A Taste of Dotty

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

Abstract

This presentation is an introduction to Dotty / Scala 3.

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

For detailed information see the Dotty documentation.

Agenda (1/2)

- Design Goals
- Project Setup
- Top Level *def*'s and *val*'s
- Indentation / Optional Braces
- New Control Syntax
- Main Methods
- Constructors without *new*
- Traits with Parameters
- Enums and ADTs
- Intersection Types
- Union Types

Agenda (2/2)

- Contextual Abstractions
- Implicit Conversions
- Extension Methods
- Givens
- Context Bounds
- Given Imports
- Typeclasses
- Resources

Design Goals[1]

[1] 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

Changes are Fundamental

» *Scala books have to be rewritten.*

» *Scala MOOCs must be rerecorded.*

(Martin Odersky at Scala Days 2019 in Lausanne)

Project Setup[2]

[2] https://dotty.epfl.ch/docs/usage/getting-started.html

IDE Support[3]

- 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.

[3] 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.[4])

[4] https://code.visualstudio.com/docs/setup/mac#_command-line

New *sbt* Project

- create new project: *sbt new lampepfl/dotty.g8*
- (or: *sbt new lampepfl/dotty-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 dottyVersion = "0.20.0-RC1"
// use latest nightly build of dotty
val dottyVersion = dottyLatestNightlyBuild.get

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

project/plugin.sbt

```
// sbt-dotty plugin
addSbtPlugin("ch.epfl.lamp" % "sbt-dotty" % "0.3.4")
```

project/build.properties

```
// change to latest sbt version
sbt.version=1.2.7
```

Top Level *def*'s and *val*'s[5]

[5] 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 could 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.

```scala
// whatever.scala
package tasty.dotty

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[6]

[6] https://dotty.epfl.ch/docs/reference/other-new-features/indentation.html

Indentation / Optional Braces

- Braces are optional.
- Without braces identation becomes significant to delimit a block of code.

with braces:

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

without braces:

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

New Control Syntax[7]

[7] 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
  "negative"
else if x == 0
  "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
do
  yield x + y
```

Main Methods[8]

[8] 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 then
      ""
    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.

Constructors without *new*[9]

[9] 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* is found, the compiler looks for a suitable constructor.

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

Traits with Parameters[10]

[10] https://dotty.epfl.ch/docs/reference/other-new-features/trait-parameters.html

Traits with Parameters

- Traits can have parameters like classes.
- Arguments are evaluated before the trait is initialized.
- They replace early iniitalizers 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[11] [12]

[11] https://dotty.epfl.ch/docs/reference/enums/enums.html

[12] 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.

```
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[13]

[13] https://dotty.epfl.ch/docs/reference/new-types/intersection-types.html

Intersection Types

» Used on types, the & operator creates an intersection type.

» The type *S* & *T* represents values that are of the type *S* and *T* at the same time.

» *S* & *T* has all members of *S* and all members of *T*.

» & is commutative: *S* & *T* is the same type as *T* & *S*.

```
trait Resettable
  def reset(): this.type

trait Growable[T]
  def add(x: T): this.type

type ResetGrowable[T] =
  Resettable & Growable[T]
```

```scala
class MyClass(var x : Int = 0) extends Resettable with 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 testIntersect: Unit =
  val obj = new MyClass(42) // 42
  obj.reset() // 0
  obj.add(10) // 10
  f(obj) // 21
```

Union Types[14]

[14] https://dotty.epfl.ch/docs/reference/new-types/union-types.html

Union Types

- A union type *A* | *B* comprises all values of type *A* and also all values of type *B*.
- Union types are duals of intersection types.
- | is commutative: *A* | *B* is the same type as *B* | *A*.
- Union types will – in the long run – replace compound types: *A with B*
- *with* ist not commutative.

```scala
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 either: Password | UserName =
  if (true) name else password
```

Contextual Abstractions[15]

[15] 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 mis-used 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

- Focus on intent over mechanism
- Implicit conversions are hard to mis-use.
- Concise syntax for extension methods
- New keyword *given*
- *given* instances focus on types instead of terms.
- *given* clauses replace *implicit* parameters.
- *given* imports are distict from regular imports.
- 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.
- Scala 2 implicits remain available in parallel for a long time.

Implicit Conversions[16]

[16] 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 derive *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:

```scala
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[17]

[17] https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html

Extension Methods

- Extension methods are methods that have a parameter clause in front of the defined identifier.
- They translate to methods where the leading parameter section is moved to after the defined identifier.
- They can be invoked both ways:
  method(param) or *param.method*
- They replace implicit classes of Scala 2.

Extension Methods

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

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

val circle = Circle(0, 0, 1)

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

Givens

Context Bounds

Given Imports

Typeclasses

Resources

Links

» This presentation: code and slides
https://github.com/hermannhueck/taste-of-dotty

Talks

» Martin Odersky: Scala 3 is Coming (July 2019)
https://www.youtube.com/watch?v=U2tjcwSag_0