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
- Type Lambdas
- 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

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

without braces:

```scala
// 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

```scala
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)

```scala
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)

```scala
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

```scala
@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.

```scala
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*.

```scala
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.

```scala
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.

```scala
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 $A \mid B$ comprises all values of type A and also all values of type B.
- Union types are duals of intersection types.
- $\mid$ is commutative: $A \mid B$ is the same type as $B \mid A$.
- 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

```scala
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

Givens

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

Givens: *Future* Example 1

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

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

// implicit val ec: ExecutionContext = ExecutionContext.global // Scala 2
given ec: ExecutionContext = ExecutionContext.global

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

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

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

given Instances

- 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.

given Instances for *Ord*

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

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



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

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

given Clauses

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

given Clauses using Symbols

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

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

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

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

Anonymous *given* Clauses (without Symbols)

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

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

def descending[T](given 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])(given Ord[T]) =
  maximum(xs)(given descending)
```

Usages

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

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

max(2, 3) // max of two Ints
max(2, 3)(given 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)(given descending) // maximum element of a List (in desc order)
```

Context Bounds

» These remain nearly unchanged.

» A context bound is syntactic sugar for the last given clause of a method.

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

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

Given Imports

```scala
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 that given instances of A

object C
  import A.{given, _} // import givens and non-givens with a single import

object D
  import A.{given A.TC} // importing by type
```

Type Lambdas

Type Lambdas

- Type Lambdas are new feature in Scala 3.
- Type Lambdas can be expressed in Scala 2 using a weird syntax with existiential types and type projections.
- The *kind-projector* compiler plugin brought a more convenient type lambda syntax to Scala 2.
- Existential types and type projections are dropped from Scala 3.
- Type lambdas remove the need for *kind-projector*.

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
// 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
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: Monad Example

Typeclasses: Monad Trait

- The previous 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)

```scala
trait Functor[F[_]] {
  def [A, B](x: F[A]) map (f: A => B): F[B]
}
trait Monad[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def [A, B](fa: F[A]) flatMap (f: A => F[B]): F[B]
  override def [A, B] (fa: F[A]) map (f: A => B): F[B] =
    flatMap(fa)(f andThen pure)
}
```

Typeclasses: Monad Instances

```
object Monad {

  given Monad[List]
    override def pure[A](a: A): List[A] = List(a)
    override def [A, B](list: List[A]) flatMap (f: A => List[B]): List[B] =
      list flatMap f

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

  given [L]: Monad[[R] =>> Either[L, R]]
    def pure[A](a: A): Either[L, A] = Right(a)
    def [A, B](fa: Either[L, A]) flatMap (f: A => Either[L, B]): Either[L, B] =
      fa flatMap f
}
```

Typeclasses: Using the Monad Instances

```scala
def compute[F[_]: Monad](fInt1: F[Int], fInt2: F[Int]): F[(Int, Int)] =
  for
    i1 <- fInt1
    i2 <- fInt2
  yield (i1, i2)

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

Opaque Type Aliases

- Opaque types aliases provide type abstraction without any overhead.
- No Boxing !!!
- They are defined like normal type aliases, but prefixed with the new keyword *opaque*.
- They must be defined within the scope of an object, trait or class.
- The alias definition is visible only within the scope.
- Outside the scope only the defined alias is visible.
- Opaque type aliases are compiled away and have no runtime overhead.

```scala
object Geometry {
  opaque type Length = Double
  opaque type Area = Double

  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

  object Length { def apply(d: Double): Length = d }
  object Area { def apply(d: Double): Area = d }

  def (length: Length) l2Double: Double = length
  def (area: Area) a2Double: Double = area
}
```

- 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: Area = circle.area
val cAreaDouble: Double = cArea.a2Double

val cCircumference: Length = circle.circumference
val cCircumferenceDouble: Double = cCircumference.l2Double
```

Implicit Function Types

Implicit Function Types

- Implicit functions are functions with (only) implicit parameters.
- Their types are implicit function types with their parameters preceeded with the keyword *given*.

Implicit Function Literals

- Like their types, implicit function literals are also prefixed with *given*.
- They differ from normal function literals in two ways:
  - Their parameters are defined with a given clause.
  - Their types are implicit function types.

Example with *ExecutionContext*

```
type Executable[T] = (given ExecutionContext) => T

given ec: ExecutionContext = ExecutionContext.global

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

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

Example: Postconditions

```scala
object PostConditions {

  opaque type WrappedResult[T] = T

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

  def [T](x: T) ensuring(condition: (given WrappedResult[T]) => Boolean): T =
    assert(condition(given x))
    x
}

import PostConditions.{ensuring, result}

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

Dependent Function Types

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).

```scala
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)
```

Match Types

Tuples are HLists

Tuples are HLists

- Tuples and HList express the same semantic concept.
- Scala 3 provides Tuple syntax and HList syntax to express this concept.
- Both are completely equivalent.
- In Scala 2 the number of Tuple members is limited to 22, in Scala 3 it is unlimited.

```scala
// Scala 2 + 3: Tuple syntax
val isb1: (Int, String, Boolean) = (42, "foo", true)
// Scala 3: HList syntax
val isb2: Int *: String *: Boolean *: Unit = 42 *: "foo" *: true *: ()
// HList in Scala 2 with 'shapeless'
// val isb3: Int :: String :: Boolean :: HNil = 42 :: "foo" :: true :: HNil

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

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

Match Types

- Match types are a *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 a scrutinee type.

```
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 can 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 Unit => Ys
  case x *: xs => x *: Concat[xs, Ys]
```

Export Clauses

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 the 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

```scala
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

Typeclass derivation

inline

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