Scala 3 by Example - ADTs for DDD

Algebraic Data Types for **Domain Driven Design**

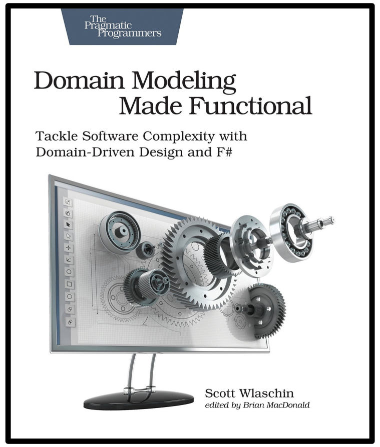
based on **Scott Wlaschin**'s book

Domain Modeling Made Functional

- Part 2 -

Scott Wlaschin

@ScottWlaschin



The Pragmatic Programmers

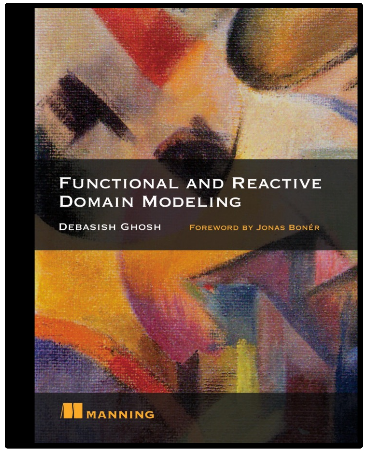
Domain Modeling
Made Functional

Tackle Software Complexity with
Domain-Driven Design and F#

Scott Wlaschin
edited by Brian MacDonald

Debasish Ghosh

@debasishg



FUNCTIONAL AND REACTIVE
DOMAIN MODELING

DEBASISH GHOSH   FOREWORD BY JONAS BONÉR

MANNING

Erik Osheim

@d6

Jorge Vicente Cantero

@jvican

slides by

@philip_schwarz

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

@philip_schwarz

Before we get started, I noticed that since I finished part 1 of this series there has been a change in **Scala 3** that affects the code in both part 1 and part 2.

In **dotty** 0.22 it was possible to replace pairs of braces using the **with** keyword, whereas in **dotty** 0.24 I see that the **with** keyword is no longer supported and seems to have been replaced by a **colon**.

See the next slide for relevant extracts from the documentation of versions 0.22 and 0.24 of **dotty**.

See the slide after that for the original version of the main code from part 1, which eliminates pairs of braces by replacing them with the **with** keyword.

See the slide after that for a new version of the code which instead eliminates pairs of braces by replacing them with a **colon**.

From https://dotty.epfl.ch/docs/reference/changed-features/main-functions.html

Dotty Documentation
0.22.0-bin-SNAPSHOT

Main Methods

Scala 3 offers **a new way to define programs that can be invoked from the command line: A @main annotation on a method turns this method into an executable program**.

From https://dotty.epfl.ch/docs/reference/other-new-features/indentation-new.html

Optional Braces

As an experimental feature, Scala 3 enforces some rules on indentation and **allows some occurrences of braces {...} to be optional**.

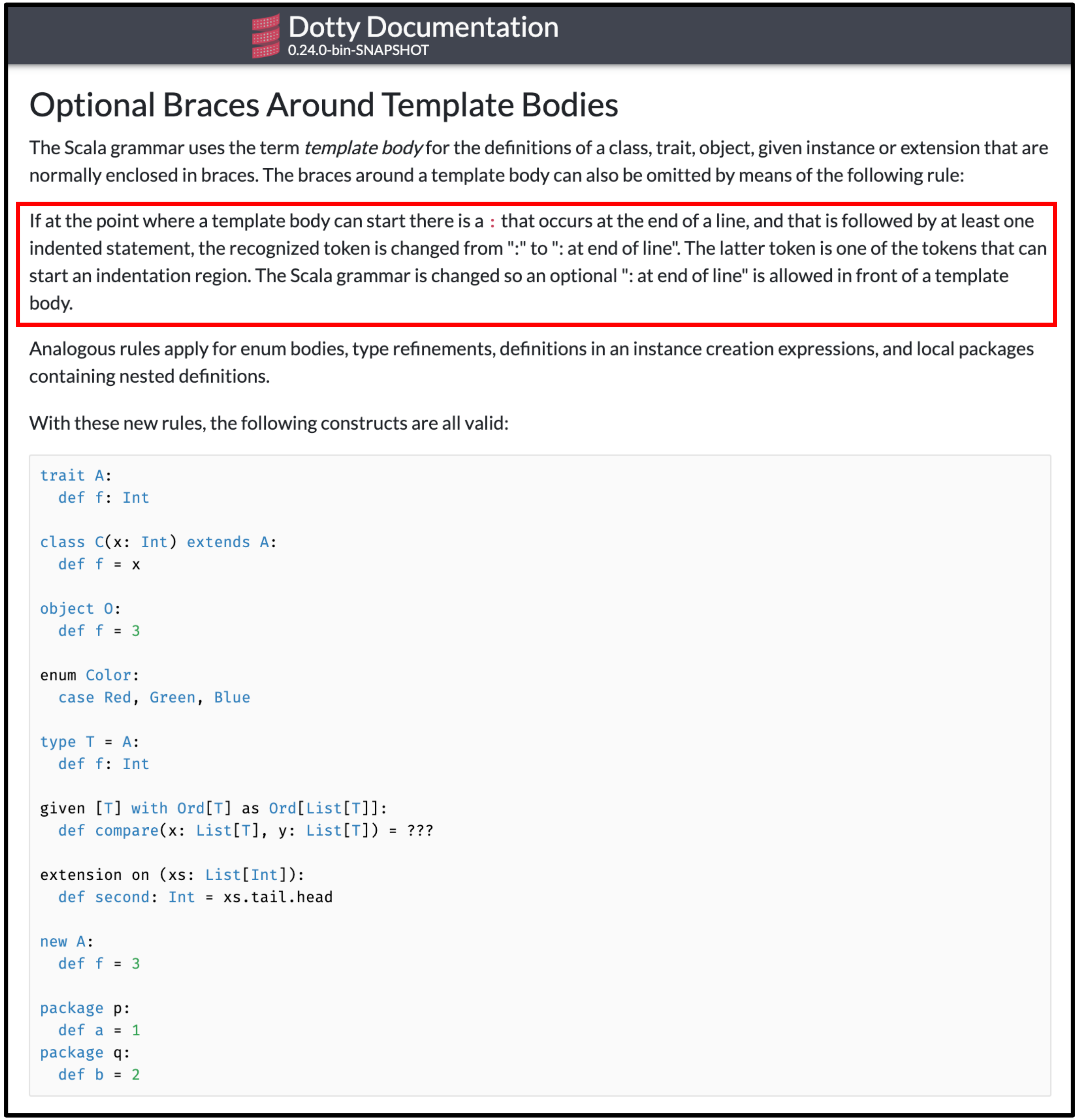
- First, some badly indented programs are ruled out, which means they are flagged with warnings.
- Second, **some occurrences of braces {...} are made optional**. Generally, the rule is that adding a pair of optional braces will not change the meaning of a well-indented program.

…

New Role of With

To make braces optional for constructs like class bodies, the syntax of the language is changed so that **a class body or similar construct may optionally be prefixed with with**.

…



Dotty Documentation
0.24.0-bin-SNAPSHOT

Optional Braces Around Template Bodies

The Scala grammar uses the term *template body* for the definitions of a class, trait, object, given instance or extension that are normally enclosed in braces. The braces around a template body can also be omitted by means of the following rule:

If at the point where a template body can start there is a `:` that occurs at the end of a line, and that is followed by at least one indented statement, the recognized token is changed from ":" to ": at end of line". The latter token is one of the tokens that can start an indentation region. The Scala grammar is changed so an optional ": at end of line" is allowed in front of a template body.

Analogous rules apply for enum bodies, type refinements, definitions in an instance creation expressions, and local packages containing nested definitions.

With these new rules, the following constructs are all valid:

```
trait A:
  def f: Int

class C(x: Int) extends A:
  def f = x

object O:
  def f = 3

enum Color:
  case Red, Green, Blue

type T = A:
  def f: Int

given [T] with Ord[T] as Ord[List[T]]:
  def compare(x: List[T], y: List[T]) = ???

extension on (xs: List[Int]):
  def second: Int = xs.tail.head

new A:
  def f = 3

package p:
  def a = 1
package q:
  def b = 2
```

dotty 0.22 - replacing pairs of braces using the with **keyword**

```
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
```

```
import OpaqueTypes._

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

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

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

```
@main def main =

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

  val check350USD = Payment(
    PaymentAmount(350),
    Currency.USD,
    PaymentMethod.Check(CheckNumber(123)))

  println(cash10EUR)
  println(check350USD)
```

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

dotty 0.24 - replacing pairs of braces using a **colon**

```
enum CardType:
  case Visa, Mastercard

enum Currency:
  case EUR, USD

object OpaqueTypes:

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

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

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

```
import OpaqueTypes._

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

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

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

```
@main def main =

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

  val check350USD = Payment(
    PaymentAmount(350),
    Currency.USD,
    PaymentMethod.Check(CheckNumber(123)))

  println(cash10EUR)
  println(check350USD)
```

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

@philip_schwarz

With that out of the way, let's get started.

In part 1, when **Scott Wlaschin** showed us **Simple types**, I translated them to **Scala 3 opaque types**.

Why? I want to explain the reason for that.

To get us started, on the next slide **Scott Wlaschin** explains why he models **simple values** using **wrapper types**, which he calls **Simple types**.

Part of his explanation acts as a useful reminder of ideas already covered in part 1.

Modeling Simple Values

Let's first look at **the building blocks of a domain**: **simple values**.

As we found out when we gathered the requirements, **a domain expert does not generally think in terms of `int` and `string` but instead in terms of domain concepts such as `OrderId` and `ProductCode`. Furthermore, it's important that `OrderIds` and `ProductCodes` don't get mixed up. Just because they're both represented by `ints`, say, doesn't mean that they are interchangeable**. **So to make it clear that these types are distinct, we'll create a "wrapper type"— a type that wraps the primitive representation**.

As we mentioned earlier, **the easiest way to create a wrapper type in F# is to create** a "**single-case**" **union type**, **a choice type with only one choice**.

Here's an example:

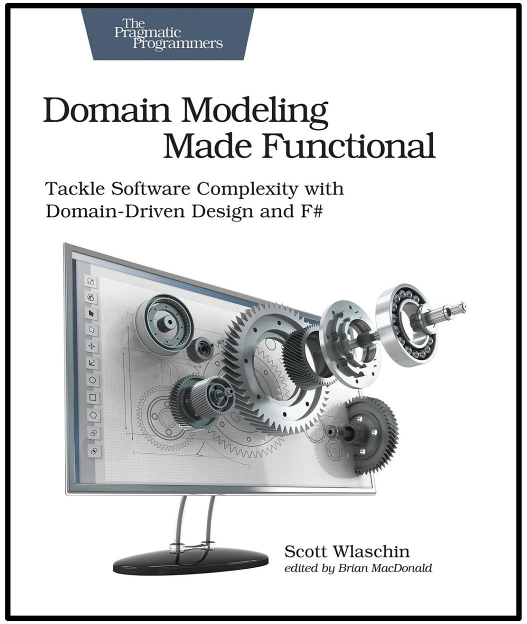
```
type CustomerId =
  | CustomerId of int
```

Since there's only one case, we invariably write the whole type definition on one line, like this:

```
type CustomerId = CustomerId of int
```

We'll call these kinds of wrapper types "simple types" to distinguish them both from compound types (such as records) and the raw primitive types (such as `string` and `int`) that they contain.

@ScottWlaschin



In our domain, the **simple types** would be modeled this way:

```
type WidgetCode = WidgetCode of string
type UnitQuantity = UnitQuantity of int
type KilogramQuantity = KilogramQuantity of decimal
```

The definition of a **single case union** has two parts: the name of the type and the "**case**" label:

```
type CustomerId = CustomerId of int
//   ^type name   ^case label
```

As you can see from the examples above, the label of the (single) case is typically the same as the name of the type. This means that **when using the type, you can also use the same name for constructing and deconstructing it**, as we'll see next.

Working with Single Case Unions

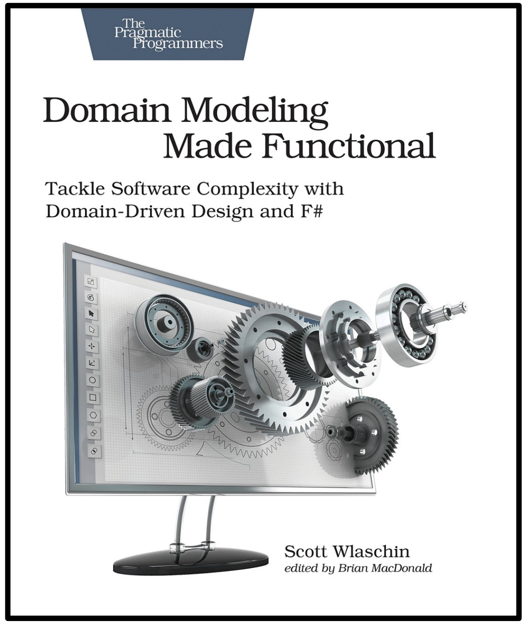
To create a value of a single case union, **we use the case name as a constructor function**. That is, we've defined a simple type like this:

```
type  CustomerId = CustomerId  of   int
//                  ^this case name will be the constructor function
```

Now we can create it by using the **case name** as a constructor function:

```
let  customerId = CustomerId 42
//                 ^this is a function with an int parameter
```

@ScottWlaschin



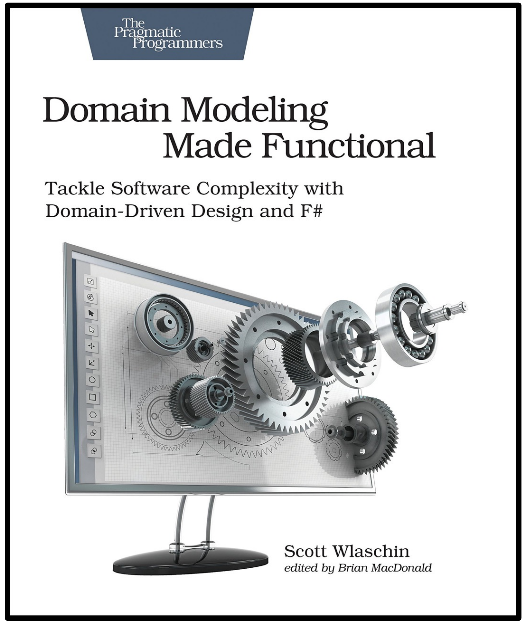
Creating simple types like this ensures that we can’t confuse different types by accident. For example, if we create a `CustomerId` and an `OrderId` and try to compare them, we get a compiler error:

```
// define some types
type  CustomerId = CustomerId of int
type  OrderId = OrderId of int

// define some values
let  customerId = CustomerId 42
let  orderId = OrderId 42

// try to compare them -- compiler error!
printfn "%b" (orderId = customerId)
//                       ^ This expression was expected to
//                       have type 'OrderId’
```

Or if we have defined a function that takes a `CustomerId` as input, then trying to pass it an `OrderId` is another compiler error:

```
// define a function using a CustomerId
let  processCustomerId (id:CustomerId) = ...

// call it with an OrderId -- compiler error!
processCustomerId orderId
//                 ^ This expression was expected to
//                 have type 'CustomerId' but here has
//                 type 'OrderId'
```

@ScottWlaschin



The Pragmatic Programmers

Domain Modeling
Made Functional

Tackle Software Complexity with
Domain-Driven Design and F#

Scott Wlaschin
edited by Brian MacDonald

@philip_schwarz

How should we translate **F#** **Simple types** into **Scala**?

Could it work if we translated **F#** **Simple types** **CustomerId** and **OrderId** to **Scala type aliases**?

It turns out that **it would’t work**. Let’s see why on the next four slides.

Let’s define **CustomerId** and **OrderId** as **type aliases** for **Int**.

```
type CustomerId = Int
type OrderId = Int
```

Now we can define companion objects **CustomerId** and **OrderId** and get them to provide **apply** functions allowing us to use **CustomerId** and **OrderId** as **constructors**.

```
type CustomerId = Int
object CustomerId:
  def apply(id: Int): CustomerId = id

type OrderId = Int
object OrderId:
  def apply(id: Int): OrderId = id
```

And now we can use the **constructors** to create a **CustomerId** and an **OrderId**

```
val customerId = CustomerId(42)
val orderId = OrderId(42)
```

But **there is a problem**: the compiler does not distinguish between a **CustomerId** and an **OrderId**: it treats them both just as **Int** values.

Remember **Scott Wlaschin**’s two sample situations where **we want the compiler to distinguish between `CustomerId` and `OrderId`**?

```
// try to compare them -- compiler error!
prtfn "%b" (orderId = customerId)
//                       ^ This expression was expected to
//                       have type 'OrderId’

// call it with an OrderId -- compiler error!
processCustomerId orderId
//                ^ This expression was expected to
//                have type 'CustomerId' but here has
//                type 'OrderId’
```

Unfortunately the compiler does allows us to compare a **`CustomerId`** and an **`OrderId`**:

```
// we would like this not to compile, but it does
assert( customerId == orderId )
```

Similarly, if we define a function that takes a **`CustomerId`** as a parameter

```
def display(id: CustomerId): Unit =
  println(s"customerId=$id")
```

we are able to invoke the function by passing in an **`OrderId`** as well as by passing in a **`CustomerId`** - **the compiler does not complain** about an **`OrderId`** not being a **`CustomerId`**:

```
// we expect this to compile and of course it does
display(customerId)

// we would like this not to compile, but it does
display(orderId)
```

```
customerId=42


customerId=42
```

The problem is further illustrated by the fact that **the following definitions all compile**!!!

```
val a: CustomerId = OrderId(10)
val b: OrderId = CustomerId(20)
val c: CustomerId = 30
val d: Int = CustomerId(40)
```

Values of the types **`OrderId`**, **`CustomerId`** and **`Int`** appear to be completely interchangeable.

So here on the right is the **type aliases** approach we just tried, which doesn’t work.

What else can we try?

What about using **value classes**?

As a refresher, in the next two slides we look at how **Programming in Scala** introduces **value classes**.

@philip_schwarz

```
type CustomerId = Int
object CustomerId:
  def apply(id: Int): CustomerId = id

type OrderId = Int
object OrderId:
  def apply(id: Int): OrderId = id

val customerId = CustomerId(42)
val orderId = OrderId(42)

// we would like this not to compile, but it does
assert( customerId == orderId )

def display(id: CustomerId): Unit =
  println(s"customerId=$id")

// we expect this to compile and of course it does
display(customerId)

// we would like this not to compile, but it does
display(orderId)
```

The root class `Any` has two subclasses: `AnyVal` and `AnyRef`. `AnyVal` is the parent class of ***value classes*** in **Scala**. While you can define your own **value classes** (see Section 11.4), **there are nine value classes built into Scala**:

`Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, and `Unit`.

The first eight of these correspond to Java's primitive types, and their values are represented at run time as Java's primitive values.

The instances of these classes are all written as literals in **Scala**. For example, `42` is an instance of `Int`, `'x'` is an instance of `Char`, and `false` an instance of `Boolean`. You cannot create instances of these classes using `new`.

…

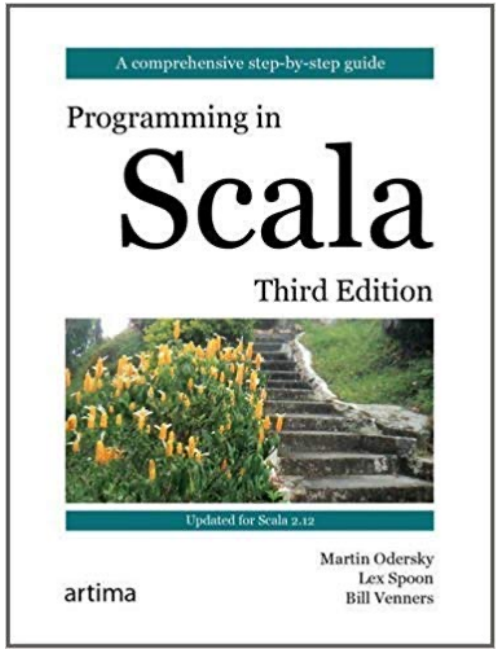
11.4 Defining your own value classes

As mentioned in Section 11.1, **you can define your own value classes to augment the ones that are built in**. **Like the built-in value classes, an instance of your value class will usually compile to Java bytecode that does not use the wrapper class**.

In contexts **where a wrapper is needed**, such as with generic code, **the value will get boxed and unboxed automatically**.

Only certain classes can be made into **value classes**. **For a class to be a value class**, **it must have exactly one parameter and it must have nothing inside it except** `def`s.

Furthermore, **no other class can extend a value class, and a value class cannot redefine** `equals` **or** `hashCode`.



To define a **value class**, make it a subclass of `AnyVal`, and put `val` before the one parameter. Here is an example **value class**:

```
class Dollars(val amount: Int) extends AnyVal {
  override def toString() = "$" + amount
}
```

As described in Section 10.6, the `val` prefix allows the `amount` parameter to be accessed as a field. For example, the following code creates an instance of the **value class**, then retrieves the amount from it:

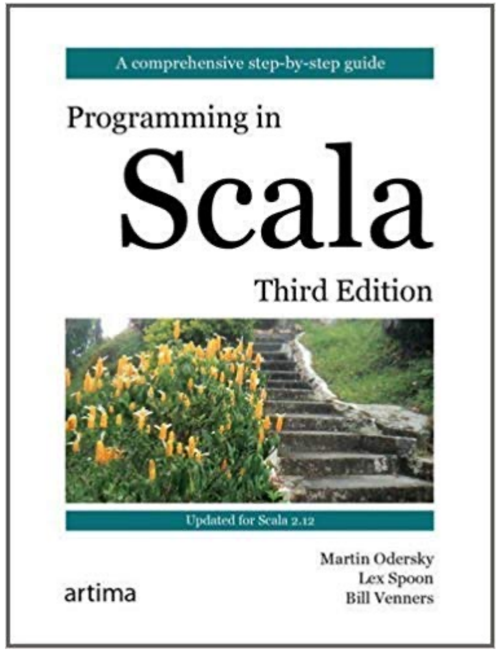
```
scala> val money = new Dollars(1000000)
money: Dollars = $1000000
scala> money.amount
res16: Int = 1000000
```

In this example, `money` refers to an instance of the value class. It is of type `Dollars` in Scala source code, but the compiled Java bytecode will use type `Int` directly. This example defines a `toString` method, and the compiler figures out when to use it. That's why printing `money` gives `$1000000`, with a dollar sign, but printing `money.amount` gives `1000000`. You can even define multiple value types that are all backed by the same `Int` value. For example:

```
class SwissFrancs(val amount: Int) extends AnyVal {
  override def toString() = amount + " CHF"
}
```

Even though **both `Dollars` and `SwissFrancs` are represented as integers**, it works fine to use them in the same scope:

```
scala> val dollars = new Dollars(1000)
dollars: Dollars = $1000
scala> val francs = new SwissFrancs(1000)
francs: SwissFrancs = 1000 CHF
```



A comprehensive step-by-step guide

Programming in

Scala

Third Edition

Updated for Scala 2.12

Martin Odersky
Lex Spoon
Bill Venners

artima

Let’s define **`CustomerId`** and **`OrderId`** as **value classes** wrapping an **`Int`**.

```
class CustomerId(val id: Int) extends AnyVal:
  override def toString() = "CustomerId" + id

class OrderId(val id: Int) extends AnyVal:
  override def toString() = "OrderId" + id
```

In **Scala 2**, we create instances of **`CutomerId`** and **`OrderId`** by newing them up:

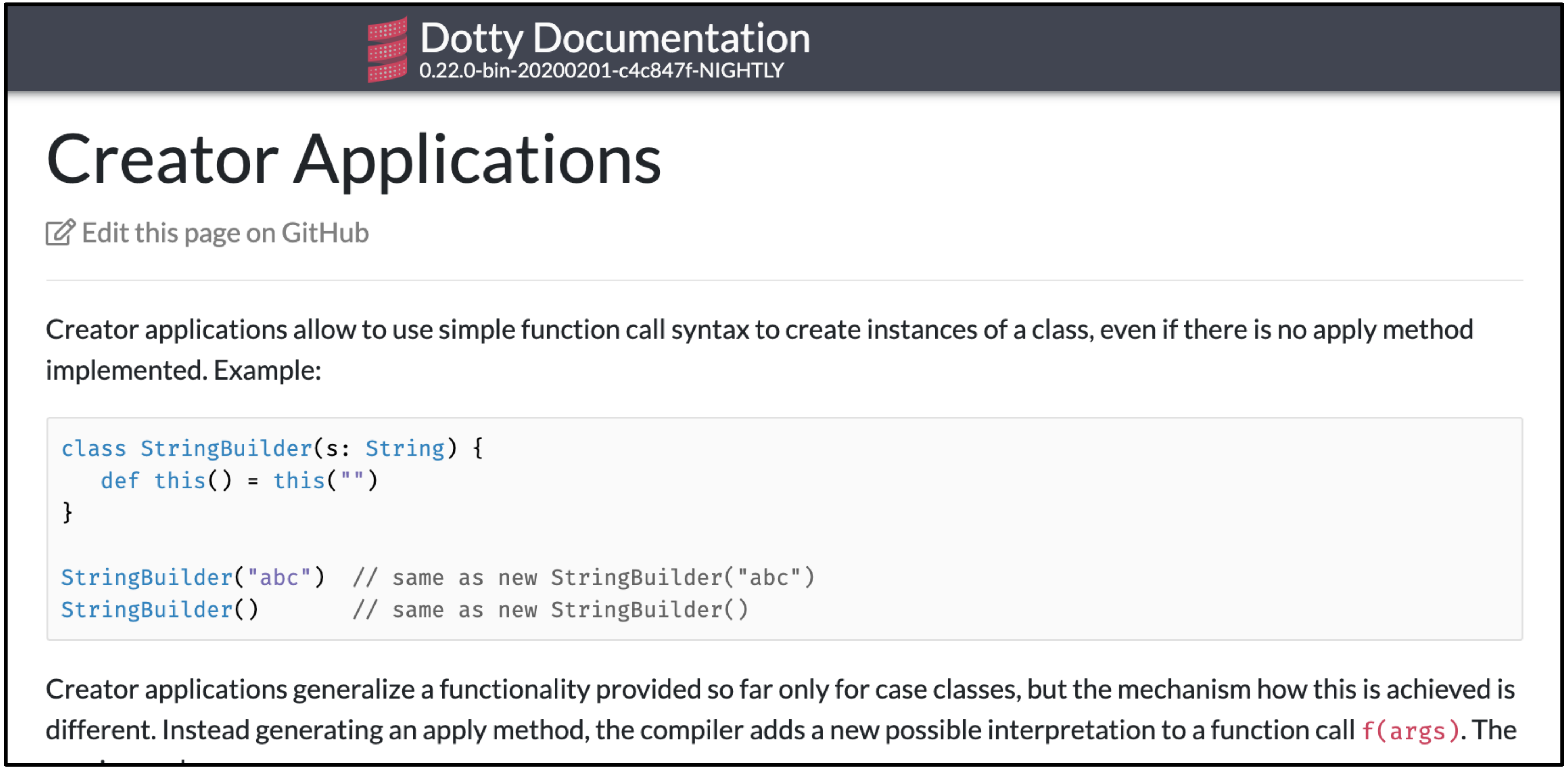
```
val customerId = new CustomerId(42)
val orderId = new OrderId(42)
```

In **Scala 3**, we can drop the ‘new’ keyword:

```
val customerId = CustomerId(42)
val orderId = OrderId(42)
```

See next slide for a brief introduction to why that is possible.

Just for reference, here is why in **Scala 3** we can dispense with the ‘new’ keyword when instantiating classes



Dotty Documentation
0.22.0-bin-20200201-c4c847f-NIGHTLY

Creator Applications

Edit this page on GitHub

Creator applications allow to use simple function call syntax to create instances of a class, even if there is no apply method implemented. Example:

```
class StringBuilder(s: String) {
   def this() = this("")
}

StringBuilder("abc")  // same as new StringBuilder("abc")
StringBuilder()       // same as new StringBuilder()
```

Creator applications generalize a functionality provided so far only for case classes, but the mechanism how this is achieved is different. Instead generating an apply method, the compiler adds a new possible interpretation to a function call `f(args)`. The

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

@philip_schwarz

Translating **Scott**'s **Simple types** to **value classes** works better than translating them to **type aliases**. With **type aliases**, the following assertion compiles and succeeds

```
// we would like this not to compile, but it does
assert( customerId == orderId )
```

With **value classes**, the above assertion fails.

```
java.lang.AssertionError: assertion failed
```

Not only that, but in **Scala 2** the assertion generates the following warning:

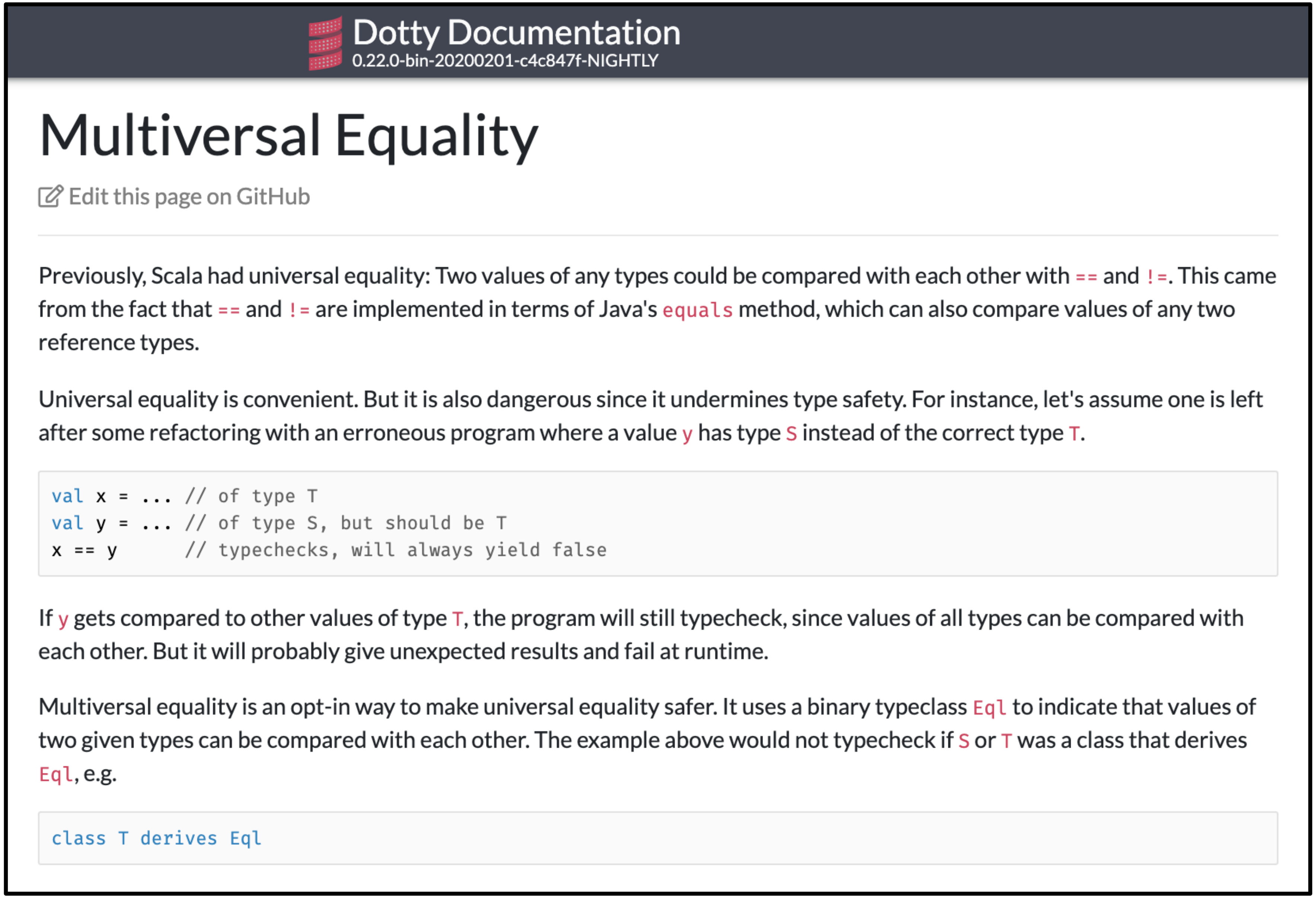
```
<console>:24: warning: CustomerId and OrderId are unrelated: they will never compare equal
         assert( customerId == orderId )
                            ^
```

Interestingly, in Scala 3 the assertion does not generate the warning! The reason the assertion compiles is that the comparison is using **universal equality**, which allows comparisons of two values of any type. **In Scala 3 we can opt into multiversal equality, which makes universal equality safer**. See the next slide for a quick intro to **multiversal equality**. **Here is how in Scala 3 we can make comparison of a CustomerId with a value of any other type illegal (similarly for OrderId):**

```
class CustomerId(val id: Int) extends AnyVal derives Eql:
  override def toString() = "CustomerId" + id

class OrderId(val id: Int) extends AnyVal derives Eql:
  override def toString() = "OrderId" + id
```

```
[error] 14 |  assert( customerId == orderId )
[error]    |          ^^^^^^^^^^^^^^^^^^^^^
[error]    |Values of types CustomerId and OrderId cannot be compared with == or !=
```

Just for reference, here is the very first part of the documentation on **Multiversal Equality**



Dotty Documentation
0.22.0-bin-20200201-c4c847f-NIGHTLY

Multiversal Equality

Edit this page on GitHub

Previously, Scala had universal equality: Two values of any types could be compared with each other with `==` and `!=`. This came from the fact that `==` and `!=` are implemented in terms of Java's `equals` method, which can also compare values of any two reference types.

Universal equality is convenient. But it is also dangerous since it undermines type safety. For instance, let's assume one is left after some refactoring with an erroneous program where a value `y` has type `S` instead of the correct type `T`.

```
val x = ... // of type T
val y = ... // of type S, but should be T
x == y      // typechecks, will always yield false
```

If `y` gets compared to other values of type `T`, the program will still typecheck, since values of all types can be compared with each other. But it will probably give unexpected results and fail at runtime.

Multiversal equality is an opt-in way to make universal equality safer. It uses a binary typeclass `Eql` to indicate that values of two given types can be compared with each other. The example above would not typecheck if `S` or `T` was a class that derives `Eql`, e.g.

```
class T derives Eql
```

https://dotty.epfl.ch/docs/reference/contextual/multiversal-equality-new.html

Using **value classes** also works better than using **type aliases** because **value classes** do ensure that passing an **`OrderId`** to a function that expects a **`CustomerId`** is not allowed - we get a compilation error

```
<console>:14: error: type mismatch;
 found   : OrderId
 required: CustomerId
       display(orderId)
               ^
```

Scala 2

```
[error] 14 |  display(orderId)
[error]    |          ^^^^^^^
[error]    | Found:    (orderId : OrderId)
[error]    | Required: CustomerId
```

Scala 3

So here is the **value classes** approach that we just tried, which works.

```
class CustomerId(val id: Int) extends AnyVal derives Eql:
  override def toString() = "CustomerId" + id

class OrderId(val id: Int) extends AnyVal derives Eql:
  override def toString() = "OrderId" + id

@main def main =

  val customerId = CustomerId(42)
  val orderId = OrderId(42)

  // this does not compile, which is what we want
  assert( customerId == orderId )

  def display(id: CustomerId): Unit =
    println( s"customer id=$id" )

  // we expect this to compile and of course it does
  display(customerId)

  // this does not compile, which is what we want
  display(orderId)
```

```
[error] 13 |  assert( customerId == orderId )
[error]    |          ^^^^^^^^^^^^^^^^^^^^^
[error]    |Values of types CustomerId and OrderId cannot be compared with == or !=
```

```
[error] 22 |  display(orderId)
[error]    |          ^^^^^^^
[error]    | Found:    (orderId : OrderId)
[error]    | Required: CustomerId
```

@philip_schwarz

An easy way to show that the **type aliases** approach was flawed was to show that it allowed the following declarations to compile:

```
val a: CustomerId = OrderId(10)
val b: OrderId = CustomerId(20)
val c: CustomerId = 30
val d: Int = CustomerId(40)
```

Since in the **value classes** approach the `CustomerId`, `OrderId` and `Int` are all distinct classes, it is obvious that those declarations are not going to compile, which is what we want:

```
[error] 27 |    val a: CustomerId = OrderId(10)
[error]    |                        ^^^^^^^^^^^
[error]    |  Found:    OrderId
[error]    |  Required: CustomerId
...
[error] 28 |    val b: OrderId = CustomerId(20)
[error]    |                     ^^^^^^^^^^^^^^
[error]    |  Found:    CustomerId
[error]    |  Required: OrderId
...
[error] 29 |    val c: CustomerId = 30
[error]    |                        ^^
[error]    |  Found:    (30 : Int)
[error]    |  Required: CustomerId
...
[error] 30 |    val d: Int = CustomerId(40)
[error]    |                 ^^^^^^^^^^^^^^
[error]    |  Found:    CustomerId
[error]    |  Required: Int
```

So, while the **type alias approach** to **simple types** doesn't work, the **value classes** approach does work.

What about using **case classes** to wrap our `CustomerId` and `OrderId`?

```scala
case class CustomerId(id: Int) derives Eql
case class OrderId(id: Int) derives Eql

@main def main =

  val customerId = CustomerId(42)
  val orderId = OrderId(42)

  // this does not compile, which is what we wan
  assert( customerId == orderId )

  def display(id: CustomerId): Unit =
    println( s"customer id=$id" )

  // we expect this to compile and of course it does
  display(customerId)

  // this does not compile, which is what we wan
  display(orderId)
```

If we get our **case classes** to derive **Eql**, then this approach also works.

```
[error] 8 |  assert( customerId == orderId )
[error]   |          ^^^^^^^^^^^^^^^^^^^^^
[error]   |Values of types CustomerId and OrderId cannot be compared with == or !=
```

```
[error] 22 |  display(orderId)
[error]    |          ^^^^^^^
[error]    | Found:    (orderId : OrderId)
[error]    | Required: CustomerId
```

In the **case classes** approach, just as in the the **value classes** approach, the **`CustomerId`**, **`OrderId`** and **`Int`** are all distinct classes, so it is just as obvious that the following declarations are not going to compile, which is what we want:

```
[error] 27 |    val a: CustomerId = OrderId(10)
[error]    |                        ^^^^^^^^^^^
[error]    |  Found:    OrderId
[error]    |  Required: CustomerId
...
[error] 28 |    val b: OrderId = CustomerId(20)
[error]    |                     ^^^^^^^^^^^^^^
[error]    |  Found:    CustomerId
[error]    |  Required: OrderId
...
[error] 29 |    val c: CustomerId = 30
[error]    |                        ^^
[error]    |  Found:    (30 : Int)
[error]    |  Required: CustomerId
...
[error] 30 |    val d: Int = CustomerId(40)
[error]    |                 ^^^^^^^^^^^^^^
[error]    |  Found:    CustomerId
[error]    |  Required: Int
```

It turns out that the value class approach and the case class approach are not completely general solutions in that they both suffer from performance issues in some use cases.

It also turns out that opaque types address those performance issues.

In the SIP (Scala Improvement Proposal) for opaque types (SIP 35), there is a motivation section which first explains the problem with type aliases and then mentions the fact that in some use cases there are performance issues with using value classes and case classes to wrap other types.

In the next two slides we look at two sections of the the Opaque Types SIP: the beginning of the motivation section and the introduction section.

SIP-35 - OPAQUE TYPES

Authors: Erik Osheim and Jorge Vicente Cantero

Supervisor and advisor: Sébastien Doeraene

History

Date	Version
Sep 20th 2017	Initial Draft

Erik Osheim
@d6

Jorge Vicente Cantero
@jvican

https://docs.scala-lang.org/sips/opaque-types.html

…

Opaque types

Motivation

Authors often introduce type aliases to differentiate many values that share a very common type (e.g. `String`, `Int`, `Double`, `Boolean`, etc.).

In some cases, these authors may believe that using type aliases such as Id and Password means that if they later mix these values up, the compiler will catch their error.

However, since type aliases are replaced by their underlying type (e.g. `String`), these values are considered interchangeable (i.e. type aliases are not appropriate for differentiating various `String` values).

One appropriate solution to the above problem is to create case classes which wrap `String`. This works, but incurs a runtime overhead (for every `String` in the previous system we also allocate a wrapper, or a “box”). In many cases this is fine but in some it is not.

Value classes, a Scala feature proposed in SIP-15, were introduced to the language to offer classes that could be inlined in some scenarios, thereby removing runtime overhead. These scenarios, while certainly common, do not cover the majority of scenarios that library authors have to deal with. In reality, experimentation shows that they are insufficient, and hence performance-sensitive code suffers (see Appendix A).

Erik Osheim

@d6

Jorge Vicente Cantero

@jvican

https://docs.scala-lang.org/sips/opaque-types.html

Introduction

This is a proposal to introduce syntax for type aliases that only exist at compile time and emulate wrapper types.

The goal is that operations on these wrapper types must not create any extra overhead at runtime while still providing a type safe use at compile time.

Some use cases for opaque types are:

- Implementing type members while retaining parametricity. Currently, concrete type definitions are treated as type aliases, i.e. they are expanded in-place.
- **New numeric classes, such as unsigned integers. There would no longer need to be a boxing overhead for such classes. This is similar to value types in .NET and newtype in Haskell. Many APIs currently use signed integers (to avoid overhead) but document that the values will be treated as unsigned.**
- **Classes representing units of measure. Again, no boxing overhead would be incurred for these classes.**
- **Classes representing different entities with the same underlying type, such as Id and Password being defined in terms of String.**

@philip_schwarz

For our purposes in this slide deck, the decision to implement **Simple types** using **Scala 3 Opaque types** is simply based on the considerations we have just seen in **SIP 35**.

Bear in mind however that **SIP 35** is dated 2017 and in my coverage of **value classes**, **case classes** and **opaque types** I am just **scratching the surface**.

To dispel any doubt about the fact that **I have only taken a simplistic look at the above types**, in the next three slides we look at a laundry lists of some of the many observations that someone with **Erik Osheim**'s level of expertise makes about these types.

I found the observations in the following:

Opaque types: understanding SIP-35 – **Erik Osheim** – Apr 2018
http://plastic-idolatry.com/erik/nescala2018.pdf

I don't expect you to read and understand every bullet point in the next three slides, in fact you can happily just skim through them or even skip the slides and come back to them later if you really want to know more.

It's easiest to compare opaque types with type aliases

Erik Osheim
@d6

Type aliases are **transparent**:
— code can "see through" type aliases in proper types
— authors can inline aliases present in proper types
— aliases do not introduce new types
— are completely erased before runtime
— do not produce classes

Opaque types are... well... **opaque**:
— code cannot see through an opaque type
— authors cannot inline opaque types
— opaque types do introduce new types
— are still completely erased before runtime
— still do not produce classes

Erik Osheim
@d6

Opaque types:
— work well with arrays
— work well with specialization
— avoid an "abstraction penalty"
— are useful for "subsetting" a type
— offer pleasing minimalism

However, **opaque types** also:
— require lots of boilerplate (especially wrappers)
— require a class anyway when doing enrichments
— do not act like traditional classes
— do not eliminate standard primitive boxing
— cannot participate in subtyping

Erik Osheim
@d6

Value classes were introduced in 2.10:
— defined with extends AnyVal
— very specific class requirements
— can only extend universal traits
— avoids allocating objects in some cases
— intended to support zero-cost enrichment
— class still exists at runtime

Value classes have capabilities opaque types lack:
— able to define methods
— can be distinguished from underlying type at runtime
— can participate in subtyping relationships
— can override .toString and other methods

However, **value classes** have some down sides too:
— unpredictable boxing
— constructor/accessor available by default
— cannot take advantage of specialization
— always allocates when used with arrays
— always allocates when used in a generic context

By contrast, **opaque types** are always erased.

Value classes are best used:
— to provide low-cost enrichment
— in cases where traditional wrappers are needed
— in direct contexts (e.g. fields/transient values)

(In other cases, value classes may be more marginal.)

We have seen how implementing **Simple types** with **type aliases** doesn’t work and we have shown how implementing them with **value classes** and **case classes** does work but with **potential performance issues in some use cases**.

That is why I decided to implement **Simple types** using **Scala 3 Opaque types**.

We still need to show that implementing **Simple types** using **opaque types** works.

Let’s do that before we move on.

```scala
object OpaqueTypes:

  opaque type CustomerId = Int
  object CustomerId:
    def apply(id: Int): CustomerId = id
  given Eql[CustomerId, CustomerId] = Eql.derived

  opaque type OrderId = Int
  object OrderId:
    def apply(id: Int): OrderId = id
  given Eql[OrderId, OrderId] = Eql.derived

import OpaqueTypes._

@main def main =

  val customerId = CustomerId(42)
  val orderId = OrderId(42)

  // this does not compile, which is what we want
  assert( customerId == orderId )

  def display(id: CustomerId): Unit =
    println( s"customer id=$id" )

  // we expect this to compile and of course it does
  display(customerId)

  // this does not compile, which is what we want
  display(orderId)
```

```
[error] 24 |  assert( customerId == orderId )
[error]    |          ^^^^^^^^^^^^^^^^^^^^^
[error]    |Values of types CustomerId and OrderId cannot be compared with == or !=
```

```
[error] 33 |  display(orderId)
[error]    |          ^^^^^^^
[error]    |Found:    (orderId : OrderId))
[error]    |Required: CustomerId
```

The code on the left **satisfies our requirements** in that the **assert call** and the **last line** both fail to compile.

In the case of **value classes** and **case classes**, the way we got the assertion not to compile was by attaching `derives Eql` to the class declarations to disallow usage of `==` and `!=` with `CustomerId` and `OrderId` unless both arguments are of the same type.

In the case of **opaque types**, there is no class to wich to attach `derives Eql`, so we achieve the same effect by defining the following:

```scala
given Eql[CustomerId, CustomerId] = Eql.derived
given Eql[OrderId, OrderId] = Eql.derived
```

Furthermore, unlike in the **type aliases** approach, in the **opaque type** approach the following declarations do not compile, which is what we want:

```
val a: CustomerId = OrderId(10)
val b: OrderId = CustomerId(20)
val c: CustomerId = 30
val d: Int = CustomerId(40)
```

Here are the compilation errors the declarations cause:

```
[error] 27 |    val a: CustomerId = OrderId(10)
[error]    |                        ^^^^^^^^^^^
[error]    |  Found:    OrderId
[error]    |  Required: CustomerId
...
[error] 28 |    val b: OrderId = CustomerId(20)
[error]    |                     ^^^^^^^^^^^^^^
[error]    |  Found:    CustomerId
[error]    |  Required: OrderId
...
[error] 29 |    val c: CustomerId = 30
[error]    |                        ^^
[error]    |  Found:    (30 : Int)
[error]    |  Required: CustomerId
...
[error] 30 |    val d: Int = CustomerId(40)
[error]    |                 ^^^^^^^^^^^^^^
[error]    |  Found:    CustomerId
[error]    |  Required: Int
```

@philip_schwarz

Before we move on, I just wanted to mention that in his book, **Scott Wlaschin** also looks at the **performance problems** with **Simple types** and the problem with **type aliases**. Let's see an extract in the next slide.

Avoiding Performance Issues with Simple Types

Wrapping primitive types into simple types is a great way to ensure type-safety and prevent many errors at compile time. However, it does come at a cost in memory usage and efficiency. For typical business applications a small decrease in performance shouldn't be a problem, but for domains that require high performance, such as scientific or real-time domains, you might want to be more careful. For example, looping over a large array of `UnitQuantity` values will be slower than looping over an array of raw `int`s.

But there are a couple of ways you can have your cake and eat it too.

First, **you can use type aliases instead of simple types to document the domain. This has no overhead, but it does mean a loss of type-safety**.

```
type  UnitQuantity =  int
```

Next, as of F# 4.1, you can use a **value type** (a struct) rather than a **reference type**. You'll still have **overhead from the wrapper**, but when you store them in arrays the memory usage will be contiguous and thus more cache-friendly.

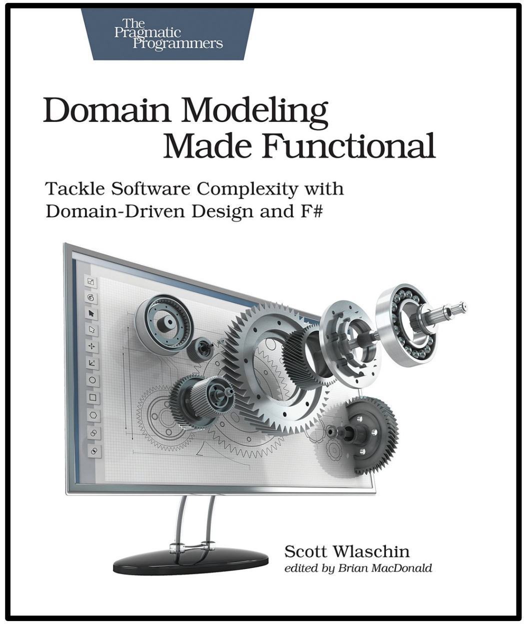
```
type  UnitQuantity = UnitQuantity  of   int
```

Finally, if you are working with large arrays, consider defining the entire collection of primitive values as a single type rather than having a collection of simple types:

```
type  UnitQuantities = UnitQuantities  of   int []
```

This will give you the best of both worlds. You can work efficiently with the raw data (such as for matrix multiplication) while preserving **type-safety** at a high level…

@ScottWlaschin



OK. Now that I have explained why I have translated **Scott**'s implementation of **Simple types** in **F#** to **opaque types** in **Scala 3**, let's look at what **Scott** has to say about constraining **Simple values**.

The Integrity of Simple Values

In the earlier discussion on modeling **simple values**, we saw that they should not be represented by `string` or `int` but by **domain-focused types** such as `WidgetCode` or `UnitQuantity`.

But we shouldn't stop there, because it's very rare to have an unbounded `integer` or `string` in a real-world domain. Almost always, these values are constrained in some way:

- An `OrderQuantity` might be represented by a signed `integer`, but it's very unlikely that the business wants it to be negative, or four billion.
- A `CustomerName` may be represented by a `string`, but that doesn't mean that it should contain tab characters or line feeds.

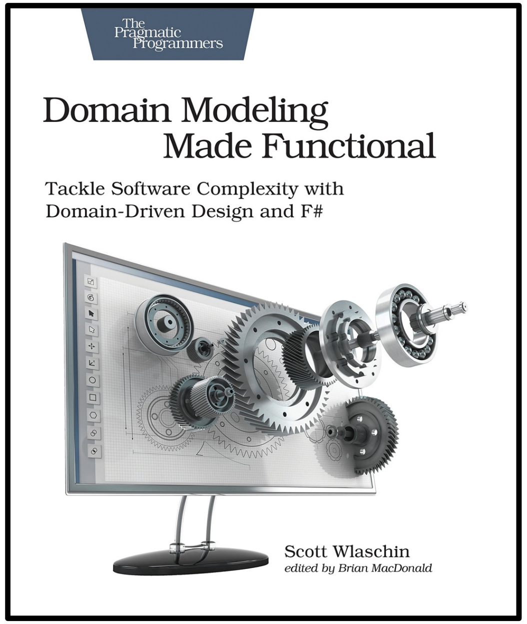
In our domain, we've seen some of these **constrained types** already. `WidgetCode` strings had to start with a specific letter, and `UnitQuantity` had to be between 1 and 1000. **Here's how we've defined them so far, with a comment for the constraint**.

```
type WidgetCode = WidgetCode of string // starting with "W" then 4 digits
type UnitQuantity = UnitQuantity of int // between 1 and 1000
type KilogramQuantity = KilogramQuantity of decimal // between 0.05 and 100.00
```

Rather than having the user of these types read the comments, we want to ensure that values of these types cannot be created unless they satisfy the constraints. Thereafter, because the data is immutable, the inner value never needs to be checked again. You can confidently use a `WidgetCode` or a `UnitQuantity` everywhere without ever needing to do any kind of defensive coding.

Sounds great. So **how do we ensure that the constraints are enforced**?

@ScottWlaschin



Answer: The same way we would in any programming language—**make the constructor private and have a separate function that creates valid values and rejects invalid values, returning an error instead. In FP communities, this is sometimes called the *smart constructor* approach**. Here's an example of this approach applied to `UnitQuantity`:

```
type UnitQuantity = private UnitQuantity of int
```

So now a `UnitQuantity` value can't be created from outside the containing module due to the `private` constructor. However, if we write code in the same module that contains the type definition above, then we *can* access the constructor.

Let's use this fact to define some functions that will help us manipulate the type. We'll start by creating a submodule with exactly the same name (`UnitQuantity`); and within that, we'll define **a create function that accepts an `int`** and returns a `Result` type (as discussed in *Modeling Errors*) to return a **success** or a **failure**. These two possibilities are made explicit in its function signature: `int -> Result<UnitQuantity,string>`.

```
// define a module with the same name as the type
module UnitQuantity =
  /// Define a "smart constructor" for UnitQuantity
  /// int -> Result<UnitQuantity,string>
  let create qty =
    if  qty < 1  then
      // failure
      Error  "UnitQuantity can not be negative"
    else if  qty > 1000  then
      // failure
      Error "UnitQuantity can not be more than 1000"
    else
      // success -- construct the return value
      Ok (UnitQuantity qty)
```

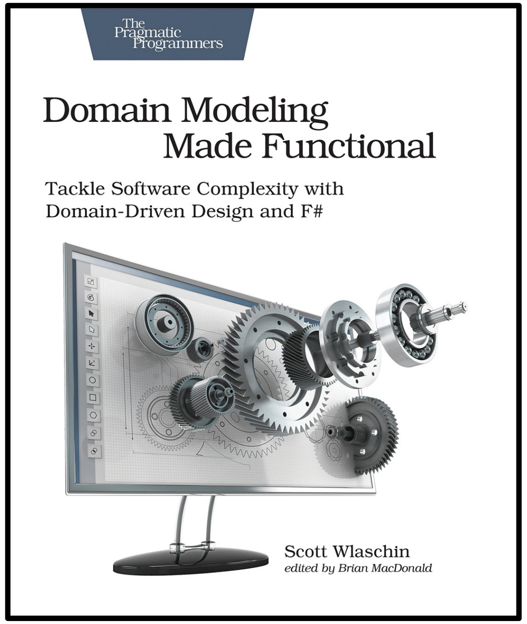
F#

```
------------------------------------------
Result<UnitQuantity,string>
Error("error message")
Ok(…xyz…)
```

Scala

```
------------------------------------------
Either[String,UnitQuantity]
Left("error message")
Right(…xyz…)
```

@ScottWlaschin



The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

Scott Wlaschin

edited by Brian MacDonald

Here is that **F#** code again, together with the **Scala 3** equivalent

```
type UnitQuantity = private UnitQuantity of int

module UnitQuantity =

  let create qty =
    if qty < 1  then
      Error  "UnitQuantity can not be negative"
    else if qty > 1000  then
      Error "UnitQuantity can not be more than 1000"
    else
      Ok (UnitQuantity qty)
```

```
opaque type UnitQuantity = Int

object UnitQuantity:

  def create(qty: Int): Either[String, UnitQuantity] =
    if qty < 1
      Left(s”UnitQuantity can not be negative: $qty ")
    else if qty > 1000
      Left(s" UnitQuantity can not be more than 1000: $qty ")
    else
      Right(qty)
```

One downside of a **private constructor** is that **you can no longer use it to pattern-match and extract the wrapped data**. One workaround for this is to define a separate `value` function, also in the `UnitQuantity` module, that extracts the inner value.

```
/// Return the wrapped value
let  value (UnitQuantity qty) = qty
```

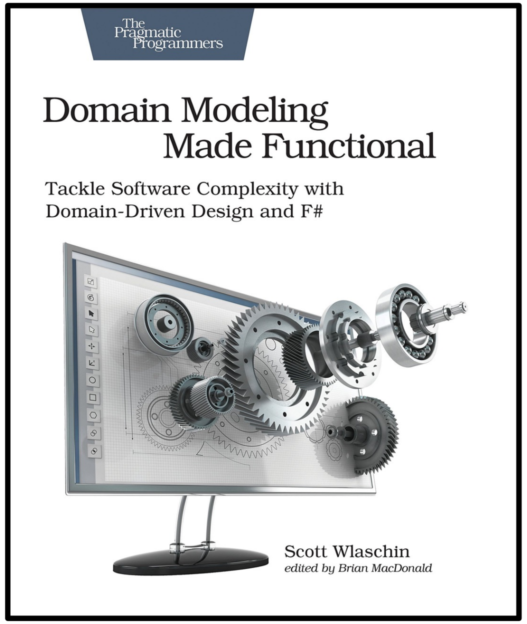
Let's see how this all works in practice. First, if we try to create a `UnitQuantity` directly, we get a compiler error:

```
let  unitQty = UnitQuantity 1
//            ^ The union cases of the type 'UnitQuantity'
//              are not accessible
```

But if we use the `UnitQuantity.create` function instead, it works and we get back a `Result`, which we can then match against:

```
let  unitQtyResult = UnitQuantity.create 1

match  unitQtyResult  with
| Error msg ->
  printfn  "Failure, Message is %s"  msg
| Ok uQty ->
  printfn  "Success. Value is %A"  uQty
  let  innerValue = UnitQuantity.value uQty
  printfn  "innerValue is %i"  innerValue
```

@ScottWlaschin



@philip_schwarz

Here is that **F#** code again, together with the **Scala 3** equivalent

```
/// Return the wrapped value
let  value (UnitQuantity qty) = qty
```

we don’t need this function because using an **opaque type** means there is no problem accessing the value.

```
 let  unitQty = UnitQuantity 1
 //            ^ The union cases of the type 'UnitQuantity’
 //              are not accessible
```

```
val unitQty = UnitQuantity(3)
//            ^ too many arguments for constructor Int: (): Int
```

The reason why we get an error is not that we have made a constructor private, but rather that there is no such constructor.

```
let  unitQtyResult = UnitQuantity.create 1

match unitQtyResult with
| Error msg ->
  printfn  "Failure, Message is %s" msg
| Ok uQty ->
  printfn  "Success. Value is %A"  uQty
  let  innerValue = UnitQuantity.value uQty
  printfn  "innerValue is %i"  innerValue
```

```
val unitQtyResult = UnitQuantity.create(1)

unitQtyResult match {
  case Left(msg) =>
    println(s"Failure, Message is $msg")
  case Right(uQty) =>
    println(s"Success. Value is $uQty")
}
```

```
Success. Value is 1
```

```
opaque type UnitQuantity = Int

object UnitQuantity:

  def create(qty: Int): Either[String, UnitQuantity] =
    if qty < 1
      Left(s”UnitQuantity can not be negative: $qty ")
    else if qty > 1000
      Left(s" UnitQuantity can not be more than 1000: $qty ")
    else
      Right(qty)
```

Here again are the declarations of our **UnitQuantity opaque type** and our **UnitQuantity** object.

In **Scala 3** we are better off calling the constructor **apply**, because that makes constructing instances more convenient, so on the next slide we rename **create** to **apply**.

```
@main def main: Unit =

  val quantities: List[Either[String,UnitQuantity]] =
    List( UnitQuantity.create(-5),
          UnitQuantity.create(99),
          UnitQuantity.create(2000) )

  val expectedQuantities: List[Either[String,UnitQuantity]] =
    List( Left("cannot be negative: -5"),
          UnitQuantity.create(99),
          Left("cannot be more than 1000: 2000") )

  assert(quantities == expectedQuantities)

  quantities.foreach { maybeQuantity =>
    println(
      maybeQuantity.fold(
        error => s"invalid UnitQuantity - $error",
        qty   => s"UnitQuantity($qty)")
    )
  }
```

Here is some code exercising and testing **UnitQuantity**. Notice how the test is not quite satisfactory in that we are not able to specify an expected value of `Right(99)`. Instead, we have to specify an expected value of `UnitQuantity.create(99)`, i.e. we have to use the same code that we are actually testing!

To address that, all we need to do is define an alternative ‘**unsafe**’ constructor that can be used in special situations, e.g. in tests. On the next slide we add such a constructor and improve our test to reflect that.

There is another issue with our code. If we print a **UnitQuantity**, the printing is done by the `toString` method of the type being aliased, i.e. `Int`

```
UnitQuantity.create(5).foreach(println) // prints 5
```

If we want to customise the printing of a **UnitQuantity**, then it turns out that **opaque type** aliases cannot redefine methods of Any such as toString, and so what we have to do is define a **UnitQuantity** **extension method**, **asString**, say, and call this method explicitly. On the next slide we define such a method and get our code to use it.

```
opaque type UnitQuantity = Int

object UnitQuantity:

  def apply(qty: Int): Either[String, UnitQuantity] =
    if qty < 1
      Left(s”UnitQuantity can not be negative: $qty ")
    else if qty > 1000
      Left(s" UnitQuantity can not be more than 1000: $qty ")
    else
      Right(qty)

  def unsafe(qty: Int): UnitQuantity = qty

  extension unitQuantityOps on (qty: UnitQuantity):
    def asString : String =
      s"UnitQuantity($qty)”
```

The ‘safe’ constructor function has been renamed from **create** to **apply**, so creating **UnitQuantity** values is more convenient.
There is now an **unsafe** constructor function used e.g. by test code.
There is now an **asString** function to be used instead of Int’s **toString**.

```
val quantities: List[Either[String,UnitQuantity]] =
  List(
    UnitQuantity(-5),
    UnitQuantity(99),
    UnitQuantity(2000)
  )

val expectedQuantities: List[Either[String,UnitQuantity]] =
  List(
    Left("UnitQuantity can not be negative: -5"),
    Right(UnitQuantity.unsafe(99)),
    Left("UnitQuantity can not be more than 1000: 2000")
  )

assert(quantities == expectedQuantities)
```

```
import UnitQuantity.unitQuantityOps

quantities.foreach { maybeQuantity =>
  println(
    maybeQuantity.fold(
      error => s"Error - $error",
      qty   => qty.asString
    )
  )
}
```

```
Error - UnitQuantity can not be negative: -5
UnitQuantity(99)
Error - UnitQuantity can not be more than 1000: 2000
```

In the next three slides we look at the first **smart constructor** example that **Debasish Ghosh** provides in his book **Functional and Reactive Domain Modeling**.

We then convert the example to use an **opaque type**.

3.3.2. The smart constructor idiom

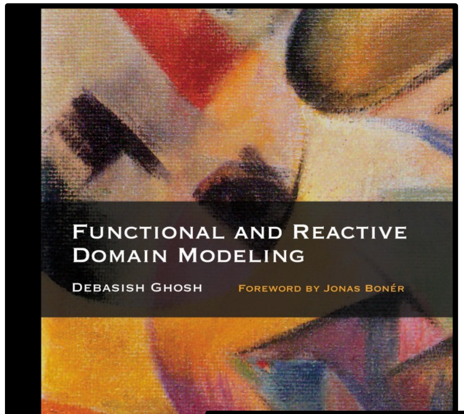
The standard technique for allowing **easy construction of objects that need to honor a set of constraints** is popularly known as **the *smart constructor* idiom**. You **prohibit the user from invoking the basic constructor of the algebraic data type and instead provide a smarter version that ensures the user gets back a data type from which she can recover either a valid instance of the domain object or an appropriate explanation of the failure**. Let's consider an example.

In our personal banking domain, many jobs may need to be scheduled for execution on specific days of the week. Here you have **an abstraction**—a ***day of the week*** **that you can implement so that you can have it validated as part of the construction process**.

You may represent a day of the week as an integer value, but obviously **it needs to honor some constraints in order to qualify as a valid day of a week**—it has to be a value between 1 and 7, 1 representing a Monday and 7 representing a Sunday. Will you do the following?

```
case class DayOfWeek(day: Int) {
  if (day < 1 or day > 7)
    throw new IllegalArgumentException("Must lie between 1 and 7")
}
```

This **violates** our **primary criterion** of **referential transparency**—an **exception** isn't one of the benevolent citizens of functional programming. Let's be smarter than that.




Debasish Ghosh
@debasishg

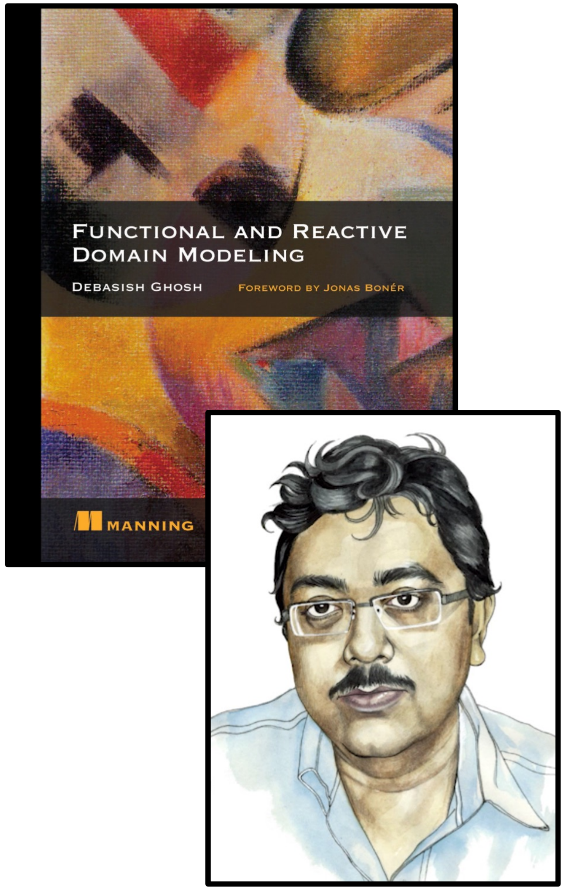
The following listing illustrates the **smart constructor idiom** for this abstraction.

Take a look at the code and then we’ll dissect it to identify the rationale.

```
sealed trait DayOfWeek {

  val value: Int

  override def toString =
    value match {
      case 1 => "Monday"
      case 2 => "Tuesday"
      case 3 => "Wednesday"
      case 4 => "Thursday"
      case 5 => "Friday"
      case 6 => "Saturday"
      case 7 => "Sunday"
    }
}
```

```
object DayOfWeek {

  private def unsafeDayOfWeek(d: Int) =
    new DayOfWeek { val value = d }

  private val isValid: Int => Boolean =
    { i => i >= 1 && i <= 7 }

  def dayOfWeek(d: Int): Option[DayOfWeek] =
    if (isValid(d))
      Some(unsafeDayOfWeek(d))
    else None

}
```

Listing 3.4. A DayOfWeek using the smart constructor idiom



Debasish Ghosh

@debasishg

Let's explore some of the features that this implementation offers that make the implementation a **smart** one:

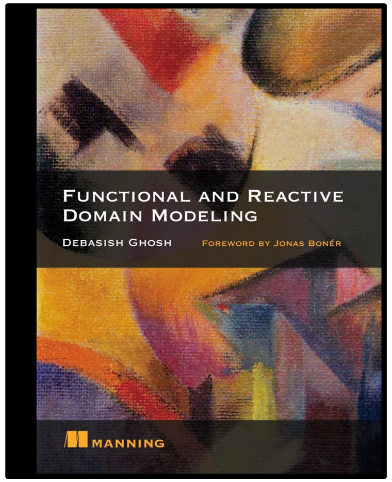
- The primary interface for creating a **DayOfWeek** has been named **unsafe** and marked **private**. It's not exposed to the user and can be used only within the implementation. There's no way the user can get back an instance of **DayOfWeek** by using this function call. This is intentional, because the instance may not be a valid one if the user passed an out-of-range integer as the argument to this function.
- The only way to get an instance of a data type representing either a valid constructed object or an absence of it is to use **dayOfWeek**, the **smart constructor** from the companion object **DayOfWeek**.
- Note the return type of the smart constructor, which is **Option**[**DayOfWeek**]. If the user passed a valid integer, then she gets back a **Some**(**DayOfWeek**), or else it's a **None**, representing the absence of a value.
- To keep the example simple, **Option** is used to represent the optional presence of the constructed instance. But for data types that may have more complex validation logic, your client may want to know the reason that the object creation failed. This can be done by using more expressive data types such as **Either** or **Try**, which allow you to return the reason, as well, in case the creation fails. You'll see an illustration of this in the next example.
- Most of the domain logic for creation and **validation** is moved away from the core abstraction, which is the **trait**, to the companion object, which is the **module**. This is what I meant by **skinny model** implementation, as opposed to **rich models** that **OO** espouses.
- A typical invocation of the smart constructor could be **DayOfWeek**.dayOfWeek(n).foreach(schedule), where schedule is a function that schedules a job on the **DayOfWeek** that it gets as input.

Debasish Ghosh
@debasishg

```
sealed trait DayOfWeek {

  val value: Int

  override def toString =
    value match {
      case 1 => "Monday"
      case 2 => "Tuesday"
      case 3 => "Wednesday"
      case 4 => "Thursday"
      case 5 => "Friday"
      case 6 => "Saturday"
      case 7 => "Sunday"
    }
}
```

```
object DayOfWeek {

   private def unsafeDayOfWeek(d: Int) =
     new DayOfWeek { val value = d }

   private val isValid: Int => Boolean =
     { i => i >= 1 && i <= 7 }

   def dayOfWeek(d: Int): Option[DayOfWeek] =
     if (isValid(d))
       Some(unsafeDayOfWeek(d))
     else None

 }
```

```
sealed trait DayOfWeek {

  val value: Int

  override def toString = value match {
    case 1 => "Monday"
    case 2 => "Tuesday"
    case 3 => "Wednesday"
    case 4 => "Thursday"
    case 5 => "Friday"
    case 6 => "Saturday"
    case 7 => "Sunday"
  }

}

object DayOfWeek {

  private def unsafeDayOfWeek(d: Int) =
    new DayOfWeek { val value = d }

  private val isValid: Int => Boolean =
    { i => i >= 1 && i <= 7 }

  def dayOfWeek(d: Int): Option[DayOfWeek] =
    if (isValid(d))
      Some(unsafeDayOfWeek(d))
    else None

}
```



Here we take the smart constructor example on the left, by **Debasish Ghosh**, and convert it to use an **opaque type**.

```
opaque type DayOfWeek = Int

object DayOfWeek {

  def apply(d: Int): Option[DayOfWeek] =
    if (isValid(d))
      Some(d)
    else None

  def unsafe(d: Int): DayOfWeek = d

  private val isValid: Int => Boolean =
    i => i >= 1 && i <= 7

  extension dayOfWeekOps on (d: DayOfWeek):

    def asString : String =
      d match {
        case 1 => "Monday"
        case 2 => "Tuesday"
        case 3 => "Wednesday"
        case 4 => "Thursday"
        case 5 => "Friday"
        case 6 => "Saturday"
        case 7 => "Sunday"
      }

}
```

```scala
opaque type DayOfWeek = Int

object DayOfWeek {

  def apply(d: Int): Option[DayOfWeek] =
    if (isValid(d))
      Some(d)
    else None

  def unsafe(d: Int): DayOfWeek = d

  private val isValid: Int => Boolean =
    i => i >= 1 && i <= 7

  extension dayOfWeekOps on (d: DayOfWeek):

    def asString : String =
      d match {
        case 1 => "Monday"
        case 2 => "Tuesday"
        case 3 => "Wednesday"
        case 4 => "Thursday"
        case 5 => "Friday"
        case 6 => "Saturday"
        case 7 => "Sunday"
      }

}
```

Here we take our **DayOfWeek simple type** and add a test, plus some code that uses it.

@philip_schwarz

```scala
val daysOfWeek: List[Option[DayOfWeek]] =
  List(
    DayOfWeek(-5),
    DayOfWeek(3),
    DayOfWeek(10)
  )

val expectedDaysOfWeek: List[Option[DayOfWeek]] =
  List(
    None,
    Some(DayOfWeek.unsafe(3)),
    None
  )

assert(daysOfWeek == expectedDaysOfWeek)
```

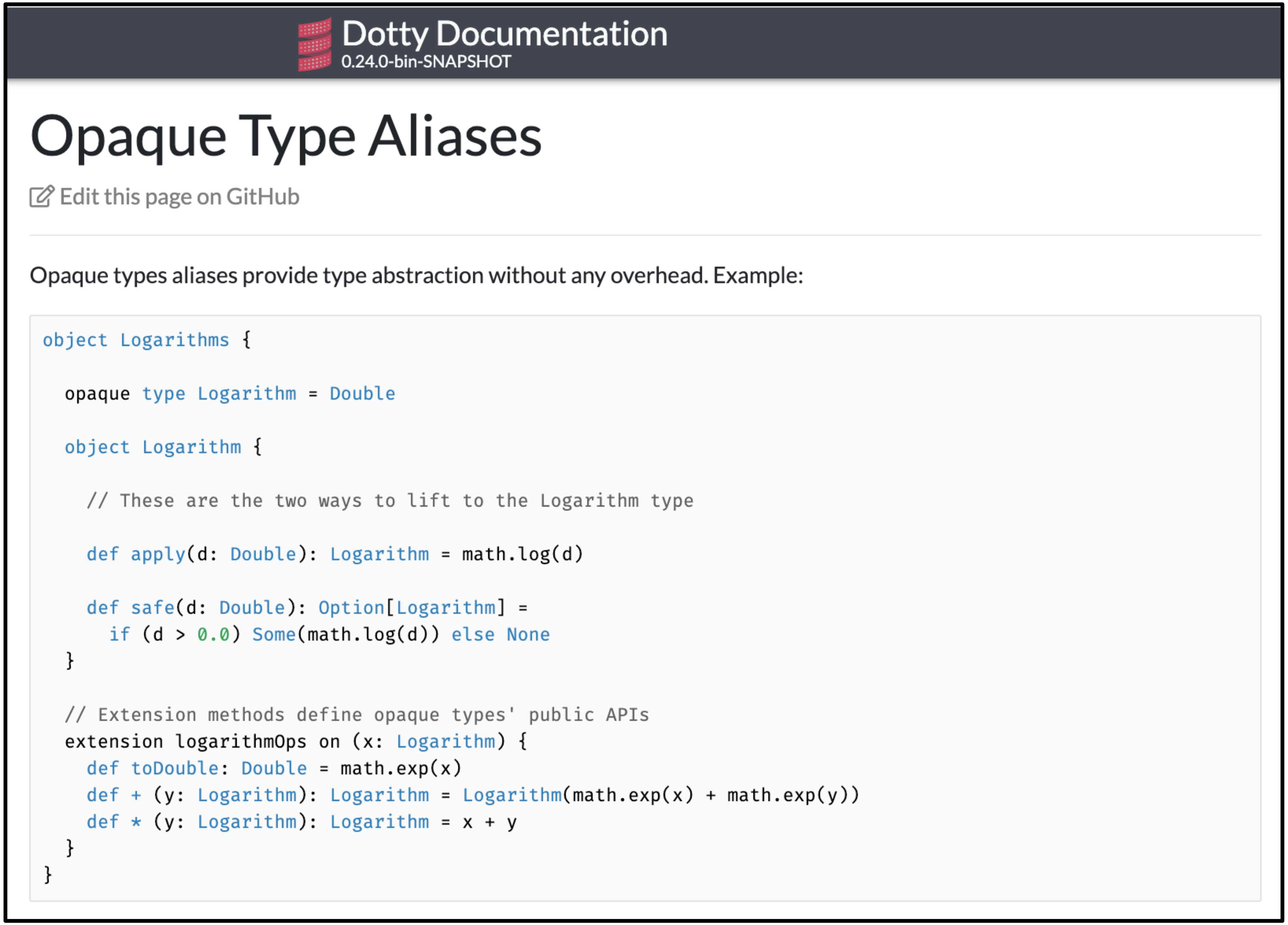
```scala
// printing DayOfWeek using toString
daysOfWeek.foreach(println)

None
Some(3)
None
```

```scala
// priting DayOfWeek using asString
daysOfWeek.foreach { maybeDay =>
  println(
    maybeDay.fold
      ("Error: undefined DayOfWeek")
      (day => day.asString)
  )
}

Error: undefined DayOfWeek
Wednesday
Error: undefined DayOfWeek
```

In his book, **Debasish Ghosh** goes on to provide a more involved example of **smart constructor**. If you are interested, see here for details: https://github.com/debasishg/frdomain/tree/master/src/main/scala/frdomain/ch3/smartconstructor

The next slide consists of an extract from the current **Dotty** documentation, which is where I got the idea of adding the **asString extension function** to our **opaque types**.

The extract is also interesting in that it shows an example where the **unsafe** constructor is called **apply** and the **safe** one is called **safe**.



Opaque Type Aliases

Opaque types aliases provide type abstraction without any overhead. Example:

```scala
object Logarithms {

  opaque type Logarithm = Double

  object Logarithm {

    // These are the two ways to lift to the Logarithm type

    def apply(d: Double): Logarithm = math.log(d)

    def safe(d: Double): Option[Logarithm] =
      if (d > 0.0) Some(math.log(d)) else None
  }

  // Extension methods define opaque types' public APIs
  extension logarithmOps on (x: Logarithm) {
    def toDouble: Double = math.exp(x)
    def + (y: Logarithm): Logarithm = Logarithm(math.exp(x) + math.exp(y))
    def * (y: Logarithm): Logarithm = x + y
  }
}
```

https://dotty.epfl.ch/docs/reference/other-new-features/opaques.html

Next, we take the e-commerce payments example on this slide and in the next two slides we add **integrity checks** for **Simple values** by adding **validation** to **Simple types** (the types in red) using the **smart constructor** pattern.

```
enum CardType:
  case Visa, Mastercard

enum Currency:
  case EUR, USD

object OpaqueTypes:

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

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

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

```
import OpaqueTypes._

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

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

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

```
@main def main =

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

  val check350USD = Payment(
    PaymentAmount(350),
    Currency.USD,
    PaymentMethod.Check(CheckNumber(123)))

  println(cash10EUR)
  println(check350USD)
```

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

@philip_schwarz

Here we add **validation** to our **simple types** (the types in red), by modifying their associated **apply** function to return either a valid value or an error message. We also introduce an **unsafe** constructor function for each type.

```scala
opaque type CardNumber = String
object CardNumber:

  def apply(n: String): Either[String, CardNumber] =
    if n < "111111"
      Left(s"CardNumber cannot be less than 111111: $n")
    else if n > "999999"
      Left(s"CardNumber cannot be greater than 999999: $n")
    else
      Right(n)

  def unsafe(n: String): CardNumber = n
```

```scala
opaque type CheckNumber = Int
object CheckNumber:

  def apply(n: Int): Either[String, CheckNumber] =
    if n < 1
      Left(s"CheckNumber cannot be less than 1: $n")
    else if n > 1000000
      Left(s"CheckNumber cannot be greater than 1,000,000: $n")
    else
      Right(n)

  def unsafe(n: Int): CheckNumber = n
```

```scala
opaque type PaymentAmount = Float
object PaymentAmount:

  def apply(amount: Float): Either[String, PaymentAmount] =
    if amount < 0
      Left(s"PaymentAmount cannot be negative: $amount")
    else if amount > 1000000
      Left(s"PaymentAmount cannot be greater than 1,000,000: $amount")
    else
      Right(amount)

  def unsafe(amount: Float): PaymentAmount = amount
```

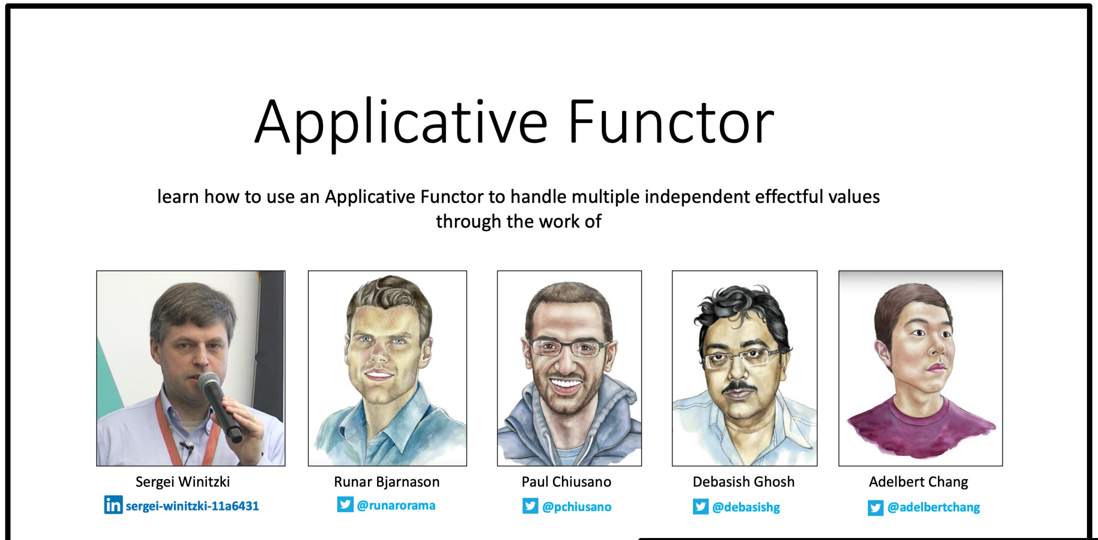
```scala
enum CardType:                                 enum Currency:
  case Visa, Mastercard                          case EUR, USD

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

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

Here we add to the **Payment** type some constructors that make the **validation** of **Payment** dependent on the **validation** of **Simple types**. We also add a test.

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

  def apply(amount: Either[String, PaymentAmount],
            currency: Currency,
            checkNumber: Either[String, CheckNumber])
           : Either[String, Payment] =
    for {
          amt <- amount
      checkNo <- checkNumber
    } yield Payment(amt, currency, PaymentMethod.Check(checkNo))

  def apply(amount: Either[String, PaymentAmount],
            currency: Currency)
           : Either[String, Payment] =
    for {
      amt <- amount
    } yield Payment(amt, currency, PaymentMethod.Cash)

  def apply(amount: Either[String, PaymentAmount],
            currency: Currency,
            cardType: CardType,
            cardNumber: Either[String, CardNumber])
           : Either[String, Payment] =
    for {
         amt <- amount
      cardNo <- cardNumber
      cardInfo = CreditCardInfo(cardType, cardNo)
    } yield Payment(amt, currency, PaymentMethod.Card(cardInfo))
}
```

```scala
val payments: List[Either[String, Payment]] =
  List(
    Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
    Payment(PaymentAmount(10),  Currency.USD, CheckNumber(2_000_000)),
    Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
    Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("005")),
    Payment(PaymentAmount(30),  Currency.EUR),
    Payment(PaymentAmount(-30), Currency.EUR)
  )

val expectedPayments List[Either[String, Payment]] =
  List(
    Right(Payment(PaymentAmount.unsafe(10.0),
                  Currency.USD,
                  PaymentMethod.Check(CheckNumber.unsafe(15)))),
    Left("CheckNumber cannot be greater than 1,000,000: 2000000"),
    Right(Payment(PaymentAmount.unsafe(20.0),
                  Currency.EUR,
                  PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                    CardNumber.unsafe("123"))))),
    Left("CardNumber cannot be less than 111111: 005"),
    Right(Payment(PaymentAmount.unsafe(30.0),
                  Currency.EUR,
                  PaymentMethod.Cash)),
    Left("PaymentAmount cannot be negative: -30.0")
  )

assert(payments == expectedPayments)

payments.foreach(println)
```

```
Right(Payment(10.0,USD,Check(15)))
Left(CheckNumber cannot be greater than 1,000,000: 2000000)
Right(Payment(20.0,EUR,Card(CreditCardInfo(Visa,123))))
Left(CardNumber cannot be less than 111111: 005)
Right(Payment(30.0,EUR,Cash))
Left(PaymentAmount cannot be negative: -30.0)
```



Applicative Functor

learn how to use an Applicative Functor to handle multiple independent effectful values
through the work of

Sergei Winitzki
sergei-winitzki-11a6431

Runar Bjarnason
@runarorama

Paul Chiusano
@pchiusano

Debasish Ghosh
@debasishg

Adelbert Chang
@adelbertchang

slides by @philip_schwarz

@philip_schwarz

The rest of this slide deck involves the use of the **Applicative** **type class**. If you are not familiar with it then one way you can learn about it is by going through the slide decks on this slide, though you might still find it useful to skim through the rest of the slides here to get some idea of how **Applicative** can be used for **validation**.

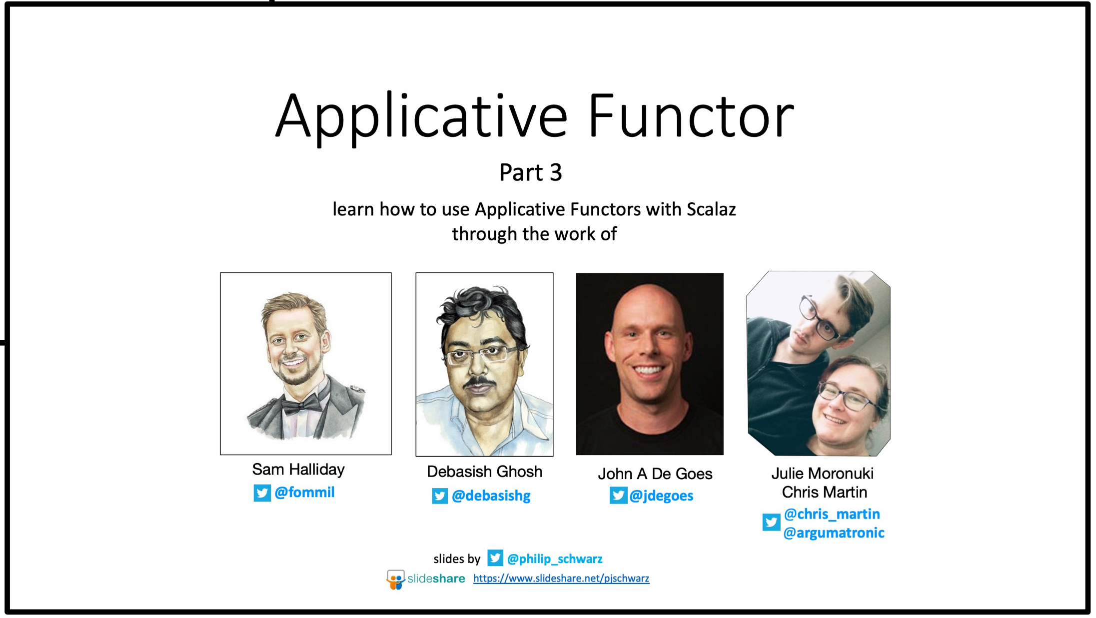
Applicative Functor

Part 2

Learn more about the canonical definition of the Applicative typeclass by looking at
a great Haskell validation example by Chris Martin and Julie Moronuki
Then see it translated to Scala

@chris_martin @argumatronic

slides by @philip_schwarz



Applicative Functor

Part 3

learn how to use Applicative Functors with Scalaz
through the work of

Sam Halliday
@fommil

Debasish Ghosh
@debasishg

John A De Goes
@jdegoes

Julie Moronuki
Chris Martin
@chris_martin
@argumatronic

slides by @philip_schwarz
slideshare https://www.slideshare.net/pjschwarz

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

The code that we have just seen has the limitation that even though the **validation** of a **Simple type** might fail due to **multiple constraints** being **violated**, our constructors return an **error** that only informs us about a **single violation**.

e.g. if we try to create a **Payment** with both an **invalid PaymentAmount** and an **invalid CheckNumber** then the resulting **error** only informs us that the **PaymentAmount** is **invalid**.

```
assert( Payment(PaymentAmount(-10),  Currency.USD, CheckNumber(2_000_000))
        ==
        Left("PaymentAmount cannot be negative: -10.0"))
```

This is happening because **Either** is a **monad**, so our **validations** are carried out **sequentially** and as soon as one of the **validations** fails then the remaining **validations** are not even attempted, they are bypassed, and we are left with an **error** informing us of that **single failure**.

```
 def apply(amt:Either[String,PaymentAmount],ccy:Currency,cardType:CardType,cardNo:Either[String,CardNumber])
          :Either[String,Payment] =
    for {
          amount <- amt
      cardNumber <- cardNo
      cardInfo = CreditCardInfo(cardType, cardNumber)
    } yield Payment(amt, ccy, PaymentMethod.Card(cardInfo))
}
```

If the **amt** **validation** is a **failure** i.e. an **Either**, then the whole **for comprehension** returns that **failure**: the **cardNo** **validation** does not even get used in the computation of the result.

In the following slides we take the code that we have just seen and improve it so that it **returns** an **error for all constraint violations, not just a single violation**. We are going to do that by changing the code so that rather than doing **validation** using the **Either monad**, it is going to do it using an **Applicative Functor**.

If we want to use **Applicative** and related abstractions to do **validation**, can’t we just use the ready-made abstractions provided by a functional programming library like **Scalaz** or **Cats**?

No we cannot, because we are using **Scala 3** and there are no versions of **Scalaz** and/or **Cats** available for **Scala 3** because the latter has not been released yet.

So we are going to use a hand-rolled **Applicative**. We are first going to use the one seen in the slide deck on the right.

Applicative Functor

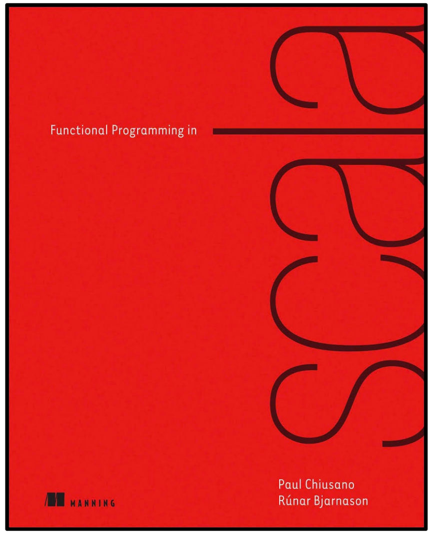
Part 2

Learn more about the canonical definition of the Applicative typeclass by looking at
a great Haskell validation example by Chris Martin and Julie Moronuki
Then see it translated to Scala

@chris_martin @argumatronic

slides by @philip_schwarz

The resulting code is going to be somewhat overcomplicated for our simple use case, so we are then going to switch to a simpler **Applicative** seen in **Functional Programming in Scala**.



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

```
trait Semigroup[A] {
  def <>(lhs: A, rhs: A): A
}

trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}

trait Applicative[F[_]] extends Functor[F] {
  def <*>[A,B](fab: F[A => B],fa: F[A]): F[B]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] =
    <*>(unit(f),fa)
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C] =
    <*>(map(fa)(f.curried), fb)
}

trait Traverse [F[_]] {
  def traverse[M[_]:Applicative,A,B](fa:F[A])(f: A => M[B]): M[F[B]]
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(x => x)
}
```

The **Applicative** instance that we are going to use will involve a **Semigroup**.

An **Applicative** is a **Functor**.

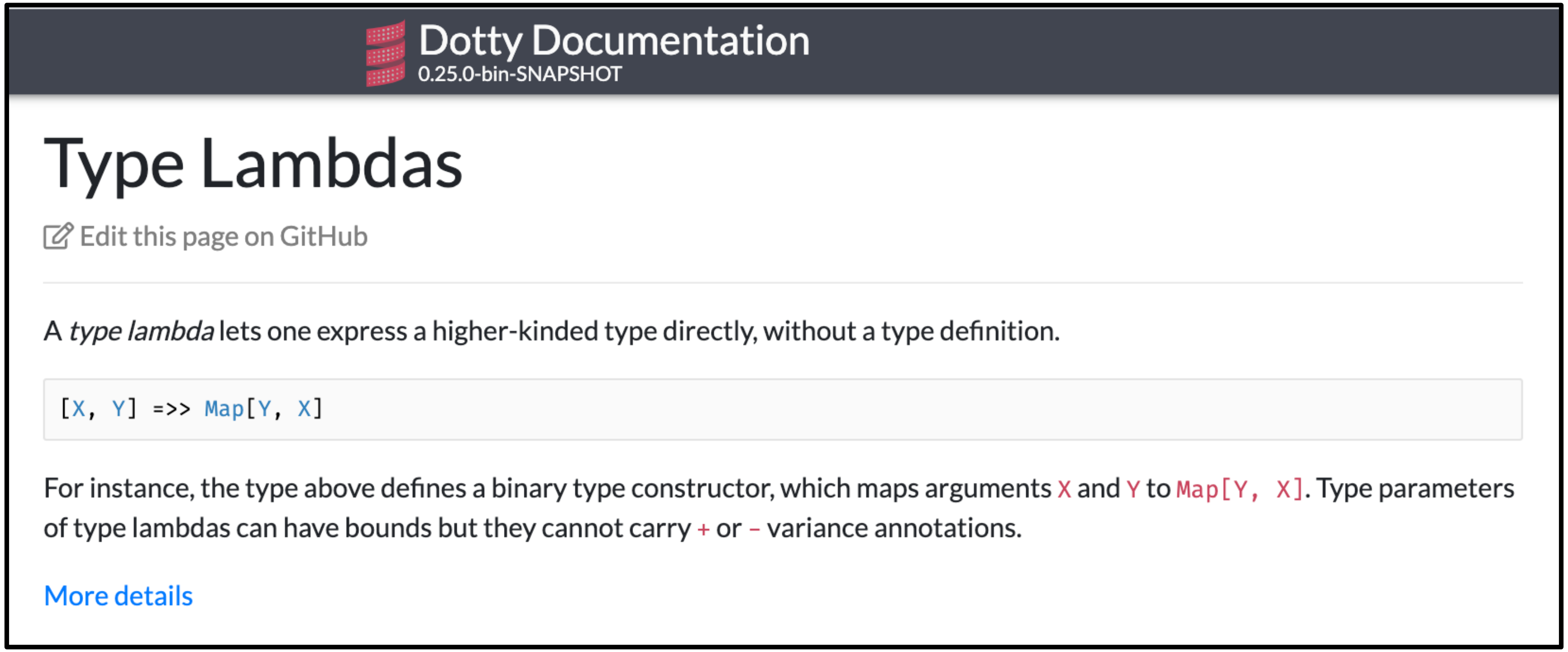
The first hand-rolled **Applicative** that we are going to use is defined in terms of **<*>** (tie-fighter) and **unit**. ***>** (right-shark) was also provided but I have removed it because we don't need it.

Whilst we could get our code to use **<*>** directly, this would not be as convenient as using **map** and **map2** and since these can be defined in terms of **<*>** and **unit**, I have added them to **Applicative**.

The **sequence** function provided by **Traverse** is going to come in handy when we write tests.

@philip_schwarz

On the next slide we see a **Scala 3** **type lambda**. While explaining **type lambdas** is out of scope for this slide deck, here are the first few lines from the dotty 0.25.0 documentation for **type lambdas**.



Dotty Documentation
0.25.0-bin-SNAPSHOT

Type Lambdas

Edit this page on GitHub

A *type lambda* lets one express a higher-kinded type directly, without a type definition.

```
[X, Y] =>> Map[Y, X]
```

For instance, the type above defines a binary type constructor, which maps arguments `X` and `Y` to `Map[Y, X]`. Type parameters of type lambdas can have bounds but they cannot carry `+` or `-` variance annotations.

More details

```
case class Error(error:List[String])

sealed trait Validation[+E, +A]
case class Failure[E](error: E) extends Validation[E, Nothing]
case class Success[A](a: A) extends Validation[Nothing, A]

implicit val errorSemigroup: Semigroup[Error] =
  new Semigroup[Error] {
    def <>(lhs: Error, rhs: Error): Error =
      Error(lhs.error ++ rhs.error)
  }

def validationApplicative[E](implicit sg:Semigroup[E])
    : Applicative[[α] =>> Validation[E,α]] =
  new Applicative[[α] =>> Validation[E,α]] {
    def unit[A](a: => A) = Success(a)
    def <*>[A,B](fab: Validation[E,A => B], fa: Validation[E,A]): Validation[E,B] =
      (fab, fa) match {
        case (Success(ab), Success(a)) => Success(ab(a))
        case (Failure(err1), Failure(err2)) => Failure(sg.<>(err1,err2))
        case (Failure(err), _) => Failure(err)
        case (_, Failure(err)) => Failure(err)
      }
}
val errorValidationApplicative: Applicative[[α] =>> Validation[Error,α]] =
  validationApplicative[Error]

val listTraverse = new Traverse[List] {
  override def traverse[M[_],A,B](as:List[A])(f: A => M[B])
                                 (implicit M:Applicative[M])
                                 : M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}
```

The **Applicative** that we are going to use is a **Validation** whose **Failure** contains an **Error** consisting of a **List** of **error messages**.

There is a **Semigroup** that can be used to combine errors by **concatenating** their lists of **error messages**.

When **<*>** is used to **apply** a function to its argument then if both the function and the argument are failed **validations** then **<*>** returns a failed **Validation** whose **error** is the combination of the errors of the two **validations**.

This is a validation **Applicative** instance for **Error**.

We are going to use a **Traverse** instance for **List**.

```
opaque type CardNumber = String
object CardNumber:
  def apply(n: String): Validation[Error,CardNumber] =
    if n < "111111"
      Failure(Error(List(s"CardNumber cannot be less than 111111: $n")))
    else if n > "999999"
     Failure(Error(List(s"CardNumber cannot be greater than 999999: $n")))
    else
      Success(n)
  def unsafe(n: String): CardNumber = n
```

```
opaque type CheckNumber = Int
object CheckNumber:
  def apply(n: Int): Validation[Error,CheckNumber] =
    if n < 1
      Failure(Error(List(s"CheckNumber cannot be less than 1: $n")))
    else if n > 1000000
     Failure(Error(List(s"CheckNumber cannot be greater than 1,000,000: $n")))
    else
      Success(n)
  def unsafe(n: Int): CheckNumber = n
```

```
opaque type PaymentAmount = Float
object PaymentAmount:
  def apply(n: Float): Validation[Error,PaymentAmount] =
    if n < 0
      Failure(Error(List(s"PaymentAmount cannot be negative: $n")))
    else if n > 1000000
     Failure(Error(List(s"PaymentAmount cannot be greater than 1,000,000: $n")))
    else
      Success(n)
  def unsafe(n: Float): PaymentAmount = n
```

```
enum CardType:
  case Visa, Mastercard

enum Currency :
 case EUR, USD

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

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

On this slide we have just changed the **Simple type** constructors so that rather than returning an `Either[String,X]` they return a `Validation[Error,X]`.

```
case class Payment private (
  amount: PaymentAmount,
  currency: Currency,
  method: PaymentMethod
)
import errorValidationApplicative._
object Payment {

  def apply(amount: Validation[Error,PaymentAmount],
            currency: Currency,
            checkNumber: Validation[Error,CheckNumber])
    : Validation[Error,Payment] =
    map2(amount, checkNumber)(
      (amt, checkNo) =>
        Payment(amt, currency, PaymentMethod.Check(checkNo))
    )

  def apply(amount: Validation[Error,PaymentAmount],
            currency: Currency)
    : Validation[Error,Payment] =
    map(amount)(amt => Payment(amt, currency, PaymentMethod.Cash))

  def apply(amount: Validation[Error,PaymentAmount],
            currency: Currency,
            card: CardType,
            cardNumber: Validation[Error,CardNumber])
    : Validation[Error,Payment] =
    map2(amount, cardNumber)(
      (amt, cardNo) =>
        Payment(amt, currency, PaymentMethod.Card(CreditCardInfo(card, cardNo)))
    )

  def unsafe(amount: PaymentAmount, currency: Currency, method: PaymentMethod): Payment =
    Payment(amount, currency, method)
}
```

Here we import the interface provided by the **Validation Applicative** for **Error**, so that we have access to its **map** and **map2** functions.

On this slide we have firstly changed the **Payment** constructors so that where they take or return an `Either[String,X]` they instead take or return a `Validation[Error,X]`.

Secondly, since we have stopped using the **Either monad** and started using a **Validation Applicative**, instead of using a **for comprehension** to process input **Validations** and return a **Validation**, we now use **map** and **map2** to do that.

We have also added an 'unsafe' constructor to be used in test code.

```
val successfulPaymentValidations: List[Validation[Error,Payment]] =
  List(Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
       Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
       Payment(PaymentAmount(30),  Currency.EUR))

val expectedSuccessfulPaymentValidations: List[Validation[Error,Payment]] =
  List(Success(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15)))),
       Success(Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                 CardNumber.unsafe("123"))))),
       Success(Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)))
```

```
assert(successfulPaymentValidations
       == expectedSuccessfulPaymentValidations)
```

The top test is new and operates purely on **successful payment validations**. See the next slide for an existing test which also deals with **failed** payment validations.

The bottom test shows off how useful the **sequence** function of **`Traverse`** can be to turn a collection of **payment validations** into a **validated** collection of **payments**. The only reason why I am explicitly passing the **`errorValidationApplicative`** to **sequence** is to remind you that it is being passed in.

@philip_schwarz

```
// turn a list of successful payment validations into
// a succesful validation of a list of payments
val successfulValidationOfPayments : Validation[Error,List[Payment]] =
  listTraverse.sequence(successfulPaymentValidations)(errorValidationApplicative)

val expectedSuccessfulValidationOfPayments : Validation[Error,List[Payment]] =
  Success(List(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15))),
               Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                 CardNumber.unsafe("123")))),
               Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)))
```

```
assert(successfulValidationOfPayments
       == expectedSuccessfulValidationOfPayments)
```

```
val successfulAndUnsuccessfulPaymentValidations: List[Validation[Error,Payment]] =
  List(Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
       Payment(PaymentAmount(-10), Currency.USD, CheckNumber(2_000_000)),
       Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
       Payment(PaymentAmount(-20), Currency.EUR, CardType.Visa, CardNumber("005")),
       Payment(PaymentAmount(30),  Currency.EUR),
       Payment(PaymentAmount(-30), Currency.EUR))

val expectedSuccessfulAndUnsuccessfulPaymentValidations
    : List[Validation[Error,Payment]] =
  List(Success(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15)))),
       Failure(Error(List("PaymentAmount cannot be negative: -10.0",
                          "CheckNumber cannot be greater than 1,000,000: 2000000"))),
       Success(Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,CardNumber.unsafe("123"))))),
       Failure(Error(List("PaymentAmount cannot be negative: -20.0",
                          "CardNumber cannot be less than 111111: 005"))),
       Success(Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)),
       Failure(Error(List("PaymentAmount cannot be negative: -30.0"))))
```

On this slide we revisit the previous two tests but include **failed validations**. This showcases how the **Validation Applicative** for **Error** is able to return **Validation failures** informing us of **all** **constraint violations**, rather than just **one** (the first one).

Note that since **sequence** is defined in terms of the **map2** function of the **applicative** that it is being passed, the **sequence** function benefits from the ability of said **map2** function to combine **failed validations** and so the end result of **sequencing** is a single **failure** whose **Error** informs us of **all** **constraint violations**.

```
assert( successfulAndUnsuccessfulPaymentValidations
        == expectedSuccessfulAndUnsuccessfulPaymentValidations)
```

```
// turn a list of partly failed payment validations into a failed validation of a list of payments
val failedValidationOfPayments : Validation[Error,List[Payment]] =
  listTraverse.sequence(successfulAndUnsuccessfulPaymentValidations)(errorValidationApplicative)

val expectedFailedValidationOfPayments : Validation[Error,List[Payment]] =
  Failure(Error(List("PaymentAmount cannot be negative: -10.0",
                     "CheckNumber cannot be greater than 1,000,000: 2000000",
                     "PaymentAmount cannot be negative: -20.0",
                     "CardNumber cannot be less than 111111: 005",
                     "PaymentAmount cannot be negative: -30.0")))
```

```
assert(failedValidationOfPayments
       == expectedFailedValidationOfPayments)
```

The next eight slides conclude this slide deck. What they do is simplify the program that we have just seen by using a simpler **Validation** and a simpler **Applicative** that doesn't rely on a **Semigroup**. The **Validation** and **Applicative** are from chapter 12 of **Functional Programming in Scala** and more specifically from Exercise 6 of that chapter.

I am not going to provide any commentary. On each slide you'll see some code from the current program together with alternative code that replaces it to produce a simplified version of the program.

Validation - much like **Either**, except it can handle more than one error

Let's invent a new data type, `Validation`, that is much like `Either` except that it can explicitly handle more than one error:

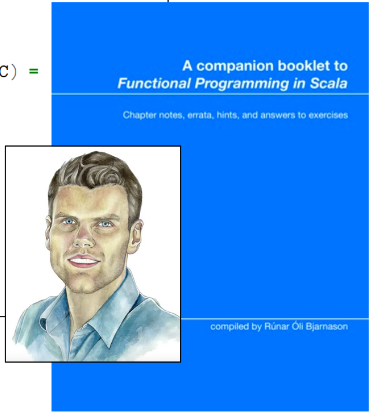
```
sealed trait Validation[+E, +A]

case class Failure[E](head: E, tail: Vector[E] = Vector())
  extends Validation[E, Nothing]

case class Success[A](a: A) extends Validation[Nothing, A]
```

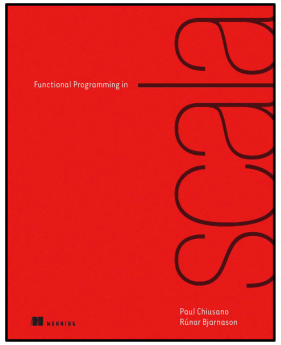
EXERCISE 12.6

Write an `Applicative` instance for `Validation` that accumulates errors in `Failure`. Note that in the case of `Failure` there's always at least one error, stored in `head`. The rest of the errors accumulate in the `tail`.

Exercise 12.06

```
def validationApplicative[E]: Applicative[({type f[x] = Validation[E,x]})#f] =
  new Applicative[({type f[x] = Validation[E,x]})#f] {
    def unit[A](a: => A) = Success(a)
    override def map2[A,B,C](fa: Validation[E,A],
                             fb: Validation[E,B])(f: (A, B) => C) =
      (fa, fb) match {
        case (Success(a), Success(b)) => Success(f(a, b))
        case (Failure(h1, t1), Failure(h2, t2)) =>
          Failure(h1, t1 ++ Vector(h2) ++ t2)
        case (e@Failure(_, _), _) => e
        case (_, e@Failure(_, _)) => e
      }
  }
```





Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

```scala
case class Error(error:List[String])
```

```scala
sealed trait Validation[+E, +A]
case class Failure[E](error: E)
          extends Validation[E, Nothing]
case class Success[A](a: A)
          extends Validation[Nothing, A]
```

```scala
sealed trait Validation[+E, +A]
case class Failure[E](head: E, tail: Vector[E] = Vector())
          extends Validation[E, Nothing]
case class Success[A](a: A)
          extends Validation[Nothing, A]
```

```scala
trait Semigroup[A] {
  def <>(lhs: A, rhs: A): A
}
```

```scala
trait Applicative[F[_]] extends Functor[F] {
  def <*>[A,B](fab: F[A => B],fa: F[A]): F[B]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] =
    <*>(unit(f),fa)
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C] =
    <*>(map(fa)(f.curried), fb)
}
```

```scala
trait Applicative[F[_]] extends Functor[F] {
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] =
    map2(fa,unit(()))((a,_) => f(a))
}
```

```scala
def validationApplicative[E](implicit sg:Semigroup[E]): Applicative[[α] =>> Validation[E,α]] =
  new Applicative[[α] =>> Validation[E,α]] {
    def unit[A](a: => A) = Success(a)
    def <*>[A,B](fab: Validation[E,A => B], fa: Validation[E,A]): Validation[E,B] =
      (fab, fa) match {
        case (Success(ab), Success(a)) => Success(ab(a))
        case (Failure(err1), Failure(err2)) => Failure(sg.<>(err1,err2))
        case (Failure(err), _) => Failure(err)
        case (_, Failure(err)) => Failure(err)
      }
}
val errorValidationApplicative: Applicative[[α] =>> Validation[Error,α]] =
  validationApplicative[Error]
```

```scala
def validationApplicative[E]: Applicative[[α] =>> Validation[E,α]] =
  new Applicative[[α] =>> Validation[E,α]] {
    def unit[A](a: => A) = Success(a)
    def map2[A,B,C](fa: Validation[E,A], fb: Validation[E,B])(f: (A,B) => C): Validation[E,C] =
      (fa, fb) match {
        case (Success(a), Success(b)) => Success(f(a, b))
        case (Failure(h1, t1), Failure(h2, t2)) => Failure(h1, t1 ++ Vector(h2) ++ t2)
        case (e@Failure(_, _), _) => e
        case (_, e@Failure(_, _)) => e
      }
}
val errorValidationApplicative: Applicative[[α] =>> Validation[String,α]] =
  validationApplicative[String]
```

```scala
opaque type CheckNumber = Int
object CheckNumber:
  def apply(n: Int): Validation[Error,CheckNumber] =
    if n < 1
      Failure(Error(List(s"CheckNumber cannot be less than 1: $n")))
    else if n > 1000000
      Failure(Error(List(s"CheckNumber cannot be greater than 1,000,000: $n")))
    else
      Success(n)
  def unsafe(n: Int): CheckNumber = n
```

```scala
opaque type CheckNumber = Int
object CheckNumber:
  def apply(n: Int): Validation[String,CheckNumber] =
    if n < 1
      Failure("CheckNumber cannot be less than 1: $n")
    else if n > 1000000
      Failure(s"CheckNumber cannot be greater than 1,000,000: $n")
    else
      Success(n)
  def unsafe(n: Int): CheckNumber = n
```

```scala
opaque type CardNumber = String
object CardNumber:
  def apply(n: String): Validation[Error,CardNumber] =
    if n < "111111"
      Failure(Error(List(s"CardNumber cannot be less than 111111: $n")))
    else if n > "999999"
      Failure(Error(List(s"CardNumber cannot be greater than 999999: $n")))
    else
      Success(n)
  def unsafe(n: String): CardNumber = n
```

```scala
opaque type CardNumber = String
object CardNumber:
  def apply(n: String): Validation[String,CardNumber] =
    if n < "111111"
      Failure(s"CardNumber cannot be less than 111111: $n")
    else if n > "999999"
      Failure(s"CardNumber cannot be greater than 999999: $n")
    else
      Success(n)
  def unsafe(n: String): CardNumber = n
```

```scala
opaque type PaymentAmount = Float
object PaymentAmount:
  def apply(n: Float): Validation[Error,PaymentAmount] =
    if n < 0
      Failure(Error(List(s"PaymentAmount cannot be negative: $n")))
    else if n > 1000000
      Failure(Error(List(s"PaymentAmount cannot be greater than 1,000,000: $n")))
    else
      Success(n)
  def unsafe(n: Float): PaymentAmount = n
```

```scala
opaque type PaymentAmount = Float
object PaymentAmount:
  def apply(n: Float): Validation[String,PaymentAmount] =
    if n < 0
      Failure(s"PaymentAmount cannot be negative: $n")
    else if n > 1000000
      Failure(s"PaymentAmount cannot be greater than 1,000,000: $n")
    else
      Success(n)
  def unsafe(n: Float): PaymentAmount = n
```

```scala
case class Payment private (
  amount: PaymentAmount,
  currency: Currency,
  method: PaymentMethod
)
import errorValidationApplicative._
object Payment {
  def apply(amount: Validation[Error,PaymentAmount],
            currency: Currency,
            checkNumber: Validation[Error,CheckNumber])
            : Validation[Error,Payment] =
    map2(amount, checkNumber)(
      (amt, checkNo) =>
        Payment(amt, currency, PaymentMethod.Check(checkNo))
    )
  def apply(amount: Validation[Error,PaymentAmount],
            currency: Currency)
            : Validation[Error,Payment] =
    map(amount)(amt => Payment(amt, currency, PaymentMethod.Cash))
  def apply(amount: Validation[Error,PaymentAmount],
            currency: Currency,
            card: CardType,
            cardNumber: Validation[Error,CardNumber])
            : Validation[Error,Payment] =
    map2(amount, cardNumber)(
      (amt, cardNo) =>
        Payment(amt,
                currency,
                PaymentMethod.Card(CreditCardInfo(card, cardNo)))
    )
  def unsafe(amount: PaymentAmount,
             currency: Currency,
             method: PaymentMethod): Payment =
    Payment(amount, currency, method)
}
```

```scala
case class Payment private (
  amount: PaymentAmount,
  currency: Currency,
  method: PaymentMethod
)
import errorValidationApplicative._
object Payment {
  def apply(amount: Validation[String,PaymentAmount],
            currency: Currency,
            checkNumber: Validation[String,CheckNumber])
            : Validation[String,Payment] =
    map2(amount, checkNumber)(
        (amt, checkNo) =>
          Payment(amt, currency, PaymentMethod.Check(checkNo))
    )
  def apply(amount: Validation[String,PaymentAmount],
            currency: Currency)
            : Validation[String,Payment] =
    map(amount)(amt => Payment(amt, currency, PaymentMethod.Cash))
  def apply(amount: Validation[String,PaymentAmount],
            currency: Currency,
            card: CardType,
            cardNumber: Validation[String,CardNumber])
            : Validation[String,Payment] =
    map2(amount, cardNumber)(
        (amt, cardNo) =>
          Payment(amt,
                  currency,
                  PaymentMethod.Card(CreditCardInfo(card, cardNo)))
    )
  def unsafe(amount: PaymentAmount,
             currency: Currency,
             method: PaymentMethod): Payment =
    Payment(amount, currency, method)
}
```

```
val successfulPaymentValidations: List[Validation[Error,Payment]] =
  List(Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
       Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
       Payment(PaymentAmount(30),  Currency.EUR))

val expectedSuccessfulPaymentValidations: List[Validation[Error,Payment]] =
  List(Success(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15)))),
       Success(Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                                CardNumber.unsafe("123"))))),
       Success(Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)))
```

```
assert(successfulPaymentValidations
         == expectedSuccessfulPaymentValidations)
```

```
val successfulPaymentValidations: List[Validation[String,Payment]] =
  List(Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
       Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
       Payment(PaymentAmount(30),  Currency.EUR))

val expectedSuccessfulPaymentValidations: List[Validation[String,Payment]] =
  List(Success(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15)))),
       Success(Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                                CardNumber.unsafe("123"))))),
       Success(Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)))
```

```scala
// turn a list of successful payment validations into
// a succesful validation of a list of payments
val successfulValidationOfPayments : Validation[Error,List[Payment]] =
  listTraverse.sequence(successfulPaymentValidations)(errorValidationApplicative)

val expectedSuccessfulValidationOfPayments : Validation[Error,List[Payment]] =
  Success(List(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15))),
               Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                                CardNumber.unsafe("123")))),
               Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)))
```

```scala
assert(successfulValidationOfPayments
       == expectedSuccessfulValidationOfPayments)
```

```scala
// turn a list of successful payment validations into
// a succesful validation of a list of payments
val successfulValidationOfPayments : Validation[String,List[Payment]] =
  listTraverse.sequence(successfulPaymentValidations)(errorValidationApplicative)

val expectedSuccessfulValidationOfPayments : Validation[String,List[Payment]] =
  Success(List(Payment.unsafe(PaymentAmount.unsafe(10.0),
                              Currency.USD,
                              PaymentMethod.Check(CheckNumber.unsafe(15))),
               Payment.unsafe(PaymentAmount.unsafe(20.0),
                              Currency.EUR,
                              PaymentMethod.Card(CreditCardInfo(CardType.Visa,
                                                                CardNumber.unsafe("123")))),
               Payment.unsafe(PaymentAmount.unsafe(30.0),
                              Currency.EUR,
                              PaymentMethod.Cash)))
```

```scala
val successfulAndUnsuccessfulPaymentValidations
    : List[Validation[Error,Payment]] =
  List(
    Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
    Payment(PaymentAmount(-10), Currency.USD, CheckNumber(2_000_000)),
    Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
    Payment(PaymentAmount(-20), Currency.EUR, CardType.Visa, CardNumber("005")),
    Payment(PaymentAmount(30),  Currency.EUR),
    Payment(PaymentAmount(-30), Currency.EUR)
  )

val expectedSuccessfulAndUnsuccessfulPaymentValidations
    : List[Validation[Error,Payment]] =
  List(
    Success(Payment.unsafe(PaymentAmount.unsafe(10.0),
                           Currency.USD,
                           PaymentMethod.Check(CheckNumber.unsafe(15)))),
    Failure(Error(List("PaymentAmount cannot be negative: -10.0",
                  "CheckNumber cannot be greater than 1,000,000: 2000000"))),
    Success(Payment.unsafe(PaymentAmount.unsafe(20.0),
                           Currency.EUR,
                           PaymentMethod.Card(
                             CreditCardInfo(CardType.Visa,
                             CardNumber.unsafe("123"))))),
    Failure(Error(List("PaymentAmount cannot be negative: -20.0",
                  "CardNumber cannot be less than 111111: 005"))),
    Success(Payment.unsafe(PaymentAmount.unsafe(30.0),
                           Currency.EUR,
                           PaymentMethod.Cash)),
    Failure(Error(List("PaymentAmount cannot be negative: -30.0")))
  )
```

```scala
val successfulAndUnsuccessfulPaymentValidations
    : List[Validation[String,Payment]] =
 List(
    Payment(PaymentAmount(10),  Currency.USD, CheckNumber(15)),
    Payment(PaymentAmount(-10), Currency.USD, CheckNumber(2_000_000)),
    Payment(PaymentAmount(20),  Currency.EUR, CardType.Visa, CardNumber("123")),
    Payment(PaymentAmount(-20), Currency.EUR, CardType.Visa, CardNumber("005")),
    Payment(PaymentAmount(30),  Currency.EUR),
    Payment(PaymentAmount(-30), Currency.EUR)
  )

val expectedSuccessfulAndUnsuccessfulPaymentValidations
    : List[Validation[String,Payment]] =
  List(
    Success(Payment.unsafe(PaymentAmount.unsafe(10.0),
                           Currency.USD,
                           PaymentMethod.Check(CheckNumber.unsafe(15)))),
    Failure("PaymentAmount cannot be negative: -10.0",
            Vector("CheckNumber cannot be greater than 1,000,000: 2000000")),
    Success(Payment.unsafe(PaymentAmount.unsafe(20.0),
                           Currency.EUR,
                           PaymentMethod.Card(
                             CreditCardInfo(CardType.Visa,
                             CardNumber.unsafe("123"))))),
    Failure("PaymentAmount cannot be negative: -20.0",
            Vector("CardNumber cannot be less than 111111: 005")),
    Success(Payment.unsafe(PaymentAmount.unsafe(30.0),
                           Currency.EUR,
                           PaymentMethod.Cash)),
    Failure("PaymentAmount cannot be negative: -30.0")
  )
```

```scala
assert( successfulAndUnsuccessfulPaymentValidations
        == expectedSuccessfulAndUnsuccessfulPaymentValidations)
```

```
// turn a list of partly failed payment validations into
// a failed validation of a list of payments
val failedValidationOfPayments : Validation[Error,List[Payment]] =
  listTraverse.sequence(successfulAndUnsuccessfulPaymentValidations)(errorValidationApplicative)

val expectedFailedValidationOfPayments : Validation[Error,List[Payment]] =
  Failure(Error(List("PaymentAmount cannot be negative: -10.0",
                     "CheckNumber cannot be greater than 1,000,000: 2000000",
                     "PaymentAmount cannot be negative: -20.0",
                     "CardNumber cannot be less than 111111: 005",
                     "PaymentAmount cannot be negative: -30.0")))
```

```
assert(failedValidationOfPayments
       == expectedFailedValidationOfPayments)
```

```
// turn a list of partly failed payment validations into
// a failed validation of a list of payments
val failedValidationOfPayments : Validation[String,List[Payment]] =
  listTraverse.sequence(successfulAndUnsuccessfulPaymentValidations)(errorValidationApplicative)

val expectedFailedValidationOfPayments : Validation[String,List[Payment]] =
  Failure(
    "PaymentAmount cannot be negative: -10.0",
    Vector("CheckNumber cannot be greater than 1,000,000: 2000000",
           "PaymentAmount cannot be negative: -20.0",
           "CardNumber cannot be less than 111111: 005",
           "PaymentAmount cannot be negative: -30.0")
  )
```

to be continued