

# A Taste of Dotty

copyright 2021 Hermann Hueck

<https://github.com/hermannhueck/taste-of-dotty>

# Abstract

This presentation is an introduction to Dotty / Scala 3 and an overview of many features.

It covers those features which I deem most important for Scala application developers.

For detailed information see the [Dotty documentation](#).

The code examples are partly my own. I also took many examples (unchanged or modified) from the Dotty Documentation.

The presentation also contains many links to specific chapters in the Dotty docs.

# Agenda (1/3)

- » Design Goals
- » Project Setup
- » Using Scala 2 Libraries
- » Top Level *def*'s and *val*'s
- » Indentation / Optional Braces
- » New Control Syntax
- » Main Methods
- » Constructor Invocations w/o *new*
- » Traits Parameters
- » Enums and ADTs
- » Intersection Types

# Agenda (2/3)

- » Union Types
- » Contextual Abstractions
- » Implicit Conversions
- » Extension Methods
- » Givens
- » Type Lambdas
- » Typeclasses
- » Opaque Type Aliases
- » Context Functions
- » Dependent Function Types
- » Tuples are HLists

# Agenda (3/3)

- » Match Types
- » Export Clauses
- » Explicit Nulls
- » inline
- » Multiversial Equality
- » Typeclass Derivation
- » Given By-Name Parameters
- » Implicit Resolution
- » Overload Resolution
- » Other Features
- » Resources

# Design Goals<sup>1</sup>

---

<sup>1</sup><https://dotty.epfl.ch/docs/index.html>

# Design Goals

- » build on strong foundations (DOT Calculus)
- » improve language consistency,
- » eliminate surprising behaviours, puzzlers
- » better (type) safety and ergonomics, simplify where possible
- » improve performance

# Project Setup<sup>2</sup>

---

<sup>2</sup><https://dotty.epfl.ch/docs/usage/getting-started.html>

# IDE Support<sup>3</sup>

- » Dotty comes with a built-in Dotty Language Server.
- » Should work with any editor that supports LSP.  
(Language Server Protocol)
- » Only Visual Studio Code is officially supported.

---

<sup>3</sup> <https://dotty.epfl.ch/docs/usage/ide-support.html>

# Prerequisites

- » *sbt* is installed.
- » VSCode is installed.
- » Make sure you can start VSCode with the CLI command *code*. This is default on all systems except macOS.  
(macOS users should follow the instructions below to install the *code* command.<sup>4</sup>)

---

<sup>4</sup> [https://code.visualstudio.com/docs/setup/mac#\\_command-line](https://code.visualstudio.com/docs/setup/mac#_command-line)

# New sbt Project

- » create new project: *sbt new scala/scala3.g8*
- » (or: *sbt new scala/scala3-cross.g8* for a cross-build project)
- » *cd* to project directory
- » in the project directory: *sbt launchIDE*  
(starts VSCode with the current folder as workspace,  
installs the Dotty Language Server in VSCode)

# build.sbt

```
val scala3Version = "3.0.0-M3"

lazy val root = project
  .in(file("."))
  .settings(
    name := "scala3-simple",
    version := "0.1.0",
    scalaVersion := scala3Version,
    libraryDependencies += "com.novocode" % "junit-interface" % "0.11" % "test"
  )
```

# project/plugin.sbt

```
// sbt-dotty plugin  
addSbtPlugin("ch.epfl.lamp" % "sbt-dotty" % "0.5.2")
```

# project/build.properties

```
// change to latest sbt version  
sbt.version=1.4.7
```

# Using Scala 2 Libraries<sup>4a</sup>

---

<sup>4a</sup><https://github.com/lampepfl/dotty-example-project#getting-your-project-to-compile-with-dotty>

# Using Scala 2 Libraries

- » Dotty can already utilize Scala 2 libraries.
- » This works because Dotty is currently retro-compatible with Scala 2.x.
- » This allows to migrate to Dotty, even before 3rd party dependencies have been migrated.

```
// build.sbt
```

```
libraryDependencies ++= Seq(  
  "org.typelevel" %% "cats-effect" % catsEffectVersion,  
  "org.scalatest" %% "scalatest" % scalaTestVersion % Test  
).map(module => module.withDottyCompat(scalaVersion.value))
```

# Top Level *def*'s and *val*'s<sup>5</sup>

---

<sup>5</sup> <https://dotty.epfl.ch/docs/reference/dropped-features/package-objects.html>

# Top Level *def*'s and *val*'s

- » Scala 2: *def*'s and *val*'s must be defined in a *trait*, *class* or *object*.
- » Scala 3: *def*'s and *val*'s can be defined at the top level.
- » Scala 2: To provide *def*'s and *val*'s directly in a package, one has to use package objects.
- » Scala 3: Package objects are still available in 3.0, but will be deprecated and removed in 3.1 or 3.2.

```
// whatever.scala
package dotty.samples

import scala.util.chaining._

val r = scala.util.Random

def randomInt(): Int =
  r.nextInt

def boxed(what: String): String = {
  val line = "\u2500" * 50
  s"$line\n${what.toString}\n$line"
}

def printBoxed(what: String): Unit =
  what pipe boxed pipe println
```

# Indentation / Optional Braces<sup>6</sup>

---

<sup>6</sup><https://dotty.epfl.ch/docs/reference/other-new-features/indentation.html>

# Indentation / Optional Braces

- » Braces are optional.
- » Without braces indentation becomes significant to delimit a block of code.
- » An optional colon at the end of the line starts a new indentation block.
- » End markers are optional.

» with braces:

```
// Scala 2 + 3:  
trait MyTrait {  
    def boxed(what: Any): String = {  
        val line = "\u2500" * 50  
        s"$line\n${what.toString}\n$line"  
    }  
}
```

» without braces:

```
// Scala 3:  
trait MyTrait:  
    def boxed(what: Any): String =  
        val line = "\u2500" * 50  
        s"$line\n${what.toString}\n$line"  
end MyTrait // optional end marker
```

# New Control Syntax<sup>7</sup>

---

<sup>7</sup><https://dotty.epfl.ch/docs/reference/other-new-features/control-syntax.html>

# if ... then ... else

```
val x = 42
```

```
if x < 0 then -x else x
```

```
if x < 0 then
```

```
    "negative"
```

```
else if x == 0 then
```

```
    "zero"
```

```
else
```

```
    "positive"
```

# while ... do (while-loop)

```
var x = 42
```

```
def f(x: Int): Int = x - 10
```

```
while x >= 0 do x = f(x)
```

```
while
```

```
  x >= 0
```

```
do
```

```
  x = f(x)
```

# for ... do (for-loop)

```
val xs = List(1, 2, 3)  
val ys = List(10, 20, 30)
```

```
for x <- xs if x > 0  
do println(x * x)
```

```
for  
  x <- xs  
  y <- ys  
do  
  println(x + y)
```

# for ... yield (for-comprehension)

```
val xs = List(1, 2, 3)  
val ys = List(10, 20, 30)
```

```
for x <- xs if x > 0  
yield x * x
```

```
for  
  x <- xs  
  y <- ys  
yield x + y
```

# Main Methods<sup>8</sup>

---

<sup>8</sup><https://dotty.epfl.ch/docs/reference/changed-features/main-functions.html>

# Main Methods

```
@main def happyBirthday(age: Int, name: String, others: String*): Unit =  
  
  val congrats = s"Happy Birthday at age $age to $name" ++ {  
    if others.isEmpty  
      ""  
    else  
      " and " ++ others.mkString(", ")  
  } ++ ". "  
  
  println(congrats)
```

# Main Methods

- » A `@main` annotation on a method turns this method into an executable program.
- » The method must be static, i.e. not defined within a class or trait.
- » If annotated the method name is arbitrary.
- » Argument types can not only be `Array[String]`.
- » Any argument type is allowed if an instance of typeclass `scala.util.FromString` is in implicit scope.
- » Dotty checks the arguments passed against the signature of the main function.

# Constructor Invocations without `new`<sup>9</sup>

---

<sup>9</sup> <https://dotty.epfl.ch/docs/reference/other-new-features/creator-applications.html>

# Constructors without *new*

- » When constructing instances the *new* keyword is optional.
- » Works not only for case classes but also for regular classes.
- » Works for Java classes too.
- » If no *apply* method is found, the compiler looks for a suitable constructor.

```
val sb =  
  StringBuilder("The keyword 'new'")  
    .append(" is ")  
    .append("optional")  
    .append("!")
```

# Traits Parameters<sup>10</sup>

---

<sup>10</sup> <https://dotty.epfl.ch/docs/reference/other-new-features/trait-parameters.html>

# Trait Parameters

- » Traits can have parameters like classes.
- » Arguments are evaluated before the trait is initialized.
- » They replace early initializers in Scala 2 traits, which have been dropped.

```
trait Greeting(val name: String):  
    def msg = s"How are you, $name"  
  
class C extends Greeting("Bob"):  
    println(msg)  
  
class D extends C with Greeting("Bill"): // COMPILE ERROR  
// [error] trait Greeting is already implemented by superclass C  
// [error] its constructor cannot be called again
```

# Enums and ADTs<sup>11 12</sup>

---

<sup>11</sup> <https://dotty.epfl.ch/docs/reference/enums/enums.html>

<sup>12</sup> <https://dotty.epfl.ch/docs/reference/enums/adts.html>

# Simple Enums

- » *enum* is a new keyword.
- » With *enum* one can define a type consisting of a set of named values.

```
enum Color:  
    case Red, Green, Blue
```

# Java compatible Enums

- » To make your Scala-defined enums usable as Java enums, you can do so by extending *java.lang.Enum*.

```
enum Color extends java.lang.Enum[Color]:  
  case Red, Green, Blue
```

# Enums with Parameters

- » The parameters are defined by using an explicit *extends* clause.

```
enum Color(val escape: String):  
    case Red extends Color(Console.RED)  
    case Green extends Color(Console.GREEN)  
    case Blue extends Color(Console.BLUE)
```

# Methods defined for Enums

```
scala> val red = Color.Red
val red: Color = Red
scala> red.ordinal
val res0: Int = 0
```

# Methods defined on the companion object

```
scala> Color.valueOf("Blue")
val res0: Color = Blue
scala> Color.values
val res1: Array[Color] = Array(Red, Green, Blue)
```

# User-defined members of Enums

- » It is possible to add your own definitions to an enum.
- » You can also define your own methods in the *enum*'s companion object.

```
enum Color(val escape: String):  
  case Red extends Color(Console.RED)  
  case Green extends Color(Console.GREEN)  
  case Blue extends Color(Console.BLUE)  
  // user defined method  
  def colored(text: String) = s"$escape$text${Console.RESET}"  
  
import Color._  
  
val greenHello = Green.colored("Hello World!")
```

# ADTs in Scala 2

- » In Scala 2 ADTS are expressed as sealed traits with a hierarchy of case classes.
- » This syntax is still supported in Scala 3.

```
sealed trait Tree[+T]
object Tree {
  case class Leaf[T](elem: T) extends Tree[T]
  case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]
}

import Tree._

val tree: Tree[Int] = Node(Leaf(1), Node(Leaf(2), Leaf(3)))
```

# ADTs in Scala 3

- » In Scala 3 an ADT can be expressed with *enum* syntax.

```
enum Tree[+T]:  
  case Leaf(elem: T) extends Tree[T]  
  case Node(left: Tree[T], right: Tree[T]) extends Tree[T]  
  
import Tree._  
  
val tree: Tree[Int] = Node(Leaf(1), Node(Leaf(2), Leaf(3)))
```

# ADTs with Syntactic Sugar

- » The *extends* clause can be omitted in many cases.

```
enum Tree[+T]:  
  case Leaf(elem: T)  
  case Node(left: Tree[T], right: Tree[T])  
  
import Tree._  
  
val tree: Tree[Int] = Node(Leaf(1), Node(Leaf(2), Leaf(3)))
```

# ADTs with Methods

- » As all other enums, ADTs can define methods.

```
enum Tree[+T]:  
  case Leaf(elem: T)  
  case Node(left: Tree[T], right: Tree[T])  
  def count: Int = this match  
    case Leaf(_) => 1  
    case Node(left, right) => left.count + right.count  
  
import Tree._  
  
val tree: Tree[Int] = Node(Leaf(1), Node(Leaf(2), Leaf(3)))  
val count = tree.count // 3
```

# Intersection Types<sup>13</sup>

---

<sup>13</sup> <https://dotty.epfl.ch/docs/reference/new-types/intersection-types.html>

# Intersection Types

- » Used on types, the `&` operator creates an intersection type.
- » The type  $A \& B$  represents values that are of the type  $A$  and  $B$  at the same time.
- »  $A \& B$  has all members/properties of  $A$  and all members/properties of  $B$ .
- » `&` is commutative:  $A \& B$  is the same type as  $B \& A$ .
- » `with` is not commutative:  $A \text{ with } B$  is not the same type as  $B \text{ with } A$ .
- » Intersection types will - in the long run - replace compound types:  $A \text{ with } B$

# Intersection Types

```
trait Resettable:  
  def reset(): this.type  
  
trait Growable[T]:  
  def add(x: T): this.type  
  
type ResetGrowable[T] =  
  Resettable & Growable[T]
```

```
class MyClass(var x : Int = 0) extends Resettable, Growable[Int]:  
  def reset() =  
    x = 0  
    this  
  def add(x: Int) =  
    this.x += x  
    this  
  
  def f(x: ResetGrowable[Int]) =  
    x.reset()  
    x.add(-21)  
  
@main def runIntersectExample: Unit =  
  val obj = new MyClass(42) // 42  
  obj.reset() // 0  
  obj.add(10) // 10  
  f(obj) // 21
```

# Union Types<sup>14</sup>

---

<sup>14</sup> <https://dotty.epfl.ch/docs/reference/new-types/union-types.html>

# Union Types

- » A union type  $A \mid B$  comprises all values of type  $A$  and also all values of type  $B$ .
- » Union types are duals of intersection types.
- »  $A \mid B$  contains all members/properties which  $A$  and  $B$  have in common.
- »  $\mid$  is commutative:  $A \mid B$  is the same type as  $B \mid A$ .
- » Pattern matching is the natural way to decide if an  $A \mid B$  is an  $A$  or a  $B$ .
- » Union types are not suited to express coproducts.

```
type Hash = Int

case class UserName(name: String)
case class Password(hash: Hash)

def help(id: UserName | Password): String =
  id match
    case UserName(name) => name
    case Password(hash) => hash.toString

val name: UserName = UserName("Eve")
val password: Password = Password(123)

val nameOrPw: UserName | Password =
  if (true) name else password
```

# Contextual Abstractions<sup>15</sup>

## (Implicits in Scala 2)

---

<sup>15</sup> <https://dotty.epfl.ch/docs/reference/contextual/motivation.html>

# Implicits

- » Implicits are the fundamental way to abstract over context in Scala 2.
- » Hard to understand, error-prone, easily misused or overused, many rough edges.
- » Implicits convey mechanism over intent.
- » One mechanism used for many different purposes:
  - » implicit conversions
  - » extension methods
  - » providing context
  - » dependency injection
  - » typeclasses

# The new Design in Scala 3 (1/2)

- » Focus on intent over mechanism
- » Implicit conversions are hard to misuse.
- » Concise syntax for extension methods
- » New keyword *given*
- » *given* instances focus on types instead of terms.
- » *using* clauses replace *implicit* parameters.
- » *given* imports are distinct from regular imports.

# The new Design in Scala 3 (2/2)

- » Typeclasses can be expressed in a more concise way (also due to the new extension methods).
- » Context bounds remain unchanged in syntax and semantics.
- » Typeclass derivation is supported.
- » Implicit Function Types provide a way to abstract over given clauses.
- » Implicit By-Name Parameters are an essential tool to define recursive synthesized values without looping.
- » Multiversal Equality introduces a special typeclass to support type safe equality.
- » Scala 2 implicits remain available in parallel for a long time (backward compatibility).

# Implicit Conversions<sup>16</sup>

---

<sup>16</sup> <https://dotty.epfl.ch/docs/reference/contextual/conversions.html>

# Implicit Conversions

» *scala.Conversion* is a subclass of *Function1*.

```
package scala  
abstract class Conversion[-T, +U] extends (T => U)
```

» Implicit Conversions must extend *Conversion*.

```
case class Token(str: String)  
given Conversion[String, Token]:  
  def apply(str: String): Token = Token(str)
```

or even more concise:

```
case class Token(str: String)  
given Conversion[String, Token] = Token(_)
```

# Implicit Conversion in Scala 2:

```
case class Token(str: String)
```

```
implicit def stringToToken(str: String): Token = Token(str)
```

Syntax can easily be mixed up with other implicit constructs.

# Extension Methods<sup>17</sup>

---

<sup>17</sup> <https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html>

# Extension Methods

- » Extension methods replace implicit classes of Scala 2.
- » They are defined using the new key word *extension*.
- » The *extension* keyword is followed by a single parameter,
- » which is the object on which the extension method(s) is/are defined.
- » The extension clause is followed by one ore more method definitions.
- » They can be invoked two ways: *method(param)* or *param.method*
- » This syntax also applies to operators.

# Extension Methods

```
case class Circle(x: Double, y: Double, radius: Double)

extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2

val circle = Circle(0, 0, 1)

val cf1 = circle.circumference
val cf2 = circumference(circle)
assert(cf1 == cf2)
```

# Collective Extensions

- » Several extension methods that share the same left-hand parameter type
- » ... can be bundled with **extension**
- » ... moving the common first parameter to the instance.

```
extension [T](xs: List[T])  
  def second: T = xs.tail.head  
  def third: T = xs.tail.second
```

```
extension [T](xs: List[T])(using Ordering[T]) // uses an additional implicit parameter  
  def largest(n: Int) = xs.sorted.takeRight(n)
```

# Givens<sup>18</sup>

---

<sup>18</sup> <https://dotty.epfl.ch/docs/reference/contextual/motivation.html>

# Givens

- » *given* is a new keyword
- » *given*'s in many ways replace implicits
- » more concise, less boilerplate
- » focus on types instead of terms

# Givens: Future Example 1

- » Future requires a *given ExecutionContext* in nearly every method.

```
import scala.concurrent.{Future, ExecutionContext}

// implicit val ec: ExecutionContext = ExecutionContext.global // Scala 2
given ec: ExecutionContext = ExecutionContext.global // variable ec can be omitted

def someComputation(): Int = ???
val future: Future[Int] = Future { someComputation() }

future onComplete {
  case Success(value) => println(value)
  case Failure(throwable) => println(throwable)
}
```

# Givens: Future Example 2

- » This example provides the *ExecutionContext* via *import*.

```
import scala.concurrent.{Future, ExecutionContext}

// import ExecutionContext.Implicits.global // Scala 2
import ExecutionContext.Implicits.{given ExecutionContext}

def someComputation(): Int = ???
val future: Future[Int] = Future { someComputation() }

future onComplete {
  case Success(value) => println(value)
  case Failure(throwable) => println(throwable)
}
```

# Given Instances: *Ord* Example<sup>19</sup>

```
// a type class
trait Ord[T]:
    def compare(x: T, y: T): Int
    extension (x: T)
        def < (y: T) = compare(x, y) < 0
        def > (y: T) = compare(x, y) > 0
```

Typeclass instances to be defined as *given's* ...

---

<sup>19</sup> <https://dotty.epfl.ch/docs/reference/contextual/givens.html>

# *given* Instances

- » They replace *implicit val*'s, *def*'s and *object*'s.
- » They can be defined with only a type omitting a name/symbol.
- » Symbols - if omitted - are synthesized by the compiler.
- » The *given* clause ends with the key word *with*
- » followed by a set of method defs implementing the trait (*Ord* in this case).

# *given Instances for Ord*

```
// instances with symbols
given intOrd: Ord[Int] with
  def compare(x: Int, y: Int) = ???

given listOrd[T](using ord: Ord[T]): Ord[List[T]] with
  def compare(xs: List[T], ys: List[T]): Int = ???

// instances without symbols
given Ord[Int] with
  def compare(x: Int, y: Int) = ???

given [T](using Ord[T]): Ord[List[T]] with
  def compare(xs: List[T], ys: List[T]): Int = ???
```

# *using* Clauses<sup>20</sup>

- » They replace the implicit parameter list.
- » Multiple *using* clauses are allowed.
- » Anonymous *using*'s: Symbols are optional.
- » *given* instances can be summoned with the function *summon*.
- » *summon* replaces Scala 2's *implicitly*.

---

<sup>20</sup><https://dotty.epfl.ch/docs/reference/contextual/using-clauses.html>

# *using* Clauses with Symbols

```
def max[T](x: T, y: T)(using ord: Ord[T]): T =  
  if (ord.compare(x, y) < 0) y else x
```

```
def maximum[T](xs: List[T])(using ord: Ord[T]): T =  
  xs.reduceLeft(max)
```

```
def descending[T](using asc: Ord[T]): Ord[T] = new Ord[T] {  
  def compare(x: T, y: T) = asc.compare(y, x)  
}
```

```
def minimum[T](xs: List[T])(using ord: Ord[T]) =  
  maximum(xs)(using descending)
```

# Anonymous *using* Clauses (without Symbols)

```
def max[T](x: T, y: T)(using Ord[T]): T =  
  if (summon[Ord[T]].compare(x, y) < 0) y else x
```

```
def maximum[T](xs: List[T])(using Ord[T]): T =  
  xs.reduceLeft(max)
```

```
def descending[T](using Ord[T]): Ord[T] = new Ord[T] {  
  def compare(x: T, y: T) = summon[Ord[T]].compare(y, x)  
}
```

```
def minimum[T](xs: List[T])(using Ord[T]) =  
  maximum(xs)(using descending)
```

# Usages

- » When passing a *given* explicitly, the keyword *using* is required in front of the symbol.

```
val xs = List(1, 2, 3)
```

```
max(2, 3) // max of two Ints
```

```
max(2, 3)(using intOrd) // max of two Ints - passing the given explicitly
```

```
max(xs, Nil) // max of two Lists
```

```
minimum(xs) // minimum element of a List
```

```
maximum(xs)(using descending) // maximum element of a List (in desc order)
```

# Context Bounds<sup>21</sup>

- » These remain nearly unchanged.
- » A context bound is syntactic sugar for the last given clause of a method.

```
// using an anonymous given
def maximum[T](xs: List[T])(using Ord[T]): T =
  xs.reduceLeft(max)
```

```
// using context bound
def maximum[T: Ord](xs: List[T]): T =
  xs.reduceLeft(max)
```

---

<sup>21</sup><https://dotty.epfl.ch/docs/reference/contextual/context-bounds.html>

# Given Imports<sup>22</sup>

```
object A:  
  class TC  
  given tc: TC = ???  
  def f(given TC) = ???  
  
object B:  
  import A._ // imports all members of A except the given instances  
  import A.given // imports only the given instances of A  
  
object C:  
  import A.{given, _} // import givens and non-givens with a single import  
  
object D:  
  import A.TC  
  import A.{given TC} // importing by type
```

---

<sup>22</sup><https://dotty.epfl.ch/docs/reference/contextual/given-imports.html>

# Type Lambdas<sup>23</sup>

---

<sup>23</sup> <https://dotty.epfl.ch/docs/reference/new-types/type-lambdas.html>

# Type Lambdas

- » Type Lambdas are new feature in Scala 3.
- » Type Lambdas can be expressed in Scala 2 using a weird syntax with type alias and type projection.
- » The *kind-projector* compiler plugin brought a more convenient type lambda syntax to Scala 2.
- » Type projections are dropped from Scala 3.<sup>24</sup>
- » Type lambdas remove the need for *kind-projector*.

---

<sup>24</sup> <https://dotty.epfl.ch/docs/reference/dropped-features/type-projection.html>

# Type Lambdas

- » A type lambda lets one express a higher-kinded type directly, without a type definition.
- » Type parameters of type lambdas can have variances and bounds.

A parameterized type definition or declaration such as

```
type T[X] = (X, X)
```

is a shorthand for a plain type definition with a type-lambda as its right-hand side:

```
type T = [X] =>> (X, X)
```

# Type Lambda Example: Either Monad Instance

```
// Scala 2 without kind-projector
implicit def eitherMonad[L]: Monad[({type lambda[x] = Either[L, x]})#lambda] = ...
```

```
// Scala 2 using kind-projector
implicit def eitherMonad[L]: Monad[lambda[x => Either[L, x]]] = ...
```

```
// Scala 2 using kind-projector with ? syntax (use * in newer versions of kind-projector)
implicit def eitherMonad[L]: Monad[Either[L, ?]] = ...
```

```
// Scala 3 using a type lambda
given eitherMonad[L]: Monad[[R] -> Either[L, R]] { ... }
```

```
// Scala 3 using compiler option -Ykind-projector
given eitherMonad[L]: Monad[Either[L, *]] { ... }
```

# Typeclasses<sup>26</sup>

## (Monad Example)

---

<sup>26</sup><https://dotty.epfl.ch/docs/reference/contextual/type-classes.html>

# Typeclasses: Monad Trait

```
trait Functor[F[?]]:
```

```
extension [A, B](x: F[A])
```

```
  def map (f: A => B): F[B]
```

```
trait Monad[F[?]] extends Functor[F]:
```

```
def pure[A](a: A): F[A]
```

```
extension [A, B](fa: F[A])
```

```
  def flatMap (f: A => F[B]): F[B]
```

```
extension [A, B] (fa: F[A])
```

```
  override def map (f: A => B): F[B] = flatMap(fa)(f andThen pure)
```

```
extension [A](fa: F[F[A]])
```

```
  def flatten: F[A] = flatMap(fa)(identity)
```

# Typeclasses: Monad Trait

- Type class *Ord* defined an Ordering for some type *A*.
- *Ord* was polymorphic and parameterized with type *A*.
- *Functor* and *Monad* are parameterized with the higher-kinded type  $F[?]$ . (Higher-kinded polymorphism)

# Typeclasses: Monad Instances

```
object Monad:
```

```
  given Monad[List] with
    override def pure[A](a: A): List[A] = List(a)
    extension [A, B](fa: List[A])
      override def flatMap(f: A => List[B]): List[B] =
        fa flatMap f
```

```
  given Monad[Option] with
    override def pure[A](a: A): Option[A] = Some(a)
    extension[A, B](fa: Option[A])
      override def flatMap(f: A => Option[B]): Option[B] =
        fa flatMap f
```

```
  given[L]: Monad[Either[L, *]] with // requires -Ykind-projector
    override def pure[A](a: A): Either[L, A] = Right(a)
    extension [A, B](fa: Either[L, A])
      override def flatMap(f: A => Either[L, B]): Either[L, B] =
        fa flatMap f
```

# Typeclasses: Using the Monad Instances

```
def compute[F[?]: Monad, A, B](fa: F[A], fb: F[B]): F[(A, B)] =  
  for  
    a <- fa  
    b <- fb  
  yield (a, b)  
  
val l1 = List(1, 2, 3)  
val l2 = List(10, 20, 30)  
val lResult = compute(l1, l2) // List((1,10), (1,20), (1,30), (2,10), (2,20), (2,30), (3,10), (3,20), (3,30))  
  
val o1 = Option(1)  
val o2 = Option(10)  
val oResult = compute(o1, o2) // Some((1,10))  
  
val e1 = Right(1).withLeft[String]  
val e2 = Right(10).withLeft[String]  
val eResult = compute(e1, e2) // Right((1,10))
```

# Opaque Type Aliases<sup>27</sup>

---

<sup>27</sup> <https://dotty.epfl.ch/docs/reference/other-new-features/opaques.html>

# Opaque Type Aliases

- » Opaque types aliases provide type abstractions without any overhead.
- » No Boxing !!!
- » They are defined like type aliases, but prefixed with the keyword *opaque*.
- » They must be defined within the scope of an object, trait or class.
- » The alias definition is visible only within that scope.
- » Outside the scope only the defined alias is visible.
- » Opaque type aliases are compiled away and have no runtime overhead.
- » In Scala 2 one could use Value Classes to avoid boxing. (Many limitations!)

```
object Geometry:
```

```
    opaque type Length = Double
```

```
    opaque type Area = Double
```

```
object Length:
```

```
    def apply(d: Double): Length = d
```

```
    extension (length: Length)
```

```
        def double: Double = length
```

```
object Area:
```

```
    def apply(d: Double): Area = d
```

```
    extension (area: Area)
```

```
        def double: Double = area
```

```
enum Shape:

    case Circle(radius: Length)
    case Rectangle(width: Length, height: Length)

    def area: Area = this match
        case Circle(r) => math.Pi * r * r
        case Rectangle(w, h) => w * h

    def circumference: Length = this match
        case Circle(r) => 2 * math.Pi * r
        case Rectangle(w, h) => 2 * w + 2 * h

end Geometry
```

# Opaque Type Aliases

- » Outside the *object Geometry* only the types *Length* and *Area* are known.
- » These types are not compatible with *Double*.
- » A *Double* value cannot be assigned to a variable of type *Area*.
- » An *Area* value cannot be assigned to a variable of type *Double*.

```
import Geometry._  
import Geometry.Shape._  
  
val circle = Circle(Length(1.0))  
  
// val cArea: Double = circle.area // error: found: Area, required: Double  
val cArea: Area = circle.area  
val cAreaDouble: Double = cArea.double  
  
// val cCircumference: Double = circle.circumference // error: found: Length, required: Double  
val cCircumference: Length = circle.circumference  
val cCircumferenceDouble: Double = cCircumference.double
```

# Context Functions<sup>28</sup>

---

<sup>28</sup><https://dotty.epfl.ch/docs/reference/contextual/context-functions.html>

# Context Functions

- » Context functions are functions with (only) context parameters.
- » Their types are context function types.
- » They are written using `?=>` as the "arrow" sign.

# Context Function Literals

- » Like their types, context function literals are written using `?=>`.
- » They differ from normal function literals:
  - » Their types are context function types.

# Example: *Executable*

```
type Executable[T] = ExecutionContext ?=> T

given ec: ExecutionContext = ExecutionContext.global

def f(x: Int): Executable[Int] = {
    val result: AtomicInteger = AtomicInteger(0)
    def runOnEC(using ec: ExecutionContext): Int =
        ec.execute(() => result.set(x * x)) // execute a Runnable
        Thread.sleep(100L) // just for demo: wait for the Runnable to be executed
        result.get
    runOnEC
}

val res1: Int = f(2)(using ec)    //=> 4 // ExecutionContext passed explicitly
val res2: Int = f(2)              //=> 4 // ExecutionContext resolved implicitly
```

# Example: *PostConditions*

```
object PostConditions:

    opaque type WrappedResult[T] = T

    def result[T](using r: WrappedResult[T]): T = r

    def [T](x: T) ensuring(condition: WrappedResult[T] ?=> Boolean): T =
        assert(condition(using x))
        x

import PostConditions.{ensuring, result}

val sum = List(1, 2, 3).sum.ensuring(result == 6)
```

# Dependent Function Types<sup>29</sup>

---

<sup>29</sup> <https://dotty.epfl.ch/docs/reference/new-types/dependent-function-types.html>

# Dependent Function Types

- » In a dependent method the result type refers to a parameter of the method.
- » Scala 2 already provides dependent methods (but not dependent functions).
- » Dependent methods could not be turned into functions (there was no type that could describe them).

```

trait Entry { type Key; val key: Key }

def extractKey(e: Entry): e.Key = e.key           // a dependent method
val extractor: (e: Entry) => e.Key = extractKey // a dependent function value
//           ||  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ||
//           ||      Dependent      ||
//           ||    Function Type   ||
//           ||  _____||

val intEntry = new Entry { type Key = Int; val key = 42 }
val stringEntry = new Entry { type Key = String; val key = "foo" }

val intKey1 = extractKey(intEntry) // 42
val intKey2 = extractor(intEntry) // 42
val stringKey1 = extractKey(stringEntry) // "foo"
val stringKey2 = extractor(stringEntry) // "foo"

assert(intKey1 == intKey2)
assert(stringKey1 == stringKey2)

```

# Tuples are HLists

# Tuples are HLists

- » Tuples and HList express the same semantic concept.
- » Scala 3 provides Tuple syntax (like Scala 2) and HList syntax to express this concept.
- » Both are completely equivalent.
- » In Scala 2 Tuple members are limited to 22, in Scala 3 unlimited.
- » *shapeless3* must no longer convert between Tuples and HLists.

# Tuples are HLists

```
// Scala 2 + 3: Tuple syntax
```

```
val isb1: (Int, String, Boolean) = (42, "foo", true)
```

```
// Scala 3: HList syntax
```

```
val isb2: Int *: String *: Boolean *: EmptyTuple = 42 *: "foo" *: true *: EmptyTuple
```

```
// HList in Scala 2 with 'shapeless'
```

```
// val isb3: Int :: String :: Boolean :: HNil = 42 :: "foo" :: true :: HNil
```

```
summon[(Int, String, Boolean) := Int *: String *: Boolean *: EmptyTuple] // identical types
```

```
assert(isb1 == isb2) // identical values
```

# Match Types<sup>30</sup>

---

<sup>30</sup><https://dotty.epfl.ch/docs/reference/new-types/match-types.html>

# Match Types

- » Match types are *match* expressions on the type level.
- » The syntax is analogous to *match* expressions on the value level.
- » A match type reduces to one of a number of right hand sides, depending on the scrutinee type.

# Match Types

```
type Elem[X] = X match
  case String => Char
  case Array[t] => t
  case Iterable[t] => t

// proofs
summon[Elem[String]]      =:= Char
summon[Elem[Array[Int]]]   =:= Int
summon[Elem[List[Float]]]  =:= Float
summon[Elem[Nil.type]]     =:= Nothing
```

# Recursive Match Types

- » Match types may be recursive.

```
type LeafElem[X] = X match
  case String => Char
  case Array[t] => LeafElem[t]
  case Iterable[t] => LeafElem[t]
  case AnyVal => X
```

- » Recursive match types may have an upper bound.

```
type Concat[Xs <: Tuple, +Ys <: Tuple] <: Tuple = Xs match
  case EmptyTuple => Ys
  case x *: xs => x *: Concat[xs, Ys]
```

# Export Clauses<sup>31</sup>

---

<sup>31</sup><https://dotty.epfl.ch/docs/reference/other-new-features/export.html>

# Export Clauses aka Export Aliases

- » An export clause syntactically has the same format as an import clause.
- » An export clause defines aliases for selected members of an object.
- » Exported members are accessible from inside the object as well as from outside ...
  - » ... even when the aliased object is private.
- » Export aliases encourage a best practice: Prefer composition over inheritance.
- » They also fill the gap left by deprecated/removed package objects which inherited from some class or trait.
- » A *given* instance can also be exported, if the exported member is also tagged with *given*.

# Export Clauses

```
class A:  
  def a1 = 42  
  def a2 = a1.toString  
  
class B:  
  private val a = new A  
  export a.{a2 => aString} // exports a.a2 aliased to aString  
  
val b = new B  
  
// a.a1 and a.a2 are not directly accessible as a is private in B.  
// The export clause makes a.a2 (aliased to aString) accessible as a member of b.  
val bString = b.aString ensuring (_ == 42.toString)
```

# Explicit Nulls<sup>32</sup>

---

<sup>32</sup><https://dotty.epfl.ch/docs/reference/other-new-features/explicit-nulls.html>

# Explicit Nulls

- » Explicit nulls are an opt-in feature, enabled via the `-Yexplicit-nulls` compiler flag.
- » This modifies the Scala type system, making reference types (anything that extends `AnyRef`) non nullable.
- » Explicit nulls change the type hierarchy, so that `Null` is only a subtype of `Any`, as opposed to every reference type.
- » After erasure, `Null` remains a subtype of all reference types (as forced by the JVM).
- »  $T \mid Null$  expresses nullability. It is the type of a nullable value.

```
// error: found `Null`, but required `String`
val s1: String = null
```

```
// ok
val s2: String | Null = null
```

# Explicit Nulls: Unsoundness

- » There are still instances where an expression has a non-nullable type like String, but its value is null.
- » The unsoundness occurs in corner cases, because uninitialized fields in a class start out as null.

```
class C:  
    val f: String = foo(f)  
    def foo(f2: String|Null): String = if (f2 == null) "field is null" else f2  
  
val c = new C()  
// c.f == "field is null"
```

# Explicit Nulls: Equality Checks

- » Comparison between AnyRef and Null (using `=`, `!=`, `eq` or `ne`) is no longer allowed.
- » `null` can only be compared with `Null`, nullable union ( $T \mid Null$ ), or `Any` type.

```
val x: String = "foo"
val y: String | Null = "foo"

x == null      // error: Values of types String and Null cannot be compared with == or !=
x eq null     // error
"hello" == null // error

y == null      // ok
y == x         // ok

(x: String | Null) == null // ok
(x: Any) == null          // ok
```

# Working with Nulls

- » The extension method `.nn` can "cast away" nullability.

```
val strOrNull: String|Null = "foo"  
val str: String = strOrNull.nn
```

# Java Interop

- » Java reference types (loaded from source or from byte code) are always nullable.
- » E.g. a Java value or method returning *String* is patched to return *String|UncheckedNull*.
- » *UncheckedNull* is an alias for *Null* with 'magic properties' (see [documentation](#)).

*inline*<sup>33</sup>

---

<sup>33</sup> <https://dotty.epfl.ch/docs/reference/metaprogramming/inline.html>

# *inline*

- » Scala 3 introduces a new modifier *inline*, to be used with ...
- » ... methods, *val*'s, parameters, conditionals and match expressions.
- » *val*'s and parameters, expressions must be fixed at compile time to be inlinable.
- » The compiler guarantees inlining or fails to compile.
- » In Scala 2 the `@inline` annotation was a hint to the compiler to inline if possible.
- » Use annotation `@forceInline` when cross-compiling for Scala 2 and Scala 3.
- » `@forceInline` is equivalent to the modifier *inline* in Scala 3 and ignored in Scala 2.
- » For cross-compilation use both annotations: `@forceInline` and `@inline`.

# *inline* Example

```
object Config:  
  inline val logging = false // RHS must be a constant expression (i.e. known at compile time)
```

```
object Logger:  
  inline def log[T](msg: String)(op: => T) =  
    if Config.logging // Config.logging is a constant condition known at compile time.  
      println(s"START: $msg")  
      val result = op  
      println(s"END: $msg; result = $result")  
      result  
    else  
      op
```

- » *inline* method *log* will always be inlined at the point of call.
- » if-then-else with a constant condition will be rewritten to its then- or else-part.

# Recursive Inline Methods

```
inline def power(x: Double, inline n: Int): Double = // for inlining n must be a constant.  
  if n == 0 then 1.0  
  else if n == 1 then x  
  else  
    val y = power(x, n / 2)  
    if n % 2 == 0 then y * y else y * y * x  
  
power(expr, 10)  
// translates to:  
//   val x = expr  
//   val y1 = x * x    // ^2  
//   val y2 = y1 * y1 // ^4  
//   val y3 = y2 * x  // ^5  
//   y3 * y3          // ^10
```

# inline Conditionals

- » If the condition of an if-then-else expression is a constant expression then it simplifies to the selected branch.
- » When prefixing an if-then-else expression with inline the condition has to be a constant expression.
- » This guarantees that the conditional will always simplify.

```
inline def update(delta: Int) =  
  inline if delta >= 0 then increaseBy(delta)  
  else decreaseBy(-delta)
```

A call *update(22)* would rewrite to *increaseBy(22)* as 22 is a compile-time constant.  
If *update* was not called with a constant, this code snippet doesn't compile.

# Inline Matches

- » A match expression in the body of an inline method def may be prefixed by the inline modifier.
- » If there is enough static information to unambiguously take a branch, the expression is reduced to that branch.
- » Otherwise a compile-time error is raised that reports that the match cannot be reduced.

```
inline def g(x: Any) <: Any = inline x match
  case x: String => (x, x) // return type: Tuple2[String, String](x, x)
  case x: Double => x      // return type: Double

val res1: Double = g(1.0d) // Has type 1.0d which is a subtype of Double
val res2: (String, String) = g("test") // Has type (String, String)
```

# Multiversial Equality<sup>34</sup> <sup>35</sup>

---

<sup>34</sup> <https://dotty.epfl.ch/docs/reference/contextual/multiversal-equality.html>

<sup>35</sup> <https://heikoseeberger.rocks/2020/01/07/2020-01-07-dotty-4/>

# Universial Equality

- » In Scala 2 and Scala 3 you can compare values of any two different types with `=` and `!=`.
- » These comparisons are not type safe.
- » The operators internally use `Any#equals`.
- » This is the heritage of `java.lang.Object#equals`

```
final case class Foo()
```

```
Foo() == Foo()           // true
```

```
Foo() == Option(Foo()) // false - but should not compile
```

# Multiversal or Strict Equality (1/6)

- » Scala 3 gives you strict equality as an opt-in feature.
- » Import `scala.language.strictEqual` or add `-language:strictEquality` to `scalacOptions`.
- » Enabling this feature prevents successful compilation for all comparisons ...
- » ... with some exceptions: you can still compare numbers.

## `scala.language.strictEqual`

```
Foo() == Foo()          // does not compile
Foo() == Option(Foo()) // does not compile
```

## Multiversal or Strict Equality (2/6)

- » For the types you want to compare you have to provide an `CanEqual` instance.

```
given CanEqual[Foo, Foo] = CanEqual.derived
```

```
Foo() == Foo()          // compiles; result: true
```

```
Foo() == Option(Foo()) // does not compile, as we want
```

# Multiversal or Strict Equality (3/6)

- » You can also use the *derives* clause when defining your class.

```
final case class Bar() derives CanEqual
// this is equivalent to
// given CanEqual[Bar, Bar] = CanEqual.derived

Bar() == Bar()          // true
Bar() == Option(Bar())  // does not compile

// you can still not compare _Foo_’s with _Bar_’s.
Foo() == Bar()          // does not compile
Bar() == Foo()          // does not compile
```

# Multiversial or Strict Equality (4/6)

- » If you want to compare *Foo*'s with *Bar*'s ...
- » provide two *CanEqual* instances which allow the comparison.

```
given CanEqual[Foo, Bar] = CanEqual.derived
```

```
given CanEqual[Bar, Foo] = CanEqual.derived
```

```
Foo() == Bar()           // compiles; result is false
```

```
Bar() == Foo()           // compiles; result is false
```

# Multiversal or Strict Equality (5/6)

- » The Scala standard library provides bidirectional `CanEqual` instances for several types:
- » Numeric types can be compared with each other and with `java.lang.Number`.
- » `Boolean` can be compared to `Boolean` and to `java.lang.Boolean`.
- » `Char` can be compared to `Char` and to `java.lang.Character`.
- » `Seq[T]` can be compared to `Seq[T]` (or any sub type) if their element types can be compared.
- » `Set[T]` can be compared to `Set[T]` (or any sub type) if their element types can be compared.
- » Any subtype of `AnyRef` can be compared with `Null`.

# Multiversal or Strict Equality (6/6)

```
42 == 42L           // true  
42 == 42.0         // true  
List(1, 2, 3) == Vector(1, 2, 3) // true  
Foo() == null      // false
```

# Typeclass Derivation<sup>36</sup>

---

<sup>36</sup><https://dotty.epfl.ch/docs/reference/contextual/derivation.html>

# Typeclass Derivation

- » In Scala 2 type class derivation wasn't baked into the language.
- » Type class derivation was provided by 3rd party libraries: shapeless, Magnolia, scalaz-derived.
- » These libraries were based on Scala 2 macros.
- » Scala 3 comes with low level mechanics for typeclass derivation,
- » which are provided primarily for library authors.

# Typeclass Derivation (how it works)

- » Type class derivation supports product types (case classes) and sum types (enums, ADTs).
- » The typeclass author provides the typeclass trait and a method *derived* in the typeclass companion object.
- » *derived* - not necessarily but typically - has a parameter of type *Mirror*.
- » *Mirror* provides the meta information of the typeclass, useful to implement *derived*.
- » The typeclass user can easily create an instance of the type class by adding a *derives* clause to a type.
- » Code structure in the subsequent slides
- » For details see the [Dotty documentation](#)

# Typeclass Derivation (code structure 1/3)

```
import scala.deriving._

trait Eq[T]:    // type class 'Eq'
  def eqv(x: T, y: T): Boolean
  extension (x: T)
    def === (y: T): Boolean = eqv(x, y)
    def !=== (y: T): Boolean = !eqv(x, y)

object Eq:      // type class 'Eq' companion
  inline given derived[T](given m: Mirror.Of[T]): Eq[T] =
    // use Mirror for the implementation, not shown here.
    ???
```

# Typeclass Derivation (code structure 2/3)

```
enum Opt[+T] derives Eq:      // Opt derives a given instance of Eq[Opt[T]]  
  case Sm(t: T)  
  case Nn  
  
import Opt._  
  
val eqoi = summon[Eq[Opt[Int]]]      // summon the instance to a val  
  
assert(eqoi.eqv(Sm(23), Sm(23)))    // use the instance explicitly  
assert(!eqoi.eqv(Sm(23), Sm(13)))  
assert(!eqoi.eqv(Sm(23), Nn))
```

# Typeclass Derivation (code structure 3/3)

```
enum Tree[+T] derives Eq:      // Tree derives a given instance of Eq[Tree[T]]  
  case Leaf(elem: T)  
  case Node(left: Tree[T], right: Tree[T])  
  
import Tree._  
  
// summon Eq[Tree[Int]] instance into local scope  
given Eq[Tree[Int]] = summon[Eq[Tree[Int]]]  
  
// check equality of two trees using the given instance of Eq[Tree[Int]] implicitly  
assert(Node(Leaf(1), Node(Leaf(2), Leaf(3))) === Node(Leaf(1), Node(Leaf(2), Leaf(3))))  
assert(Node(Leaf(1), Node(Leaf(2), Leaf(3))) !== Node(Leaf(2), Leaf(3)))
```

# Given By-Name Parameters<sup>37</sup>

---

<sup>37</sup> <https://dotty.epfl.ch/docs/reference/contextual/by-name-context-parameters.html>

# Given By-Name Parameters

- » Implicit parameters can be declared by-name to avoid a divergent inferred expansion.
- » Like a normal by-name parameter the argument for a *given* parameter is evaluated lazily on demand.
- » This feature is available since Scala 2.13 (but with *implicit* by-name parameters).

# Given By-Name Parameters

```
trait Codec[T]:  
    def write(x: T): Unit  
  
given intCodec: Codec[Int]:  
    def write(x: Int): Unit = println(s"x = $x")  
  
given optionCodec[T](using ev: => Codec[T]): Codec[Option[T]] with // given param ev is evaluated lazily  
    def write(xo: Option[T]) = xo match  
        case Some(x) => ev.write(x)      // evaluation of ev occurs only in the Some(x) case.  
        case None =>                   // no evaluation in the None case  
  
val s = summon[Codec[Option[Int]]]  
s.write(Some(33))  
s.write(None)
```

# Improved Implicit Resolution<sup>38</sup>

---

<sup>38</sup> <https://dotty.epfl.ch/docs/reference/changed-features/implicit-resolution.html>

# Improved Implicit Resolution

- » New algorithm which caches implicit results more aggressively for performance.
- » Types of implicit values and result types of implicit methods must be explicitly declared.
- » Nesting is now taken into account when selecting an implicit.
- » Package prefixes no longer contribute to the implicit scope of a type (which was the case in Scala 2).
- » More details and rules in the [Dotty documentation](#)

# Improved Overload Resolution<sup>39</sup>

---

<sup>39</sup> <https://dotty.epfl.ch/docs/reference/changed-features/overload-resolution.html>

# Looking Beyond the First Argument List

- » Overloading resolution now do not only take the first argument list into account when choosing among a set of overloaded alternatives.

```
def f(x: Int)(y: String): Int = 0
def f(x: Int)(y: Int): Int = 0

f(3)("")      // ok, but ambiguous overload error in Scala 2

def g(x: Int)(y: Int)(z: Int): Int = 0
def g(x: Int)(y: Int)(z: String): Int = 0

g(2)(3)(4)    // ok // but ambiguous overload error in Scala 2
g(2)(3)("")   // ok // but ambiguous overload error in Scala 2
```

# Parameter Types of Function Values

- » Improved handling of function values with missing parameter types

```
def f(x: Int)(y: String): Int = 0
def f(x: Int)(y: Int): Int = 0

def h(x: Int, h: Int => Int) = f(x)
def h(x: String, h: String => String) = f(x)

h(40, _ + 2)          // ok // but missing parameter type error in Scala 2
h("a", _.toUpperCase) // ok without -explicit-nulls // but missing parameter type error in Scala 2
h("a", _.toUpperCase.nn) // .nn needed with -explicit-nulls
```

# Parameter Untupling<sup>40</sup>

---

<sup>40</sup><https://dotty.epfl.ch/docs/reference/other-new-features/parameter-untupling.html>

# Parameter Untupling

- » In a mapping (or other) function you pattern match the tuples to dissect them into their parts.
- » Scala 3 can untuple the tuples into a parameter list of elements.
- » So you can omit the keyword *case*.

```
val l1 = List(1, 2, 3)
val l2 = List(10, 20, 30)
val tuples: List[(Int, Int)] = l1 zip l2
```

```
// Scala 2 style mapping function with pattern matching
val sums1 = tuples map { case (x, y) => x + y }
```

```
// Scala 3 style mapping function with untupled parameters
val sums2 = tuples map { (x, y) => x + y }
val sums3 = tuples map { _ + _ }
```

# Other Features

# Dropped Scala 2 Features

- » [Dropped: Limit 22 for Tuples and Functions](#)
- » [Dropped: Procedure Syntax](#)
- » [Dropped: Symbol Literals](#)
- » [Dropped: DelayedInit](#)
- » [Dropped: Auto-Application](#)
- » [Dropped: Early Initializers](#)
- » [Dropped: Existential Types](#)
- » [Dropped: General Type Projection](#)
- » [Dropped: Scala 2 Macros](#)
- » and more ...

# New or Changed Features

- » Open Classes
- » Improved Lazy Vals Initialization
- » Kind Polymorphism
- » Tupled Function
- » Option-less pattern matching
- » Macros: Quotes and Splices
- » and more ...

# Resources

# Links

- » This presentation: code and slides

<https://github.com/hermannhueck/taste-of-dotty>

- » Dotty Documentation

<https://dotty.epfl.ch/docs/>

# Talks

- » Martin Odersky at ScalaSphere Krakow: Revisiting Implicits (published October 2019)  
<https://www.youtube.com/watch?v=h4dS5WRGJtE>
- » Nicolas Stucki at ScalaDays Lausanne: Metaprogramming in Dotty (published July 2019)  
[https://www.youtube.com/watch?v=ZfDS\\_gJyPTc](https://www.youtube.com/watch?v=ZfDS_gJyPTc)
- » Guillaume Martres at ScalaDays Lausanne: Future proofing Scala through TASTY (published July 2019)  
<https://www.youtube.com/watch?v=zQFjC3zLYwo>
- » Guillaume Martres at ScalaWorld GB: Scala 3, Type Inference and You! (published September 2019)  
<https://www.youtube.com/watch?v=lMvOykNQ4zs>

# More Talks

- » Lukas Rytz at ScalaDays Lausanne: How are we going to migrate to Scala 3.0, aka Dotty? (published July 2019)  
<https://www.youtube.com/watch?v=KUl1Ilcf0b8>
- » Sébastien Doeraene at ScalaSphere Krakov: How will TASTy affect the Scala ecosystem, exactly? (published January 2020)  
[https://www.youtube.com/watch?v=\\_N7zNhLdB1Y](https://www.youtube.com/watch?v=_N7zNhLdB1Y)
- » Josh Suereth & James Ward at Devoxx Belgium: What's coming in Scala 3 (published November 2019)  
<https://www.youtube.com/watch?v=Nv-BzYOMiWY>
- » Jamie Thompson at f(by) 2020:  
Taste the difference with Scala 3: Migrating the ecosystem and more (published February 2020)  
<https://www.youtube.com/watch?v=YQmVrUdx8TU>

# Thank You !

# Questions ?

<https://github.com/hermannhueck/taste-of-dotty>