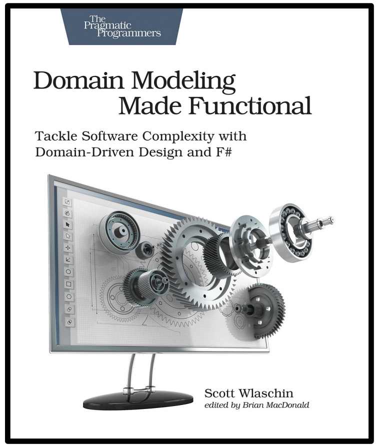
Scala 3 by Example - ADTs for DDD

Algebraic Data Types for **Domain Driven Design**

based on **Scott Wlaschin**'s book

Domain Modeling Made Functional

- Part 1 -



@ScottWlaschin

Martin Odersky **@odersky**

A Tour of Scala 3

Scala 3

slides by  **@philip_schwarz**

slideshare https://www.slideshare.net/pjschwarz

Composition of Types

You'll hear the word "**composition**" used a lot in functional programming—it's the foundation of functional design. **Composition** just means that **you can combine two things to make a bigger thing, like using Lego blocks**.

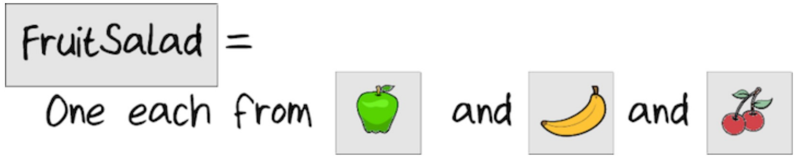
In the functional programming world, **we use composition to build new functions from smaller functions and new types from smaller types**. …

In **F#**, new types are built from smaller types in two ways:

- By _**AND**_ing them together
- By _**OR**_ing them together

"AND" Types

Let's start with building types using ***AND***. For example, we might say that to make fruit salad you need **an apple *and* a banana *and* some cherries**:

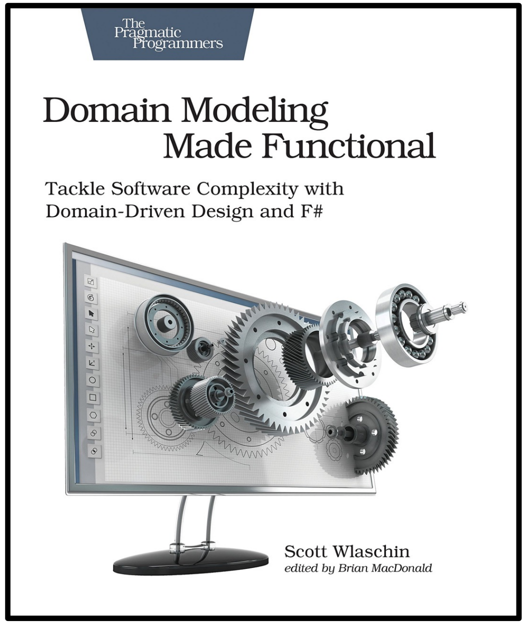


In **F#** this kind of type is called a ***record***. Here's how the definition of a `FruitSalad` record type would be written in **F#**:

```
type FruitSalad   {
  Apple: AppleVariety
  Banana: BananaVariety
  Cherries: CherryVariety
}
```

The curly braces indicate that it is a **record** type, and the three fields are `Apple`, `Banana`, and `Cherries`.

@ScottWlaschin



AND Types

In **Scala** we can define an **AND type** with a **case class**.

@philip_schwarz

F#

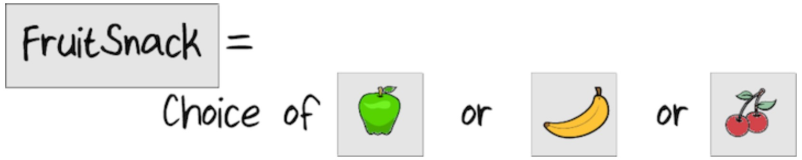
```
type FruitSalad = {
  Apple: AppleVariety
  Banana: BananaVariety
  Cherries: CherryVariety
}
```

Scala

```
case class FruitSalad (
  apple: AppleVariety,
  banana: BananaVariety,
  cherries: CherryVariety
)
```

"OR" Types

The other way of building new types is by using ***OR***. For example, we might say that for a fruit snack you need **an apple *or* a banana *or* some cherries**:



These kinds of "choice" types will be incredibly useful for modeling (as we will see throughout this book). Here is the definition of a **FruitSnack** using a choice type:

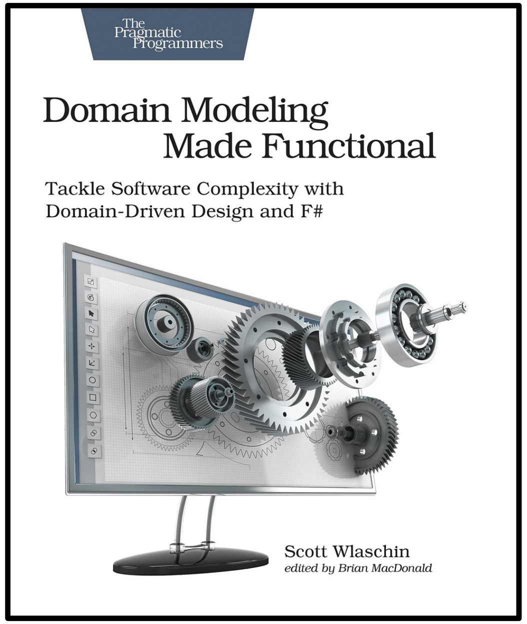
```
type FruitSnack =
  | Apple of AppleVariety
  | Banana of BananaVariety
  | Cherries of CherryVariety
```

A choice type like this is called a ***discriminated union*** in **F#**. It can be read like this:

- A **FruitSnack** is **either** an **AppleVariety** (tagged with **Apple**) ***or*** a **BananaVariety** (tagged with **Banana**) ***or*** a **CherryVariety** (tagged with **Cherries**).

The vertical bar separates each choice, and the tags (such as **Apple** and **Banana**) are needed because sometimes the two or more choices may have the same type and so tags are needed to distinguish them.

@ScottWlaschin



OR Types

```
type FruitSnack =
   | Apple of AppleVariety
   | Banana of BananaVariety
   | Cherries of CherryVariety
```

In Scala we can define an OR type with a sealed trait plus case classes or case objects. See the next slide for an example that uses both. In this case we only need case classes.

```
sealed trait FruitSnack
case class  Apple(variety: AppleVariety)  extends FruitSnack
case class Banana(variety: BananaVariety) extends FruitSnack
case class Cherry(variety: CherryVariety) extends FruitSnack
```

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Defining functional data structures

A **functional data structure** is (not surprisingly) operated on using only pure functions. Remember, a pure function must not change data in place or perform other side effects. Therefore, **functional data structures are by definition immutable**.

…

let's examine what's **probably the most ubiquitous functional data structure, the singly linked list**. The definition here is identical in spirit to (though simpler than) the `List` data type defined in **Scala**'s standard library.

…

Let's look first at **the definition of the data type**, which **begins with the keywords `sealed trait`**.

In general, we introduce a data type with the `trait` keyword.

A **`trait`** is an **abstract interface** that may optionally contain implementations of some methods.

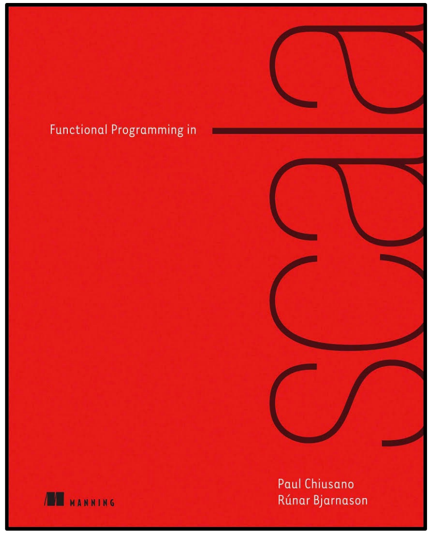
Here we're declaring a **`trait`**, called **`List`**, with **no methods** on it.

Adding **`sealed`** in front means that all implementations of the `trait` must be declared in this file.[1]

There are two such implementations, or **data constructors**, of **`List`** (each introduced with the keyword **`case`**) declared next, to represent the two possible forms a **`List`** can take.

As the figure…shows, a **`List`** can be empty, denoted by the data constructor **`Nil`**, or it can be nonempty, denoted by the data constructor **`Cons`** (traditionally short for construct). A nonempty list consists of an initial element, `head`, followed by a **`List`** (possibly empty) of remaining elements (the `tail`).

[1] We could also say **`abstract class`** here instead of **`trait`**. The distinction between the two is not at all significant for our purposes right now. …



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

OR Types

F#

```
type FruitSnack =
  | Apple of AppleVariety
  | Banana of BananaVariety
  | Cherries of CherryVariety
```

But in **Scala 3** we can also define an **OR type** in a simpler way, using an **enum**.

Scala 2

```
sealed trait FruitSnack
case class  Apple(variety: AppleVariety)  extends FruitSnack
case class Banana(variety: BananaVariety) extends FruitSnack
case class Cherry(variety: CherryVariety) extends FruitSnack
```

Scala 3

```
enum FruitSnack {
  case Apple(appleVariety: AppleVariety)
  case Banana(bananaVariety: BananaVariety)
  case Cherry(cherryVariety: CherryVariety)
}
```

The varieties of fruit are themselves defined as ***OR*** types, which in this case is used **similarly to an enum in other languages**.

```
type AppleVariety =
   | GoldenDelicious
   | GrannySmith
   | Fuji

type BananaVariety =
   | Cavendish
   | GrosMichel
   | Manzano

type CherryVariety =
   | Montmorency
   | Bing
```

This can be read as:

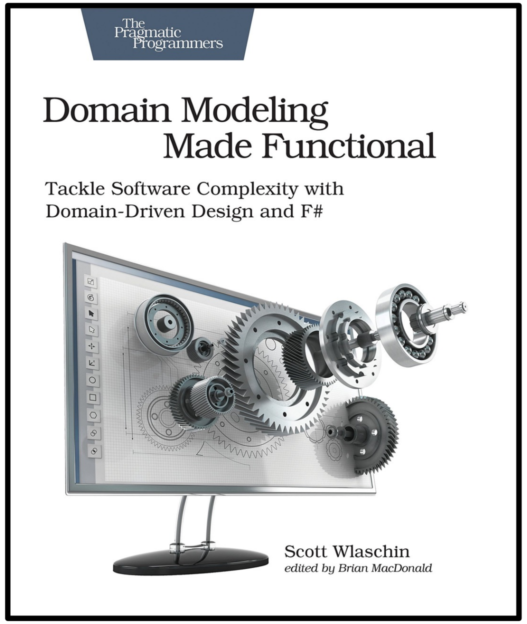
- An `AppleVariety` is **either** a `GoldenDelicious` ***or*** a `GrannySmith` ***or*** a `Fuji`, and so on.

Jargon Alert: "Product Types" and "Sum Types"

The types that are built using ***AND*** are called ***product types***.

The types that are built using ***OR*** are called ***sum types*** or ***tagged unions*** or, in **F#** terminology, ***discriminated unions***. In this book **I will often call them *choice types*, because I think that best describes their role in domain modeling**.

@ScottWlaschin



OR Types

F#

```
type AppleVariety =
  | GoldenDelicious
  | GrannySmith
  | Fuji

type BananaVariety =
  | Cavendish
  | GrosMichel
  | Manzano

type CherryVariety =
  | Montmorency
  | Bing
```

It seems that in **Scala 3** we can also use **enums** to define these basic **OR types**.

@philip_schwarz

Scala 3

```
enum AppleVariety {
  case GoldenDelicious,
       GrannySmith,
       Fuji
}

enum BananaVariety {
   case Cavendish,
        GrosMichel,
        Manzano
}

enum CherryVariety {
   case Montmorency,
        Bing
}
```

Simple Types

We will often define **a choice type with only *one* choice**, such as this:

```
type ProductCode =
  | ProductCode of string
```

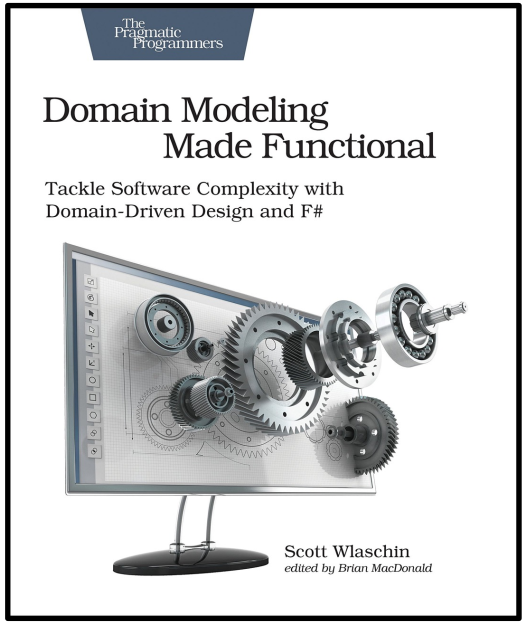
This type is almost always simplified to this:

```
type ProductCode = ProductCode of string
```

Why would we create such a type? Because it's an easy way to create a "wrapper"— a type that contains a primitive (such as a `string` or `int`) **as an inner value**.

We'll be seeing a lot of these kinds of types when we do domain modeling. In this book I will label these **single-case unions** as "**simple types**," as opposed to **compound types** like **records** and **discriminated unions**. More discussion of them is available in the section on Simple Types.

@ScottWlaschin



Simple Types

Let’s have a go at defining **simple types** using Scala 3 **opaque** types.

F#

```
type ProductCode = ProductCode of string
```

Scala 3

```
opaque type ProductCode = String
object ProductCode {
  def apply(code: String): ProductCode = code
}
```

As a recap, here are the **Scala 3 AND types** (**product types**), **OR types** (**sum types**) and **Simple types** for `FruitSnack` and `FruitSalad`.

OR type

AND type

Simple type

```scala
enum AppleVariety {
  case GoldenDelicious, GrannySmith, Fuji
}

enum BananaVariety {
  case Cavendish, GrosMichel, Manzano
}

enum CherryVariety {
  case Montmorency, Bing
}

case class FruitSalad (
  apple: AppleVariety,
  banana: BananaVariety,
  cherries: CherryVariety
)

enum FruitSnack {
  case Apple(variety: AppleVariety)
  case Banana(variety: BananaVariety)
  case Cherry(variety: CherryVariety)
}

object opaqueTypes {
  opaque type ProductCode = String
  object ProductCode {
    def apply(code: String): ProductCode = code
  }
}
```

And on the next slide you can see this code again but without braces and with a very simple example of its usage.

```scala
enum AppleVariety with
  case GoldenDelicious, GrannySmith, Fuji

enum BananaVariety with
  case Cavendish, GrosMichel, Manzano

enum CherryVariety with
  case Montmorency, Bing

case class FruitSalad (
  apple: AppleVariety,
  banana: BananaVariety,
  cherries: CherryVariety
)

enum FruitSnack with
  case Apple(variety: AppleVariety)
  case Banana(variety: BananaVariety)
  case Cherry(variety: CherryVariety)

object opaqueTypes with
  opaque type ProductCode = String
  object ProductCode with
    def apply(code: String): ProductCode = code
```

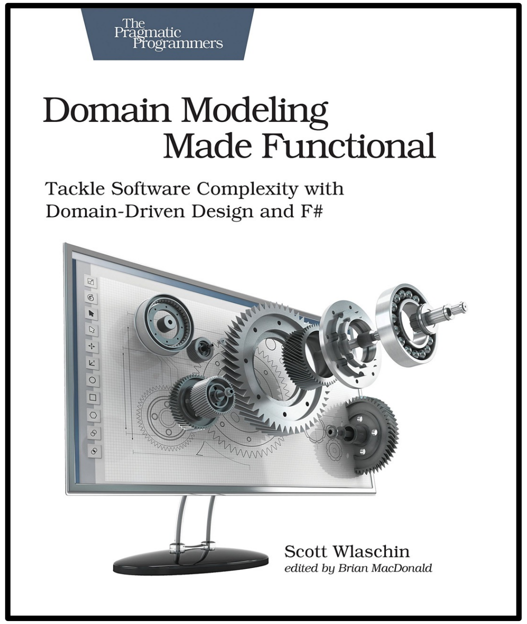
```scala
import AppleVariety._, BananaVariety._, CherryVariety._, opaqueTypes._

@main def main =

  val snack = FruitSnack.Banana(Cavendish)

  val fruitSalad = FruitSalad(
    apple = GoldenDelicious,
    banana = Cavendish,
    cherries = Bing
  )

  assert( snack == FruitSnack.Banana(Cavendish))

  assert( fruitSalad == FruitSalad(GoldenDelicious, Cavendish, Bing) )

  assert( (snack,fruitSalad) match {
    case (FruitSnack.Banana(variety),FruitSalad(_,bananaVariety,_))
      => variety == bananaVariety
    case (_,_)
      => false
  })
```

Algebraic Type Systems

Now we can define what we mean by an "**algebraic type system**." It's not as scary as it sounds—an **algebraic type system** is simply one where **every compound type is composed from smaller types by *AND*-ing or *OR*-ing them together. F#, like most functional languages (but unlike OO languages), has a built-in algebraic type system**.

Using ***AND*** and ***OR*** to build new data types should feel familiar—we used the same kind of ***AND*** and ***OR*** to document our domain. We'll see shortly that **an algebraic type system is indeed an excellent tool for domain modeling**.

@ScottWlaschin



Remember the List functional data structure from **Functional Programming in Scala** (**FPiS**) that we looked at a few slides ago as an example of an OR type implemented using case classes and case objects?

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Let's see how **FPiS** describes algebraic data types.

3.5 Trees

List is just one example of what's called an algebraic data type (ADT). (Somewhat confusingly, ADT is sometimes used elsewhere to stand for **abstract data type**.) An ADT is just a data type defined by one or more data constructors, each of which may contain zero or more arguments. We say that the data type is the sum or union of its data constructors, and each data constructor is the product of its arguments, hence the name algebraic data type.[14]

[14] The naming is not coincidental. There's a deep connection, beyond the scope of this book, between the "addition" and "multiplication" of types to form an ADT and addition and multiplication of numbers.

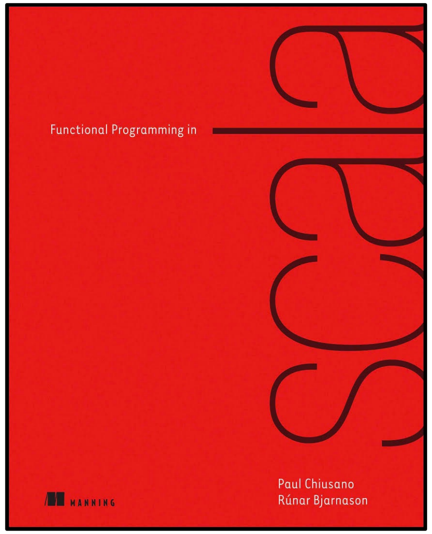
Tuple types in Scala

Pairs and tuples of other arities are also algebraic data types. They work just like the ADTs we've been writing here, but have special syntax…

Algebraic data types can be used to define other data structures. Let's define a simple binary tree data structure:

```
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

…



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Let’s recap (informally) what we just saw in **FPiS**.

@philip_schwarz

- The `List` algebraic data type is the sum of its data constructors, `Nil` and `Cons`.
- The `Nil` constructor has no arguments.
- The `Cons` constructor is the product of its arguments `head: A` and `tail: List[A]`.

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

SUM (the `Nil` and `Cons` cases)

PRODUCT (`head: A, tail: List[A]`)

- The `Tree` algebraic data type is the sum of its data constructors, `Leaf` and `Branch`.
- The `Leaf` constructor has a single argument.
- The `Branch` constructor is the product of its arguments `left: Tree[A]` and `right: Tree[A]`

```
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

SUM (the `Leaf` and `Branch` cases)

PRODUCT (`left: Tree[A], right: Tree[A]`)

Here is how the notions of **sum type** and **product type** apply to **FruitSnack**

Scala 2

```scala
sealed trait FruitSnack
case class  Apple(variety: AppleVariety)  extends FruitSnack
case class Banana(variety: BananaVariety) extends FruitSnack
case class Cherry(variety: CherryVariety) extends FruitSnack
```

SUM (the three case classes)

PRODUCT (the constructor parameters) degenerate product – single argument

Scala 3

```scala
enum FruitSnack with
  case Apple(variety: AppleVariety)
  case Banana(variety: BananaVariety)
  case Cherry(variety: CherryVariety)
```

SUM (the three cases)

PRODUCT (the constructor parameters) degenerate product – single argument

Revisiting a previous diagram to indicate that **each constructor of an OR type can be viewed as a product of its arguments**

```scala
enum AppleVariety {
  case GoldenDelicious, GrannySmith, Fuji
}

enum BananaVariety {
  case Cavendish, GrosMichel, Manzano
}

enum CherryVariety {
  case Montmorency, Bing
}

case class FruitSalad (
  apple: AppleVariety,
  banana: BananaVariety,
  cherries: CherryVariety
)

enum FruitSnack {
  case Apple(variety: AppleVariety)
  case Banana(variety: BananaVariety)
  case Cherry(variety: CherryVariety)
}

object opaqueTypes {
  opaque type ProductCode = String
  object ProductCode {
    def apply(code: String): ProductCode = code
  }
}
```

OR type

AND type

Simple type

products (AND)

degenerate products - single argument

Building a Domain Model by Composing Types

A composable type system is a great aid in doing domain-driven design because we can quickly create a complex model simply by mixing types together in different combinations. For example, say that we want to track payments for an e-commerce site. Let's see how this might be sketched out in code during a design session.

…

So there you go. In about 25 lines of code, we have defined a pretty useful set of types already.
Of course, there is no behavior directly associated with these types because this is a **functional model**, not an **object-oriented model**. To document the actions that can be taken, we instead define types that represent functions.

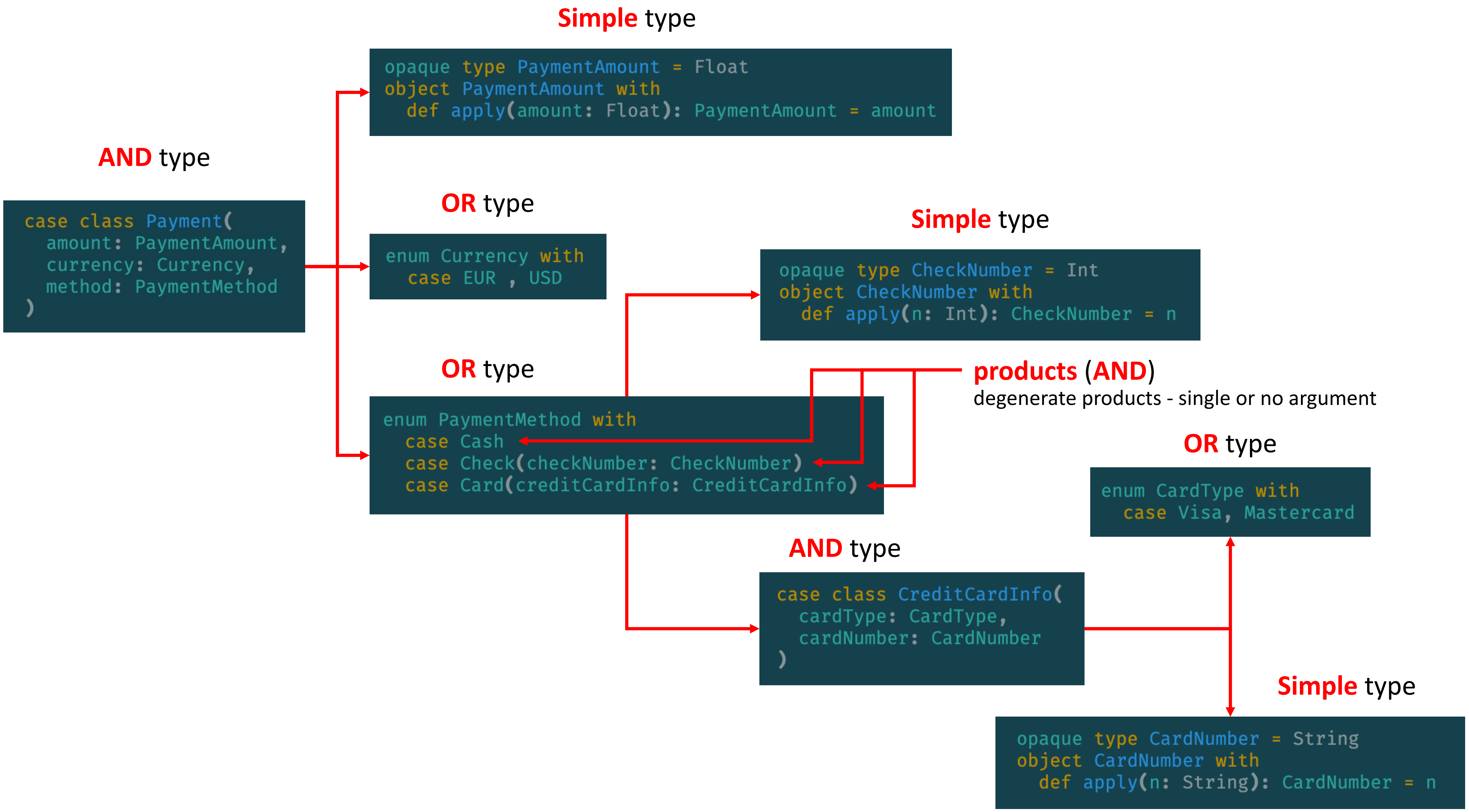
@ScottWlaschin

Here is my translation of **Scott Wlaschin**'s **F#** code into **Scala 3**.

See the next slide for a better illustration of its usage of **AND** types, **OR** types and **Simple** types.

```scala
enum CardType with
  case Visa, Mastercard

enum Currency with
  case EUR , USD

object OpaqueTypes with

  opaque type CheckNumber = Int
  object CheckNumber with
    def apply(n: Int): CheckNumber = n

  opaque type CardNumber = String
  object CardNumber with
    def apply(n: String): CardNumber = n

  opaque type PaymentAmount = Float
  object PaymentAmount with
    def apply(amount: Float): PaymentAmount = amount
```

```scala
import OpaqueTypes._

case class CreditCardInfo(
  cardType: CardType,
  cardNumber: CardNumber
)

enum PaymentMethod with
  case Cash
  case Check(checkNumber: CheckNumber)
  case Card(creditCardInfo: CreditCardInfo)

case class Payment(
  amount: PaymentAmount,
  currency: Currency,
  method: PaymentMethod
)
```



Simple type
opaque type PaymentAmount = Float
object PaymentAmount with
def apply(amount: Float): PaymentAmount = amount
AND type
case class Payment(
amount: PaymentAmount,
currency: Currency,
method: PaymentMethod
)
OR type
enum Currency with
case EUR , USD
Simple type
opaque type CheckNumber = Int
object CheckNumber with
def apply(n: Int): CheckNumber = n
OR type
enum PaymentMethod with
case Cash
case Check(checkNumber: CheckNumber)
case Card(creditCardInfo: CreditCardInfo)
products (AND)
degenerate products - single or no argument
OR type
enum CardType with
case Visa, Mastercard
AND type
case class CreditCardInfo(
cardType: CardType,
cardNumber: CardNumber
)
Simple type
opaque type CardNumber = String
object CardNumber with
def apply(n: String): CardNumber = n

Here is the whole code again plus a very simple example of its usage.

@philip_schwarz

```scala
enum CardType with
  case Visa, Mastercard

enum Currency with
  case EUR , USD

object OpaqueTypes with

  opaque type CheckNumber = Int
  object CheckNumber with
    def apply(n: Int): CheckNumber = n

  opaque type CardNumber = String
  object CardNumber with
    def apply(n: String): CardNumber = n

  opaque type PaymentAmount = Float
  object PaymentAmount with
    def apply(amount: Float): PaymentAmount = amount
```

```scala
import OpaqueTypes._

case class CreditCardInfo(
  cardType: CardType,
  cardNumber: CardNumber
)

enum PaymentMethod with
  case Cash
  case Check(checkNumber: CheckNumber)
  case Card(creditCardInfo: CreditCardInfo)

case class Payment(
  amount: PaymentAmount,
  currency: Currency,
  method: PaymentMethod
)
```

```scala
@main def main =

  val cash10EUR = Payment(
    PaymentAmount(10),
    Currency.EUR,
    PaymentMethod.Cash
  )

  val check10USD = Payment(
    PaymentAmount(350),
    Currency.USD,
    PaymentMethod.Check(CheckNumber(123)))

  println(cash10Eur)
  println(check10Usd)
```

```
Payment(10.0,EUR,Cash)
Payment(350.0,USD,Check(123))
```

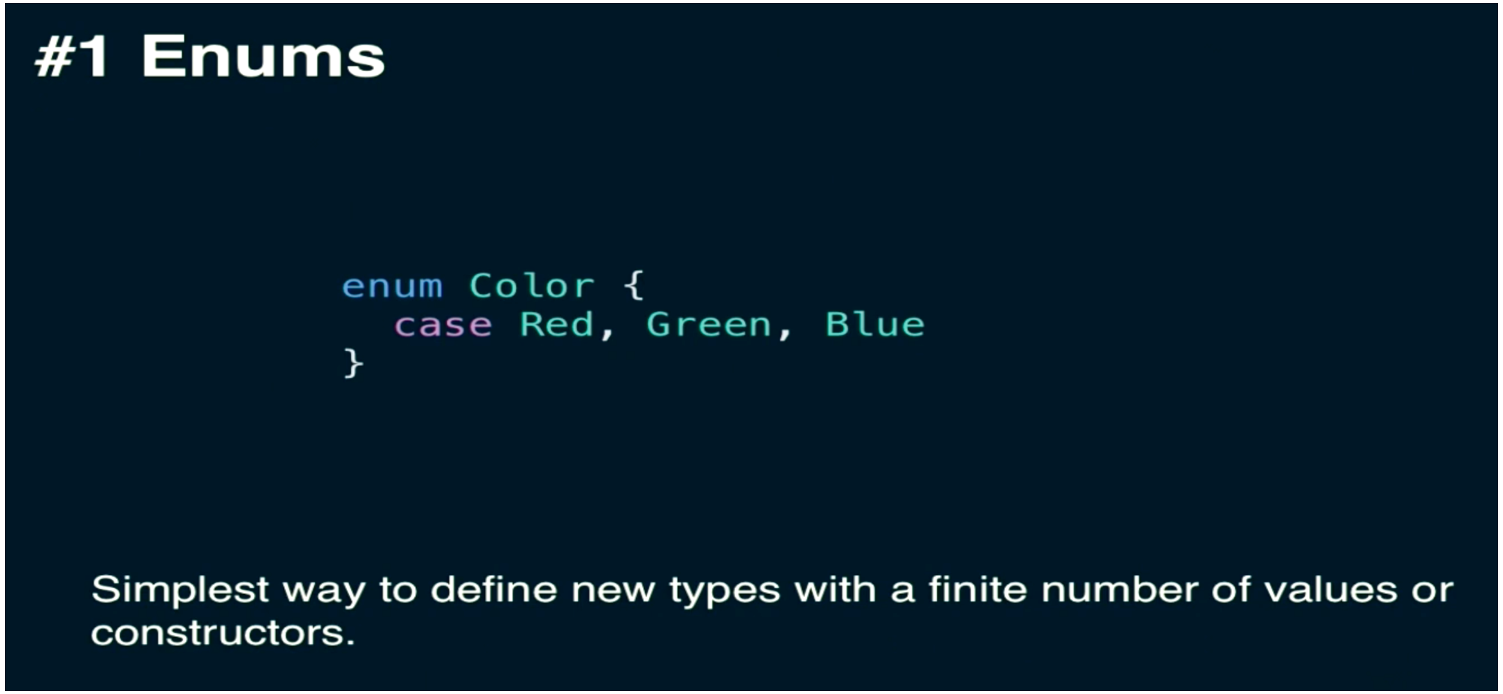
When **Scott Wlaschin** showed us **OR types**, I translated them to **Scala 3 enums**.

For the motivation, let's look at how **Martin Odersky** introduced **enums** in his talk: **A Tour of Scala 3**.

Martin Odersky @odersky

So what is the #1 Scala 3 feature for beginners? Clearly for me **#1** is **enums**. **enums** is such **a nice and simple way to define a new type with a finite number of values or constructors**.

So, **enum** `Color`, **case** `Red`, `Green`, `Blue`, finished: that's all you need.



YouTube A Tour of Scala 3 – by Martin Odersky

Right now, in **Scala**, **it wasn't actually that simple to set up something like that**. There were libraries, there was an **enumeration** type in the standard library, that sort of worked, there was a package called **enumeratum** that also sort of worked, but **it is just much much more straightforward to have this in the language**.

Martin Odersky @odersky

Furthermore, what you have is **not just the simple things, that was the simplest example, but you can actually add everything to it that a Java enum would do**, so you can have **enums** that have parameters, like this one here, you can have **cases** that pass parameters, so here the planets give you the mass and radius, you can have fields, you can have methods in these **enums**. And in fact **you can be fully Java compatible**. That is done here just by extending java.lang.**Enum**. So that's essentially a sign to the compiler that is should generate code so that **this enum is for Java an honest enum that can be used like any other enums**.

#1 Enums

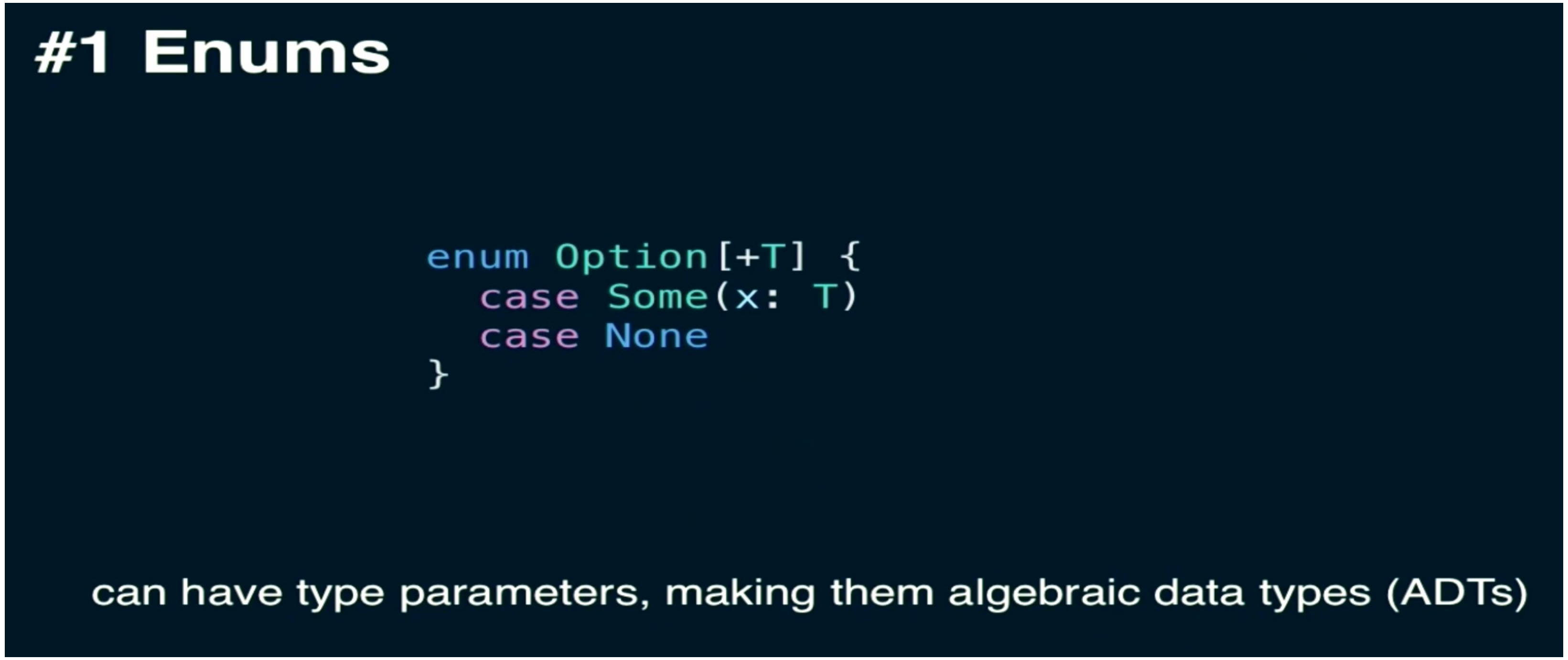
```
enum Planet(mass: Double, radius: Double)
extends java.lang.Enum {
  private final val G = 6.67300E-11
  def surfaceGravity = G * mass / (radius * radius)

  case MERCURY extends Planet(3.303e+23, 2.4397e6)
  case VENUS   extends Planet(4.869e+24, 6.0518e6)
  case EARTH   extends Planet(5.976e+24, 6.37814e6)
  case MARS    extends Planet(6.421e+23, 3.3972e6)
  ...
}
```

can have parameters
can define fields and methods
can interop with Java

YouTube A Tour of Scala 3 – by Martin Odersky

Martin Odersky @odersky

OK, so this is, again, cool, **now we have parity with Java**, **but we can actually go way further**. **enums can not only have value parameters, they also can have type parameters, like this**.



A Tour of Scala 3 – by Martin Odersky

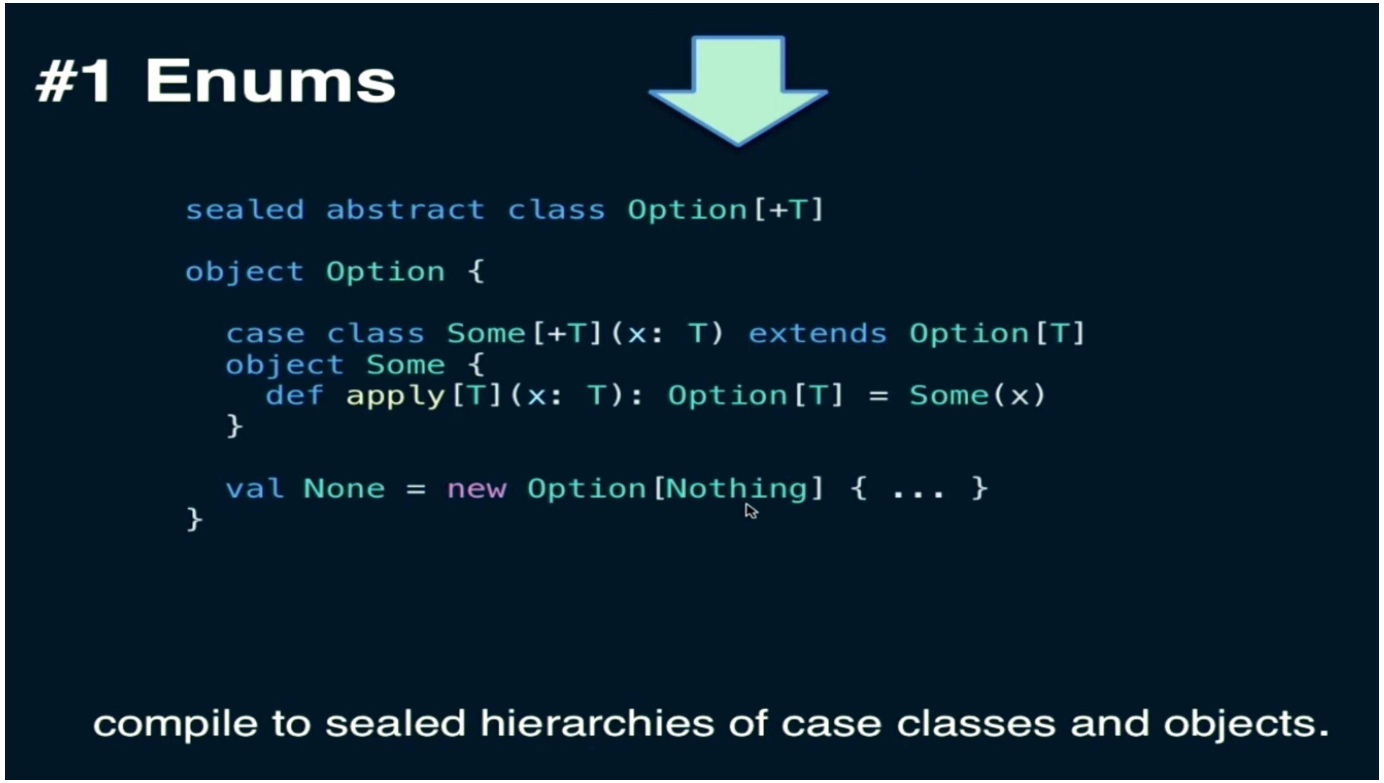
So you can have an **enum** `Option` with a covariant type parameter **T** and then two cases **Some** and **None**.

So that of course gives you what people call an Algebraic Data Type, or ADT.

Scala so far was lacking a simple way to write an ADT. What you had to do is essentially what the compiler would translate this to.

Martin Odersky @odersky

So the compiler would take this **ADT** that you have seen here and translate it into essentially this:



A Tour of Scala 3 – by Martin Odersky

And so far, if you wanted something like that, you would have written essentially the same thing. So **a sealed abstract class or a sealed abstract trait**, **`Option`**, **with a case class as one case, and as the other case, here it is a val, but otherwise you could also use a case object**.

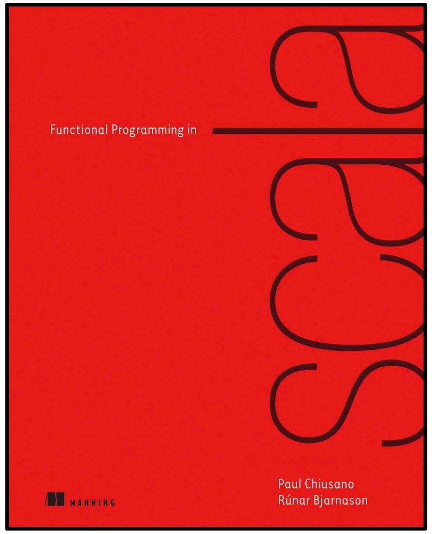
And **that of course is completely workable, but it is kind of tedious**. **When Scala started, one of the main motivations, was to avoid pointless boilerplate**. So that's why **case classes** were invented, and **a lot of other innovations that just made code more pleasant to write and more compact than Java code**, the standard at the time.

```scala
sealed abstract class Option[+T]

object Option {

  case class Some[+T](x: T) extends Option[T]
  object Some {
    def apply[T](x: T): Option[T] = Some(x)
  }

  val None = new Option[Nothing] { ... }
}
```

Martin Odersky @odersky

In **FP in Scala** we can see an example of the alternative that **Martin Odersky** just mentioned, in which the **second case** of the Option **ADT** is a **case object** rather than a **val**.

```scala
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Martin Odersky @odersky

And one has to recognise that during all these years, **the software world has shifted also a little bit and it is now much more functional than before**, so **an ADT would have been something very foreign at the time**, 2003 – 2004, when Scala came out, **but now it is pretty common**.

People write them and write more and more of them because also, with essentially more static typing, **you want to write more and more types, and more and more case hierarchies, and ADTs are just a lovely, simple way to do that**.

So in the spirit of **reducing boilerplate**, it was about time to have something that makes this case simple and straightforward to express.

But that's not even the end of it. We can do more. We can also do Generalized ADTs (GADTs). They are different from normal **ADT**s in that the cases can inherit the base class at different types…

YouTube A Tour of Scala 3 – by Martin Odersky

@philip_schwarz

Hmmm, I have a nagging doubt that some of the **OR types** I have implemented using the **Scala 3 enum** may not be bona fide **ADT**s.

I have used **enums** for two 'kinds' of **OR types**:

```scala
enum AppleVariety with
 case GoldenDelicious,
      GrannySmith,
      Fuji
```

```scala
enum FruitSnack with
  case Apple(variety: AppleVariety)
  case Banana(variety: BananaVariety)
  case Cherry(variety: CherryVariety)
```

In the case of `AppleVariety`, I think I might have got too carried away with the word **enum** when **Scott Wlaschin** said the following:

> The varieties of fruit are themselves defined as ***OR*** types, which in this case is used **similarly to an enum in other languages**.

Maybe a plain **enum** is not technically considered an **ADT**. Maybe I should have defined `AppleVariety` as follows:

```scala
enum AppleVariety with
  case GoldenDelicious
  case GrannySmith
  case Fuji
```

I say that because when explaining **enum**, **Martin Odersky** only started mentioning **ADT**s when he discussed the **`Option`** **enum**. There was no mention of **ADT**s in his first **enum** example, the **Color** **enum**.

```scala
enum Color {
  case Red, Green, Blue
}
```

```scala
sealed abstract class Option[+T]

object Option {

  case class Some[+T](x: T) extends Option[T]
  object Some {
    def apply[T](x: T): Option[T] = Some(x)
  }

  val None = new Option[Nothing] { ... }
}
```

Also, in the following dotty github issue he seems to emphasize a clear distinction between **enumerations**, as found in other languages, and **ADT**s/**GADT**s

Martin Odersky **@odersky**

Add enum construct - 13 Feb 2017 https://github.com/lampepfl/dotty/issues/1970

This is a proposal to add an **enum** construct to **Scala**'s syntax.

The construct is intended to serve at the same time as a native implementation of **enumerations as found in other languages** **and** as a **more concise notation for ADT**s and **GADT**s.

…

So I asked the following on the dotty gitter channel:

Philip @philipschwarz 07:58

is anyone able to explain what the differences (especially in their behaviour) will be between the following two:

```
enum AppleVariety with
 case GoldenDelicious, GrannySmith, Fuji
```

```
enum AppleVariety with
 case GoldenDelicious
 case GrannySmith
 case Fuji
```

Finn Hackett @fhackett 08:04

from what I've read here (https://dotty.epfl.ch/docs/reference/enums/desugarEnums.html, the first 2. bullet point) doesn't the first act as syntax sugar for the second?

and it turns out that the two definitions of `AppleVariety` are equivalent, because in the current dotty documentation pages we find the following desugaring rule:

dotty.epfl.ch/docs/reference/enums/desugarEnums.html

Dotty Documentation
0.22.0-bin-SNAPSHOT

Translation of Enums and ADTs

Edit this page on GitHub

The compiler expands enums and their cases to code that only uses Scala's other language features. As such, enums in Scala are convenient *syntactic sugar*, but they are not essential to understand Scala's core.

2. A simple case consisting of a comma-separated list of enum names

```
case C_1, ..., C_n
```

expands to

```
case C_1; ...; case C_n
```

So one definition is syntactic sugar for the other. Good to know!

What we just saw is consistent with the following

From https://en.wikipedia.org/wiki/Algebraic_data_type

Enumerated types are **a special case of sum types** in which the constructors take no arguments, as exactly one value is defined for each constructor.

From https://bartoszmilewski.com/2015/01/13/simple-algebraic-data-types/

Sum types are pretty common in Haskell, but their C++ equivalents, unions or variants, are much less common. There are several reasons for that.

First of all, **the simplest sum types are just enumerations and are implemented using enum in C++**.

The equivalent of the **Haskell** sum type:

```
data Color = Red | Green | Blue
```

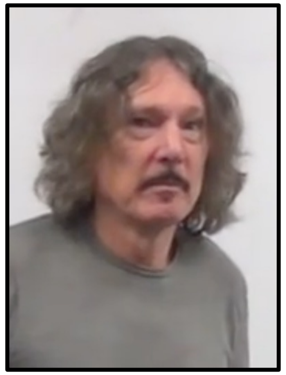
is the **C++:**

```
enum { Red, Green, Blue };
```

An even simpler sum type:

```
data Bool = True | False
```

is the primitive bool in **C++.**



Bartosz Milewski
@BartoszMilewski

@philip_schwarz

That’s all for part 1.

When **Scott Wlaschin** showed us **Simple types**, I translated them to **Scala 3 opaque types**. Why? We’ll look at that in part 2.

Another thing we’ll do in part 2 is take a look at what **Scott** has to say about working with **Simple values** and constraining **Simple values**.

That’s not all we’ll be covering in Part 2 – there will be more.