

# Lessons Learned from 15 Years of Scala in the Wild

Dean Wampler

@deanwampler

Scale by the Bay 2021



DOMINO

# We're Hiring!



The Enterprise MLOps Platform | Domino Data Lab

Products Solutions Customers Resources Partners Company Watch Demo Try Now

Domino Data Lab Secures \$100 Million Funding [Read the story](#)

**FREE TOOL**

## Unparalleled insights into your data science lifecycle

Assess what's slow. Adjust what works.  
Accelerate what scales.

[Start My Assessment](#)

LOCKHEED MARTIN

SCOR  
The Art & Science of Risk

### Our Users Say It Best

“The platform for herding cats”

Senior Director, Data Science

[programming-scala.com](https://programming-scala.com)

O'REILLY®

# Programming Scala

Scalability = Functional Programming + Objects

3rd Edition  
Covers Scala 3



Dean Wampler

The background of the slide features a wide-angle landscape of a mountain range. In the foreground, a paved road with yellow double lines curves from the bottom left towards the center. The surrounding terrain is a mix of dark, rocky mountain slopes and lush green fields with small yellow flowers. A single yellow road sign with a bicycle symbol is visible on the right side of the road.

# How Scala Has Evolved

- Greater Clarity
- From Implicits to Contextual Abstractions
- Improvements to the Type System

The background of the slide features a wide-angle landscape of a mountain range. In the foreground, a paved road with yellow center lines curves away from the viewer. The surrounding terrain is a mix of green grassy fields and rocky, forested mountain slopes. A large, puffy white cloud formation is visible in the upper left, while the upper right shows darker, more overcast skies.

# “Enterprise Scala”

- FP Over OOP
- Should *Everything* Be Typed?
- Less Code Is More

A wide-angle photograph of a rugged mountain range under a blue sky with scattered clouds. In the foreground, a winding asphalt road cuts through a green valley. The mountains are steep, with rocky slopes and patches of green vegetation. The lighting suggests it's either morning or late afternoon.

Greater Clarity

# Python-esque Syntax in Scala 3

```
// Scala 2 braces
trait Monoid[A] {

    def add(a1: A, a2: A): A

    def zero: A
}

integer match {
    case 0 => println("zero")
    case _ => println("other value")
}
```

```
// Scala 3, no braces option
trait Monoid[A]: Option[Monoid]

    def add(a1: A, a2: A): A

    def zero: A

    integer match
        case 0 => println("zero")
        case _ => println("other value")
```

# More “Intentional” Constructs

```
// Implicit Type Conversions
implicit final class ArrowAssoc[A]
  private val self: A) extends AnyVal {
    @inline def -> [B](y: B): (A, B) = (self, y)
    @deprecated("Use `->` instead...", "2.13.0")
    def →[B](y: B): (A, B) = ->(y)
}
```

```
// True Extension Methods
import scala.annotation.targetName
extension [A] (a: A)
  @targetName("arrow2")
  inline def ~>[B](b: B): (A, B) = (a, b)
```

A wide-angle photograph of a mountainous landscape. In the foreground, a field of purple and yellow wildflowers covers a rocky hillside. A dirt path leads from the bottom center towards the horizon. In the middle ground, a steep mountain slope covered in green trees and shrubs descends. The background features a range of majestic, rugged mountains under a clear blue sky.

# From Implicits to Contextual Abstractions

Implicits are a *mechanism* with idiomatic usage.  
Givens and using clauses are more intentional.

```
trait Semigroup[T]:  
  extension (t: T)  
    infix def combine(other: T): T  
    @targetName("plus")  
    def <+>(other: T): T = t.combine(other)  
  
trait Monoid[T] extends Semigroup[T]:  
  def unit: T  
  
given StringMonoid: Monoid[String] with  
  def unit: String = ""  
  extension (s: String)  
    infix def combine(other: String): String =  
      s + other
```

```
scala>"one" <+> ("two" <+> "three")  
| ("one" <+> "two") <+> "three"  
val res1: String = onetwothree  
val res2: String = onetwothree  
  
scala> "one" <+> StringMonoid.unit  
| StringMonoid.unit <+> "one"  
val res3: String = one  
val res4: String = one
```

Implicits are a *mechanism* with idiomatic usage.  
Givens and using clauses are more intentional.

```
trait Semigroup[T]:  
  extension (t: T)  
    infix def combine(other: T): T  
    @targetName("plus")  
    def <+>(other: T): T = t.combine(other)  
  
trait Monoid[T] extends Semigroup[T]:  
  def unit: T  
  
given NumericMonoid[T: Numeric]: Monoid[T] with  
  def unit: T = summon[Numeric[T]].zero  
  extension (t: T)  
    infix def combine(other: T): T =  
      summon[Numeric[T]].plus(t, other)
```

```
scala> 2 <+> (3 <+> 4)  
| (2.2 <+> 3.3) <+> 4.4  
| (BigInt(2) combine BigInt(3))  
|   combine BigInt(4)  
|  
val res5: Int = 9  
val res6: Double = 9.9  
val res7: BigInt = 9  
  
scala> 2 <+> NumericMonoid[Int].unit  
| NumericMonoid[Double].unit <+> 3.3  
val res8: Int = 2  
val res9: Double = 3.3
```



Implicits are a *mechanism* with idiomatic usage.  
Givens and using clauses are more intentional.

```
trait Context:  
  def info: String  
given Context = new Context:  
  def info: String = "Cloud!"
```

```
def process(name: String)(using Context): String =  
  s"$name-${summon[Context].info}"
```

```
scala> process("AWS")  
val res0: String = "AWS-Cloud!"  
  
scala> given ctx: Context = new Context:  
|   def info: String = "Also Cloud!"  
|  
lazy val ctx: Context  
  
scala> process("Azure")(using ctx)  
val res1: String = Azure-Also Cloud!
```

# Improvements to the Type System



## Opaque type aliases: Almost like regular types, but without the overhead.

```
object Logarithms:
    opaque type Logarithm = Double

    // These are the two ways to lift to the Logarithm type
    def apply(d: Double): Logarithm = math.log(d)
    def safe(d: Double): Option[Logarithm] =
        if d > 0.0 then Some(math.log(d)) else None

    // Extension methods define an opaque type's public APIs
    extension (x: Logarithm)
        def toDouble: Double = math.exp(x)
        def + (y: Logarithm): Logarithm = Logarithm(math.exp(x) + math.exp(y))
        def * (y: Logarithm): Logarithm = x + y
```

# Intersection Types

```
trait Resettable:  
    override def toString: String = "Resettable:"+super.toString  
    def reset(): Unit  
  
trait Growable[T]:  
    override def toString: String = "Growable:"+super.toString  
    def add(t: T): Unit  
  
def f(x: Resettable & Growable[String]): String =  
    x.reset()  
    x.add("first")  
    x.add("second")  
    x.toString
```

Only allowed values must  
be of **both** types  
Resettable **and** Growable.

# Union Types

```
case class User(name: String, password: String)

def getUser(id: String, dbc: DBConnection): String | User | Seq[User] =
  try
    val results = dbc.query(s"SELECT * FROM users WHERE id = $id")
    results.size match
      case 0 => s"No records found for id = $id"
      case 1 => results.head.as[User]
      case _ => results.map(_.as[User])
  catch
    casedbe: DBException =>dbe.getMessage

getUser("1234", myDBConnection) match
  case message: String => println(s"ERROR: $message")
  case User(name, password) => println(s"Hello user: $name")
  case seq: Seq[User] => println(s"Hello users: $seq")
```

Must use pattern matching  
to determine the actual  
type of the instance.



# “Enterprise Scala”

Unlearning *Enterprise Java* habits

A wide-angle photograph of a dramatic sunset or sunrise. The sky is filled with thick, dark clouds that are partially illuminated from below, creating a bright, glowing texture. The overall mood is moody and atmospheric.

FP Over OOP

# Is anything more concise than SQL?

```
SELECT * FROM users WHERE id = "Dean Wampler"
```

Like SQL, functional code tends to be very concise and to the point, where composable operations enable fast, efficient programming

# Parametric Polymorphism

```
def foo1[T](xs: Seq[T]): Int  
def foo2(xs: Seq[Int]): Int
```

What can we deduce about these methods?? The first can have **only one** possible implementation.  
No ambiguity!

<https://medium.com/scala-3/the-value-of-parametric-polymorphism-e76fb9a516b>



# Should *Everything* Be Typed?

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
      ports:
        - containerPort: 80
```

## When should we *avoid* static typing??

Should we faithfully  
**duplicate** this logic in our  
Scala code?? Can we use  
templates and minimize  
knowledge instead?

example from:  
<https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>

A wide-angle photograph of a mountainous landscape. In the foreground, a dark blue lake with small ripples stretches across the frame. Behind the lake, several rugged mountains rise, their slopes covered in patches of green vegetation and exposed brown rock. The sky above is filled with large, billowing white clouds against a deep blue background.

Less (Code) Is More

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
  def main(a: Array[String]) = {

    val sc = new SparkContext("local[*]", "Inverted Index")

    sc.textFile("data/crawl").map { line =>
      val Array(path, text) =
        line.split("\t", 2)
      (path, text)
    }.flatMap {
      case (path, text) =>
        text.split("""\W+""") map {
          word => (word, path)
        }
    }.map {
      case (w, p) => ((w, p), 1)
    }.reduceByKey {
      case (n1, n2) => n1 + n2
    }.map {
      case ((w, p), n) => (w, (p, n))
    }.groupByKey
    .mapValues { iter =>
      iter.toSeq.sortBy {
        case (path, n) => (-n, path)
      }.mkString(", ")
    }.saveAsTextFile("/path/out")
    sc.stop()
  }
}
```

from:

<https://deanwampler.github.io/polyglotprogramming/papers/Spark-TheNextTopComputeModel.pdf>

# “Inverted Index” in Spark

- When your code is this concise, do you *really* need:
  - Dependency injection frameworks?
  - Fancy mocking libraries for testing?
  - Lots of design patterns?
  - Factories, Adapters...
- Lots of micro services to partition the logic?

# Questions?

@deanwampler

[deanwampler.com/talks](http://deanwampler.com/talks)

<https://deanwampler.medium.com>

[dominodatalab.com](http://dominodatalab.com)

