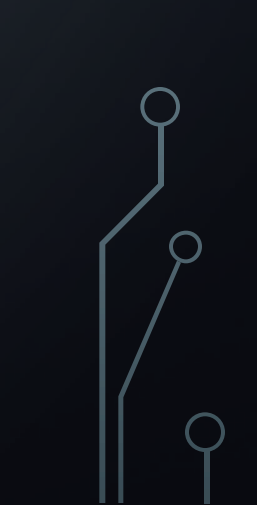
# ADVANCED SCALA: IMPLICIT

DETANGLING THREE OFTEN-CONFUSING FEATURES AND LEARNING HOW TO  
SUPERCHARGE YOUR CODE.

Murray Todd Williams  
© 2020 All rights reserved



Murray's guide on the best way to learn something:

- Read about it
  - Try to use it everywhere (over-use it)
  - Throw lots of code away and understand when it's really appropriate
  - Don't get depressed: good learning is always messy
  - Try teaching it to someone else
- 
- 

# IMPLICIT: THE GOOD AND THE BAD

## GOOD:

- Brief, succinct code. Let the compiler write your boilerplate when possible.
- Powerful ability to design extensible functionality
- Includes a sort of Dependency Injection
- Enables things like Scalaz & Cats (advanced Scala libraries that wizards use)

## BAD:

- “implicit” can refer to 3 different things, so it gets confusing.
- The easiest one to understand (implicit conversions) can cause the most problems.
- Scala 3 (Dotty) changes this by renaming and redesigning.



# THE THREE IMPLICIT

1. Implicit Conversion
  2. Implicit Classes
  3. Implicit Parameters
- 
- 
- 

# IMPLICIT CONVERSION (USE WITH CARE!!!)

**Problem:** you have class A when you really need class B (and you'd like the compiler to do some magic for you)

```
scala> case class Color(name: String)
scala> val c: Color = "Red"
error: type mismatch;
found   : String("Red")
required: Color
```

**Solution:** Define a function that converts type A to type B (i.e. that converts what you have into what you need) and declare it as *implicit*.



```
implicit def colorFromString(name: String): Color = name.toLowerCase match {  
  case "red" => Color("Red")  
  case "blue" => Color("Blue")  
  case "green" => Color("Green")  
  case _ => Color("Unknown")  
}
```

```
scala> val c: Color = "Red"  
val c: Color = Color(Red)
```



## Notes:

This is a horribly contrived example. A more typical thing would be writing a function from a tuple `(Double,Double)` to a `Geo(latitude,longitude)`



## Warning!!!

This seems cool and magical (it did to me when I first saw it) but it can be easily abused, and often you actually don't want the compiler making unexpected class conversions. (It's hard to debug, especially if you have lots of these defined all over.)

Dotty (Scala 3) will require that you import `scala.language.implicitConversions` into any scope you want this to work, just to make it harder to allow these bugs.



# IMPLICIT CLASSES (CLASS EXTENSIONS)

**Problem:** you need to add some functionality to a pre-existing class. (One that you don't have access to.)

Example: if I have two lists of doubles, I want to be able to do a pair-wise addition using just the `+` sign.

```
scala> List(1.0, 1.2) + List(3.2, 3.2)
error: type mismatch;
 found   : List[Double]
 required: String
```

**Old-fashioned solution:** create a subclass with the new functionality and just use that. This leads to inheritance tree nightmares and ultimately messy code.

**Scala solution:** create an Implicit Class to “wrap” the original object and extend the functionality!

```
implicit class Ops(list: List[Double]) {  
  def +(other: List[Double]): List[Double] = {  
    (list zip other).map(t => t._1 + t._2)  
  }  
}  
  
scala> List(1.0, 2.0) + List(3.0, 4.0)  
val res6: List[Double] = List(4.0, 6.0)
```

## Notes:

1. This is super-useful because it allows you to "bolt-on" new functionality to classes as needed.
2. Implicit classes cannot be top-level objects in a Scala source file; you must put them inside a class or object. (Often this goes in companion objects.)
3. To make this less confusing (more straightforward) Dotty (Scala 3) has replaced this with "extension methods".



# A QUICK DIVERSION: CURRYING

SO YOU DON'T GET CONFUSED IN THE NEXT SECTION





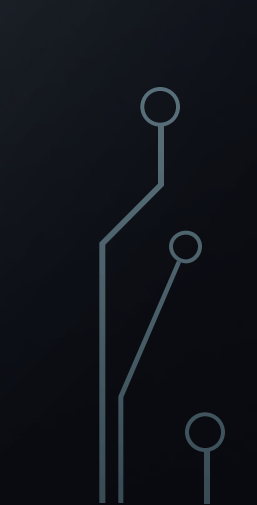

If you see something like this:

```
def foo(first: Int)(second: Int): Int = first + second
```

...you can essentially think of it as this:

```
def foo(first: Int, second: Int): Int = first + second
```

When you see this, it is called "currying" and it's a way of building "partial functions". It's a little esoteric, and it's not important to dive too deeply into this.



# IMPLICIT PARAMETERS (PART 1: QUASI-DEPENDENCY INJECTION)

**Problem:** your code is getting messy because you're passing default objects around between methods.

```
def makeString(n: Double, f: NumberFormat): String = f.format(n)

def reportError(n: Double, f: NumberFormat): String =
  makeString(n,f) + " isn't allowed here."

def printDouble(n: Double, f: NumberFormat): String =
  makeString(n * 2,f)
```

This isn't too bad looking (assuming those functions really need to have a `NumberFormat` object around), but once you start using it, the code looks a bit tedious, always passing around the formatter...

```
scala> val formatter = NumberFormat.getPercentInstance()  
val formatter: java.text.NumberFormat = java.text.DecimalFormat@674dc  
scala> reportError(1.2, formatter)  
val res3: String = 120% isn't allowed here.  
scala> printDouble(0.3, formatter)  
val res4: String = 60%
```

**Solution:** rework your functions slightly so that they take two sets of arguments: the primary parameters and then the “implicit” parameters...Then

```
def makeString(n: Double)(implicit f: NumberFormat): String = f.format(n)

def reportError(n: Double)(implicit f: NumberFormat): String =
  makeString(n)(f) + " isn't allowed here."

def printDouble(n: Double)(implicit f: NumberFormat): String =
  makeString(n * 2)(f)
```

Next, when you're ready to use these functions, define an implicit value:

```
scala> implicit val formatter = NumberFormat.getPercentInstance()
val formatter: java.text.NumberFormat = java.text.DecimalFormat@674dc
scala> reportError(-1.2)
val res1: String = -120% isn't allowed here.
```

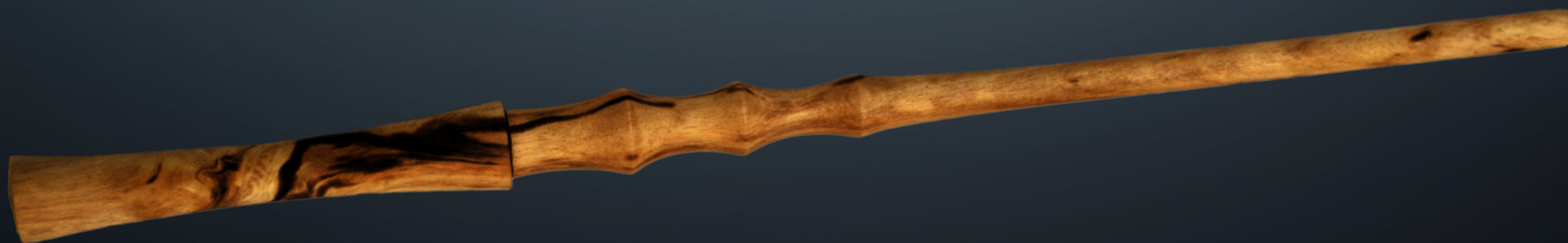


## Notes:

1. You can still explicitly pass the second set of arguments if you don't want to use implicit values. (i.e. you haven't defined an implicit value, or you want to override it with a different value)
2. You can only have one implicit value for any type/class in scope. Otherwise the compiler won't know which to use and will throw an error.
3. This example is a bit contrived (although you might pass configuration parameters this way) and more often you want to be working with implicit functions (functional programming) or typeclasses.
4. Dotty (Scala 3) replaces this with the keyword "given"

# A QUICK DIVERSION: TYPE CLASSES

THE **WIZARD'S MAGIC WAND** IN ADVANCED SCALA FUNCTIONAL PROGRAMMING



# START WITH AN OBJECT THAT HELPS YOU DO SOMETHING (IT EXPOSES FUNCTIONS)

In an earlier example, we used a Java `NumberFormat` class. This does two things:

1. Parse a string to get a number ( “103.4%” → 1.034 )
2. Turn a number into a formatted string ( 2 → “200%” )

This allows an abstraction where one `NumberFormat` object may work with percentages, another may work with currency, a third may be used to convert numbers into strict scientific notation, etc.

This is something that helps perform  
some operation. (e.g. formatting)

Here's where I  
use this helper.

```
def makeString(n: Double)(implicit f: NumberFormat): String = f.format(n)

def reportError(n: Double)(implicit f: NumberFormat): String =
  makeString(n)(f) + " isn't allowed here."

def printDouble(n: Double)(implicit f: NumberFormat): String =
  makeString(n * 2)(f)
```

Now, let's abstract this pattern to make it more  
powerful...

# INTRODUCING THE TYPE CLASS

Define the functionality,  
not any concrete  
implementation.



Generic: no pre-assumption of  
what class it will work with.



```
trait Converter[T] {  
  def parse(s: String): T  
  def format(o: T): String  
  def formalFormat(o: T): String =  
    "The answer is " + format(o)  
}
```

*Strictly speaking, a Type Class is  
something that will convert a class (T)  
into a new class/interface  
(Converter[T]).*

← May include some logic, as  
long as it's generically  
written.

# CREATING INSTANCES OF THE TYPE CLASS:

PLUG IN THE TYPE & ADD THE IMPLEMENTATION DETAILS

```
trait Converter[T] {  
  def parse(s: String): T  
  def format(o: T): String  
  def formalFormat(o: T): String = "The answer is " + format(o)  
}
```

Alternative ways to create  
instances.

```
{  
  class NumberConverter(f: NumberFormat) extends Converter[Number] {  
    override def parse(s: String): Number = f.parse(s)  
    override def format(o: Number): String = f.format(o)  
  }
```

```
{  
  def numberConverter(f: NumberFormat) = new Converter[Number] {  
    override def parse(s: String): Double = s.toDouble  
    override def format(x: Double): String = f.format(x)  
  }
```

# CREATING INSTANCES OF THE TYPE CLASS:

## PLUG IN THE TYPE & ADD THE IMPLEMENTATION DETAILS

```
trait Converter[T] {  
  def parse(s: String): T  
  def format(o: T): String  
  def formalFormat(o: T): String = "The answer is " + format(o)  
}
```

```
case class Color(red: Double, green: Double, blue: Double)
```

```
object ColorConverter extends Converter[Color] {  
  override def parse(s: String): Color = s match {  
    case "Red" => Color(1.0, 0.0, 0.0)  
    case "Green" => Color(0.0, 1.0, 0.0)  
    case "Blue" => Color(0.0, 0.0, 1.0)  
    case _ => Color(0.0, 0.0, 0.0)  
  }  
  override def format(o: Color): String = o match {  
    case Color(1.0, 0.0, 0.0) => "Red"  
    case Color(0.0, 1.0, 0.0) => "Green"  
    case Color(0.0, 0.0, 1.0) => "Blue"  
    case _ => "Unknown Color"  
  }  
}
```

# IMPLICIT PARAMETERS (PART 2: TYPE CLASSES, AKA “PIMP MY LIBRARY”)

**Goal 1:** seamlessly “attach” functionality to classes without touching the classes themselves.

**Goal 2:** write generic code that can be applied to many different classes without repetition.

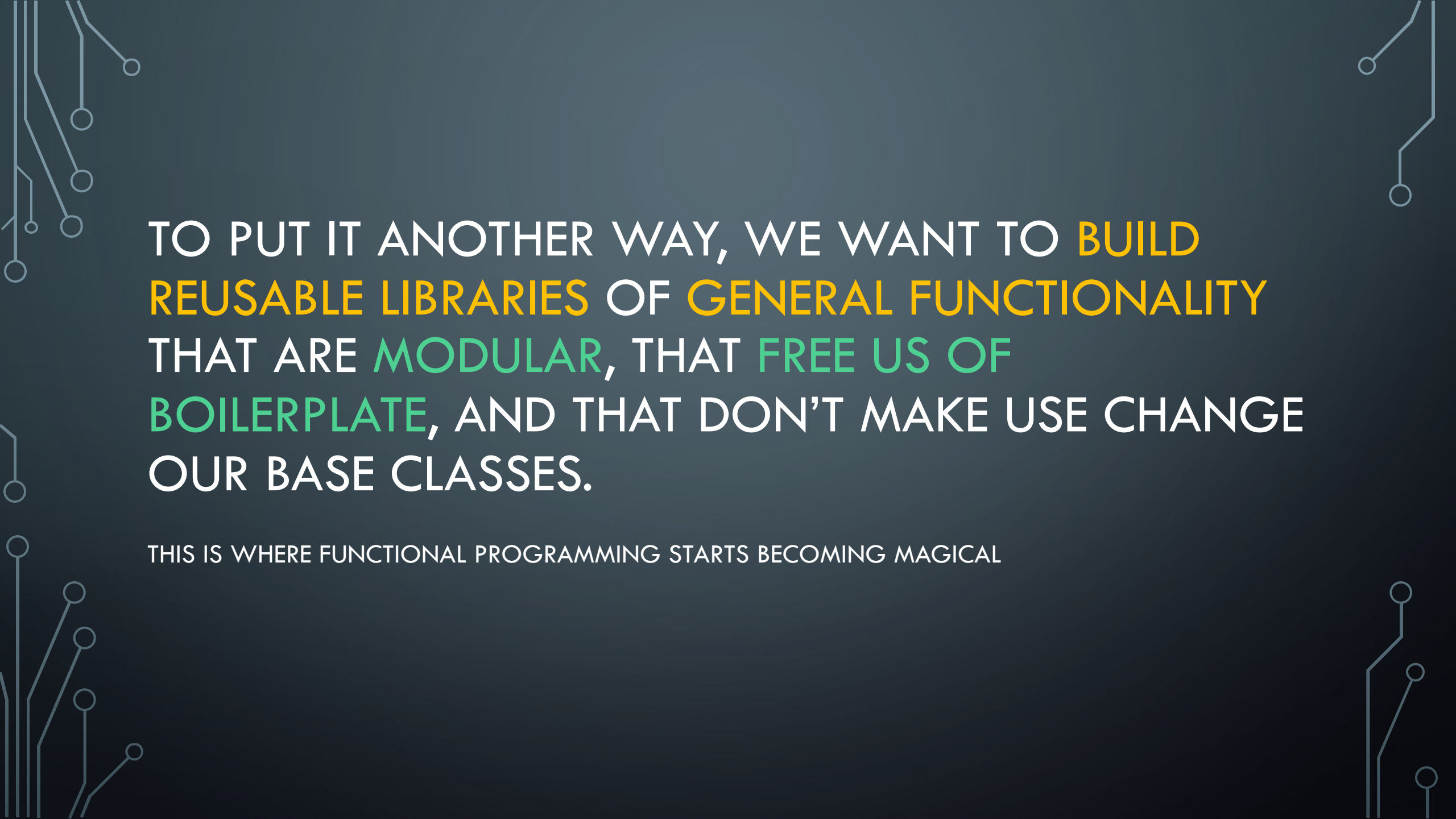
```
val color = "Red".convertTo[Color]
```

```
val pct = "130%".convertTo[Number]
```

↑  
Common class such  
as String.

↑  
Bolted-on function, generically  
written so we can add support  
for new types ad-hoc.



The background is a dark blue-grey gradient. In the corners, there are decorative white line art elements resembling circuit boards or neural networks, with lines and small circles connecting them.

TO PUT IT ANOTHER WAY, WE WANT TO BUILD  
REUSABLE LIBRARIES OF GENERAL FUNCTIONALITY  
THAT ARE MODULAR, THAT FREE US OF  
BOILERPLATE, AND THAT DON'T MAKE USE CHANGE  
OUR BASE CLASSES.

THIS IS WHERE FUNCTIONAL PROGRAMMING STARTS BECOMING MAGICAL

# PUTTING IT ALL TOGETHER IN 3 STEPS

1. Declare the Type Class
2. Create a type-specific implementation (and have the implementation be an implicit value so we can inject it in step 3)
3. Use implicit classes to bolt the implementation into the original target classes

*et voila! Magic!*

Bolt object creation onto String and  
formatting onto T with implicit classes

```
trait Converter[T] {  
  def parse(s: String): T  
  def format(o: T): String  
  def formalFormat(o: T): String = "The answer is " + format(o)  
}
```

```
object Converter {  
  implicit class StringOps(s: String) {  
    def convertTo[T](implicit c: Converter[T]): T = c.parse(s)  
  }  
  implicit class ConvOps[T](o: T) {  
    def formatObj(implicit c: Converter[T]): String = c.format(o)  
    def formal(implicit c: Converter[T]): String = c.formalFormat(o)  
  }  
}
```

Make sure implementations are defined and in scope so  
they can be found and injected.

```
implicit val color = ColorConverter  
implicit val percent =  
  new NumberConverter(NumberFormat.getPercentInstance())  
}
```

YOUR “LIBRARY” IS DONE. THE HARD PART IS OVER.  
NOW THE FUNCTIONALITY IS SUPER-EASY TO USE!

```
import example.Converter._  
import example.Color  
  
scala> val color = "Red".convertTo[Color]  
val color: Color = Color(1.0,0.0,0.0)  
scala> val pct = "130%".convertTo[Number]  
val pct: Number = 1.3  
scala> val colorStr = Color(0.0, 1.0, 0.0).formatObj  
val colorStr: String = Green  
scala> val pctStr = pct.formal  
val pctStr: String = The answer is 130%
```



# ONE LAST (MORE PRACTICAL) EXAMPLE



Leveraging the “Numeric” Type Class in Scala’s build-in core library

# PAIRWISE ADDITION OF NUMERIC SEQUENCES

**Goal:** be able to add two sequences (lists) together with the simple `+` symbol.

I.e. for `x = List(1,2)` and `y = List(3,7)`, convert this:

```
(x zip y).map(t => t._1 + t._2)
```

into this:

```
x + y
```

*And I want this to work for lists of Integers, Longs or Doubles without re-writing boilerplate code for each!*

# INTRODUCING `scala.math.Numeric[T]` TYPE CLASS

- This is part of the Scala library.
- `Numeric[T]` has implicitly defined instances for `Float`, `Integer`, `Double`, `Long`
- `Numeric[T]` provides functionality to add, multiple, compare, negate, etc. and also has values for one and zero.

# ORIGINAL (NON-GENERIC) IMPLEMENTATION

```
implicit class Ops(list: List[Double]) {  
  def +(other: List[Double]): List[Double] = {  
    (list zip other).map(t => t._1 + t._2)  
  }  
}
```



# BASIC (GENERIC) IMPLEMENTATION

```
implicit class Ops[T](list: List[T]) {  
  def +(other: List[T])(implicit ntc: Numeric[T]): List[T] = {  
    (list zip other).map(t => ntc.plus(t._1, t._2))  
  }  
}
```

# SLIGHTLY MORE TYPESAFE DEFINITION

This strictly means “for any value T where we know there is a Numeric[T] type class available”. So this implicit class will only try to bolt-on this operation to lists of “numeric” types. A List[String] would not get this functionality.

```
implicit class Ops[T: Numeric](list: List[T]) {  
  def +(other: List[T])(implicit ntc: Numeric[T]): List[T] = {  
    (list zip other).map(t => ntc.plus(t._1, t._2))  
  }  
}
```

# ADD SUPPORT FOR UNEVEN LISTS

If one list is longer than the other, then pad the shorter list with zeros before zipping and adding. Fortunately, `Numeric[T]` also provides a `.zero` member so we can properly pad the lists!

```
implicit class Ops[T: Numeric](list: List[T]) {  
  def +(other: List[T])(implicit ntc: Numeric[T]): List[T] = {  
    val len = Math.max(list.length, other.length)  
    val zero = ntc.zero  
    (list.padTo(len, zero) zip other.padTo(len, zero)).map(t =>  
      ntc.plus(t._1, t._2))  
  }  
}
```

# FOLLOW-UP READING

- Scala with Cats, chapter 1
- It's FREE
- Chapter 1 goes through implicits and type classes
- Walks through a concrete `Show[T]` example that creates a type-safe replacement for Java's `Object.toString()`
- Includes helpful exercises along the way