Scala 3

David Hoepelman

13-04-2021

https://github.com/dhoepelman/xke-scala3
URL to presentation

Roadmap

Worked on since ~2014

2014-2016: DOT calculus
2015-2019: Dotty compiler
2020: Dotty renamed to Scala 3

2021-03: 3.0.0-RC2 released

Full release in 2021 (?)

Ecosystem

✅ Sbt	1.5.0 - 2021-04-03	
☐ Scalafmt	RC expected in 2020-04	
✅ Scalatest	3.2.4 - 2021-02-19	
✅ Cats	2.3.0 - 2020-11-26	
☐ Akka		
☐ Play		
✅ IntelliJ	2020.1 - 2020-04-08	Still a tad shaky in my experience

What’s new

- Syntax
- Developer QoL
- Compatibility between Scala versions
- Type system advancements
- Implicits redesigned: intent over mechanism
- Meta-programming and macro’s redesigned

Syntax

In order from least to most controversial

Syntax

```
// import * is now the recommended alternative to import _
// Like basically all other languages
import syntax._
import syntax.*
```

```
// ? is now the generic wildcard instead of _
// Like basically all other languages
val list1: List[_] = List.empty
val list2: List[?] = List()
```

Syntax

```
// new is now optional
// Like: Kotlin
val dogNew = new Dog("Fluffy")
val dog = Dog("Fluffy")
```

```
// Because all classes have generated apply() methods like case classes
class Dog(val name: String)
val dogApply = Dog.apply( "1")
```

Syntax

```
// variable and function declarations can now be top-level
// package object is now  unnecessary and deprecated
```

```
val x = 0
def foo(): Unit = {}
def bar(): Unit = {}
```

Syntax

```
// Braces and brackets are always optional
```

```
// Scala 2 & 3
for {
 p <- products
 if p.category == "Fruit"
} yield p
```

```
// Scala 3
for
 p <- products
 if p.category == "Fruit"
yield p
```

```
// Scala 2 & 3
if(x < 0) {
 foo()
} else {
 bar()
}
```

```
// Scala 3
if x < 0 then
 foo()
else
 bar()
```

Syntax

Whitespace syntax

Very controversial change: https://contributors.scala-lang.org/t/feedback-sought-optional-braces/4702

Syntax

Whitespace syntax

```
class BracesDog(name: String) extends Animal
{
 val coatColor = "Brown"
 def bark()    = println("Woof")
}

val bracesMatch = either() match {
 case Left(x)   => true
 case Right(x)  => false
}
```

```
class WhitespaceDog(name: String) extends
Animal:
 val coatColor = "Brown"
 def bark()    = println("Woof")

val whitespaceMatch = either() match
 case Left(x)  => true
 case Right(x) => false
```

Enumerations

```
enum Color(val name: String) {
 case Red extends Color("Red")
 case Blue extends Color("Blue")
 case Green extends Color("Green")
}

// List elements
val values: Array[Color] = Color.values
// Name to element
val red: Color = Color.valueOf("Red")

// Exhaustive matching
println(red match {
 case Color.Red => "It's red!"
 case Color.Blue => "It's blue!"
 case Color.Green => "It's green!"
})

// They can be made a java-compatible enum
enum JavaCompatible(val x: String) extends Enum[JavaCompatible] {
 case Member extends JavaCompatible("x")
}
```

Language features

```
// Union types
// Dual of intersection types

type MyEither[L, R] = L | R

// Intersection is the dual, already exists in "extends ... with ..."
type CanSerialize[T] = T & Serializable

// with is now an alias for intersection, but available because order can matter
trait Base
trait Sub extends Base with Serializable
// with is normalized to &
val x: Base & Serializable = new Sub {}
```

Language features

```
// Union types
// Dual of intersection types

type MyEither[L, R] = L | R

// Intersection is the dual, already exists in "extends ... with ..."
type CanSerialize[T] = T & Serializable

// with is now an alias for intersection, but available because order can matter
trait Base
trait Sub extends Base with Serializable
// with is normalized to &
val x: Base & Serializable = new Sub {}
```

Language features

```
// However, it is not as smart as e.g. Typescript by far. This will not compile.

// Typescript can do things like this

val result: Int | "INTERNAL SERVER ERROR" = 1
def process[T : ClassTag](result: T | "INTERNAL_SERVER_ERROR"): T = {
  result match {
    case "INTERNAL_SERVER_ERROR" => throw new Exception("Uh oh")
    case t: T => t
  }
}

val y: Int = process(result)
```

Language features

```
// Explicit nulls


// Compile error if -Yexplicit-nulls is added
// val notnull: String = null

// union types to the rescue
val nullable: String | Null = null
```

Implicits

- Powerful
- Unique to scala
- Difficult to grok
- Low level, overloaded

Implicits

Currently used for

- Extension methods
- Implicit conversion
- Typeclasses
- Type-level programming
- Dependency injection
- …

Scala 3 goal: “Intent over mechanism”

Implicit replacement: Extensions

```scala
// Extension functions
trait Animal
case class Dog(name: String) extends Animal

extension (doggo: Dog) def bark1(): Unit = println("woof")
extension [T <: Animal](animal: T) def walk(): Unit = println("walk")

// Old syntax:
implicit class DogExtensions(val doggo: Dog) extends AnyVal {
  def bark2(): Unit = println("woof")
}

val woof1 = Dog("1").bark1()
val woof2 = Dog("2").bark2()
val woof3 = Dog("3").walk()
```

Implicit replacement: Conversions

```scala
case class MyString(val s: String)

val a: MyString = {
 import scala.language.implicitConversions
 given Conversion[String, MyString] with
   def apply(str: String): MyString = MyString(str)
 "A"
}

val b: MyString = {
 import scala.language.implicitConversions
 // old syntax
 implicit def stringToMyString(s: String): MyString = MyString(s)
 "b"
}
```

Implicit replacement: given and using

```
trait ToJson[T] {
  def toJson(t: T): String
}

given jsonInt: ToJson[Int] with {
 override def toJson(t: Int): String = t.toString
}

given jsonMap[T] (using toJson: ToJson[T]): ToJson[Map[String, T]] with {
 override def toJson(map: Map[String, T]): String =
   map
    .map( (k, v) => s"""${k}" : ${toJson.toJson(v)}""" )
    .mkString("{\n", ",\n", "\n}")
}

def makeJson[T](value:T)(using toJson: ToJson[T]): String = toJson.toJson(value)

makeJson(Map("a" -> 1, "b" -> 2))
```

Meta-programming

Totally redesigned

- Inline
- Compile-time operations
- Quasi-quotes
- Tasty: Scala AST for reflection

Compatibility

We’re totally prepared for the ~~Python 3~~ Scala 3 migration

- All* Scala 2.13 code is valid Scala 3
- Scala 3 can use 2.13 libraries
- Scala 2.13.5 can use Scala 3 libraries (depending on features used)
- Plan: All Scala 3.x versions can use all 3.x libraries

Compatibility

- View bounds and some other esoteric constructs: gone
- Macro’s: uh-oh

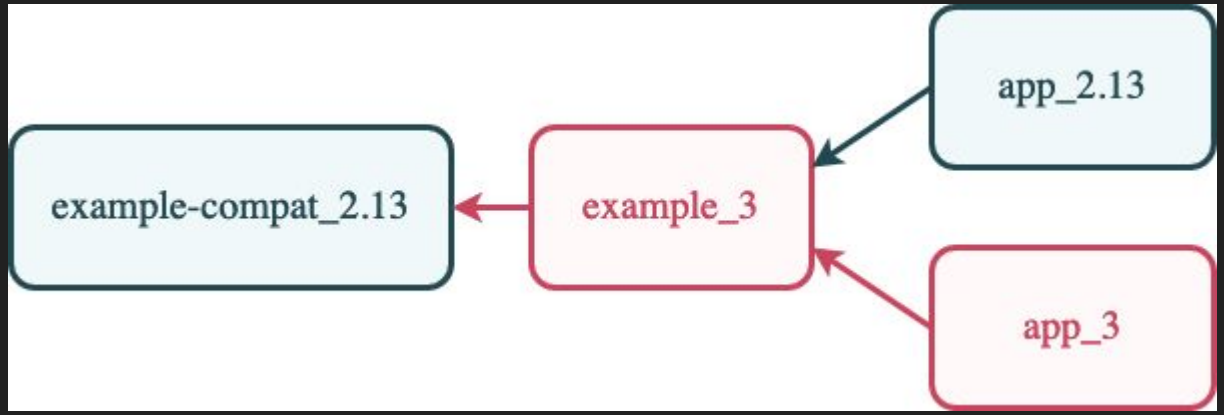
Scala 2 macro’s are Scala 2 compiler
Scala 3 macro’s are TASTY

Compatibility

- View bounds and some other esoteric constructs: gone
- Macro's: uh-oh

Scala 2 macro's are Scala 2 compiler
Scala 3 macro's are TASTY

Solution: define Scala 2 & 3 macro's in single artifact



Discussion

- Changes
- Migration
- State of Scala