
Growing a DSL
Typelevel Summit, Oslo 2016
Jan Ouwens

About me

Jan Ouwens

CODE.STAR

Programming Scala since ±2011

also: EqualsVerifier

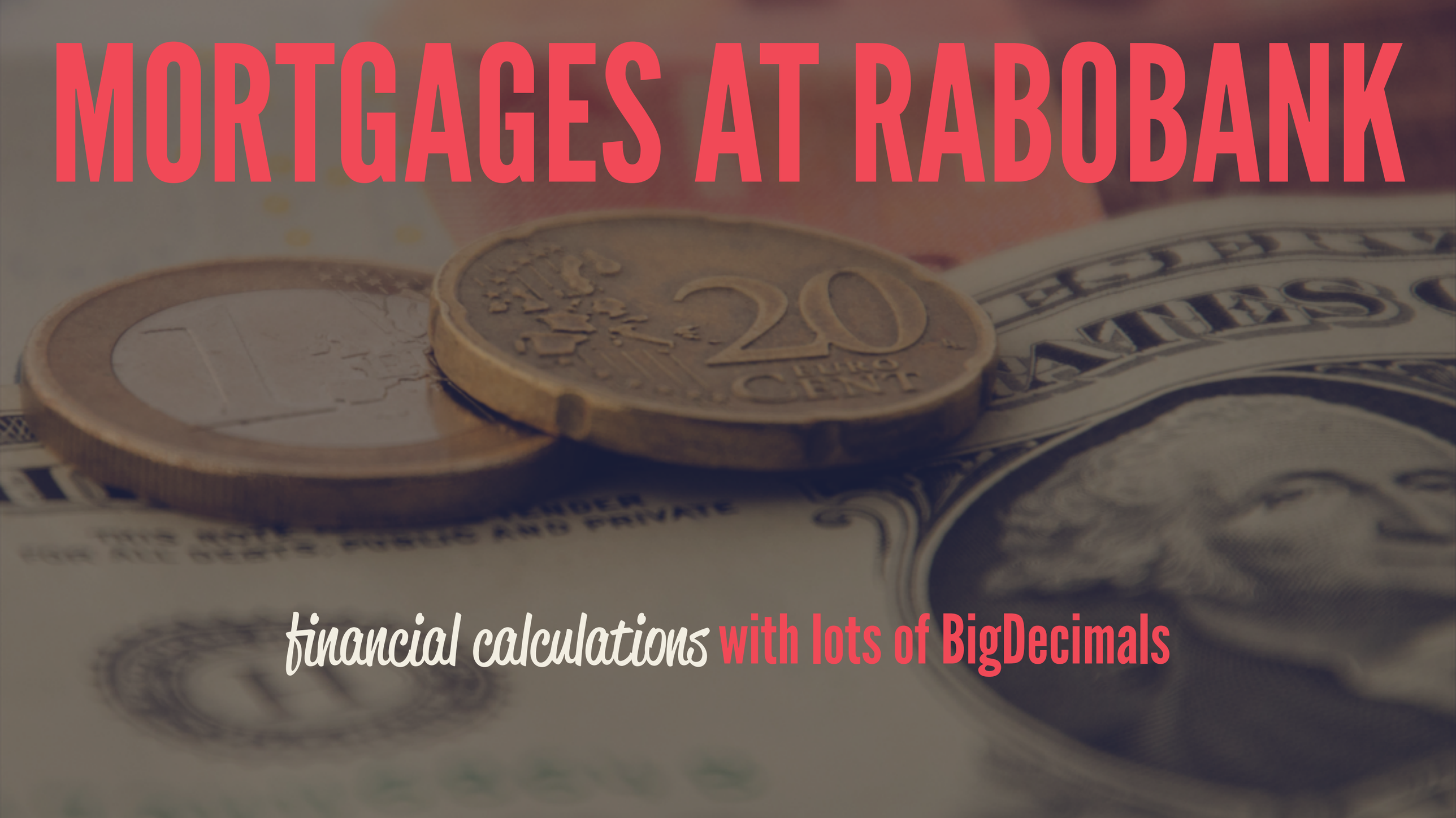
Twitter: @jqno

- 0. Introduction
  - 1. The DSL
  - 2. Evaluation

0. Introduction


Rabobank
Big bank in the Netherlands
Operates internationally
Largest supplier of mortgages

MORTGAGES AT RABOBANK



financial calculations with lots of BigDecimals

REQUIREMENTS

- Easy to use for developers
- Readable for the business
  - Correct

Growing A DSL

Exploring the domain

Exploring Scala

1. The DSL

THE ORIGINAL *Java* IMPLEMENTATION

```java
public class Annuity {
    private static final BigDecimal interest = new BigDecimal(6);
    private static final int numberOfTerms = 30 * 12;
    private static final BigDecimal costOfLiving = new BigDecimal(25);

    private final MathContext MC = MathContext.DECIMAL128;

    public BigDecimal calculateAnnuity(BigDecimal capital, BigDecimal interest, int numberOfTerms, int termCode) {
        BigDecimal monthlyInterest = interest.divide(new BigDecimal(1200).divide(new BigDecimal(termCode), MC), MC);
        BigDecimal pow = BigDecimal.ONE.add(monthlyInterest).pow(-numberOfTerms, MC);
        BigDecimal annuityFactor = monthlyInterest.divide(BigDecimal.ONE.subtract(pow), MC);
        return capital.multiply(annuityFactor);
    }

    public boolean canIAffordThisHouse(BigDecimal price, List<BigDecimal> incomes) {
        BigDecimal burden = calculateAnnuity(price, interest, numberOfTerms, 1);
        BigDecimal totalIncome = new BigDecimal(0);
        for (BigDecimal b : incomes) {
            totalIncome = totalIncome.add(b);
        }
        BigDecimal estimate = (costOfLiving.divide(new BigDecimal(100), MC)).multiply(totalIncome);
        return burden.compareTo(estimate) < 0;
    }
}
```

STEP 1: Scala

```
object Annuity {
  val interest = BigDecimal(6)
  val numberOfTerms = 30 * 12
  val costOfLiving = BigDecimal(25)

  def calculateAnnuity(capital: BigDecimal, interest: BigDecimal,
                       numberOfTerms: Int, termCode: Int): BigDecimal = {
    val monthlyInterestFactor = interest / (1200 / termCode)
    val pow = (1 + monthlyInterestFactor).pow(-numberOfTerms)
    val annuityFactor = monthlyInterestFactor / (1 - pow)
    capital * annuityFactor
  }

  def canIAffordThisHouse(price: BigDecimal, incomes: List[BigDecimal]): Boolean = {
    val burden = calculateAnnuity(price, interest, numberOfTerms, 1)
    val totalIncome = incomes.sum
    burden < (costOfLiving / 100) * totalIncome
  }
}
```

A CASE CLASS

```
case class Amount(value: BigDecimal)

val a = Amount(10)
```

READABILITY FOR THE *business*?

```
implicit class IntToAmount(value: Int) {
  def euro: Amount = Amount(value)
}

val a = 10.euro
```

MAKE THE CONSTRUCTOR PRIVATE

```
case class Amount private[dsl] (value: BigDecimal)
```

Keep it visible in package **dsl** *for internal use*

LET'S DO SOME ARITHMETIC

```
case class Amount(value: BigDecimal) {
  def + (n: Amount): Amount = Amount(value + n.value)
  def - (n: Amount): Amount = Amount(value - n.value)
}

10.euro + 20.euro should be (30.euro)
```

HOW ABOUT MULTIPLICATION?

Does it make sense to multiply amounts?

$€ 10 * € 2 = €^2 20$

This makes more sense:

$€ 10 * 2 = € 20$

COMMUTATIVITY

```
case class Amount(value: BigDecimal) {
  def * (n: BigDecimal): Amount = Amount(value * n)
}

implicit class BigDecimalToAmount(value: BigDecimal) {
  // delegates to Amount!
  def * (a: Amount): Amount = a * value
}
```

NOW WE CAN DO THIS:

```
10.euro * 2 should be 20.euro
```

```
2 * 10.euro should be 20.euro
```

A NEW CONCEPT

```
case class Percentage private[dsl] (p: BigDecimal)
```

WITH A NEW CONSTRUCTOR

```
implicit class ToPercentage(value: BigDecimal) {
  def % : Percentage = Percentage(value)
}

val five = BigDecimal(5)
five.% should be (Percentage(5))
```

WITH A NEW CONSTRUCTOR

```
implicit class IntToPercentage(value: BigDecimal) {
  def percent: Percentage = Percentage(value)
}

val five = BigDecimal(5)
five.percent should be (Percentage(5))
```

MULTIPLICATION

What we want:

```
10.percent * 50.euro should be (5.euro)

50.euro * 10.percent should be (5.euro)
```

FIRST IMPLEMENTATION

```
case class Amount(value: BigDecimal) {
  // Delegates to Percentage
  def * (p: Percentage): Amount = p * this
}
case class Percentage(bd: BigDecimal) {
  // Delegates back to Amount
  def * (a: Amount): Amount = a * (bd / 100)
}
```

TO BE CONTINUED...

ANOTHER LOOK AT THE CODE

```
val interest = BigDecimal(6)
val numberOfTerms = 30 * 12
val costOfLiving = BigDecimal(25)

def calculateAnnuity(capital: BigDecimal, interest: BigDecimal,
                     numberOfTerms: Int, termCode: Int): BigDecimal = ...
```

VS

```
val interest = 6.percent
val numberOfTerms = 30.years
val costOfLiving = 25.percent

def calculateAnnuity(capital: Amount, interest: Percentage,
                     numberOfTerms: Int, termCode: Int): Amount = ...
```

BUT WE *lost* SOMETHING

```
val totalIncome = incomes.sum
```

VS

```
val totalIncome = incomes.reduce(_ + _)
```

Let's fix that

WHY DOES SUM ONLY WORK ON *certain types?*

Taken from `scala.collection.immutable.List`:

```
class List[A] {
  def sum[B >: A](implicit num: Numeric[B]): B =
    foldLeft(num.zero)(num.plus)
}
```

SO, WHAT'S NUMERIC?

```
trait Numeric[T] extends Ordering[T] {
  def plus(x: T, y: T): T
  def minus(x: T, y: T): T
  def times(x: T, y: T): T
  def negate(x: T): T

  def zero = fromInt(0)
  def one = fromInt(1)

  // more stuff
}
```

Let's create one for AMOUNT

```
implicit val numericAmount = new Numeric[Amount] {

  override def plus(x: Amount, y: Amount): Amount = x + y



  // the rest
}
```

A GOTCHA

```
implicit val numericAmount = new Numeric[Amount] {

  override def plus(x: Amount, y: Amount): Amount = x + y

  override def times(x: Amount, y: Amount): Amount =
    throw new UnsupportedOperationException("😰")

  // the rest
}
```

NOW WE CAN DO THIS AGAIN

```
implicit val numericAmount: Numeric[Amount] = ...

val totalIncome = incomes.sum
```

THIS IS HOW WE DISCOVERED

the type class

...AND NOW, THE CONCLUSION

Amount AND Percentage ARE NOW TIGHTLY COUPLED

```
case class Percentage(p: BigDecimal) {
  def * (a: Amount): Amount = a * (p / 100)
}
case class Amount(value: BigDecimal) {
  def * (p: Percentage): Amount = p * value
  def * (n: BigDecimal): Amount = Amount(value * n)
}
```

WE CAN DO BETTER

LET'S DEFINE A TRAIT

```
trait Quantity[T] {
  def multiply(n: T, m: BigDecimal): T
  // Also: plus, minus, div
}
```

AND...

```
implicit object QuantityAmount extends Quantity[Amount] {
  override def multiply(n: Amount, m: BigDecimal) = n * m
}
```

Now we can write:

```
case class Percentage(p: BigDecimal) {
  def * [T](n: T)(implicit ev: Quantity[T]): T =
                                ev.multiply(n, p / 100)
}
```

There's another way to write this:

```
def * [T](n: T)(implicit ev: Quantity[T]): T =
                                ev.multiply(n, p / 100)

def * [T: Quantity](n: T): T =
        implicitly[Quantity[T]].multiply(n, p / 100)
```

COMMUTATIVITY AGAIN

What we want:

```
10.percent * 50.euro == 5.euro   // works

50.euro * 10.percent == 5.euro   // doesn't work yet
```

AN implicit CLASS ON A type CLASS

```
implicit class QuantityWithPercentage[T: Quantity](value: T) {
  def * (p: Percentage): T = p * value
}
```

deletegates to Percentage

NOW, IT WORKS!

```
10.percent * 10.euro == 1.euro
10.euro * 10.percent == 1.euro
```

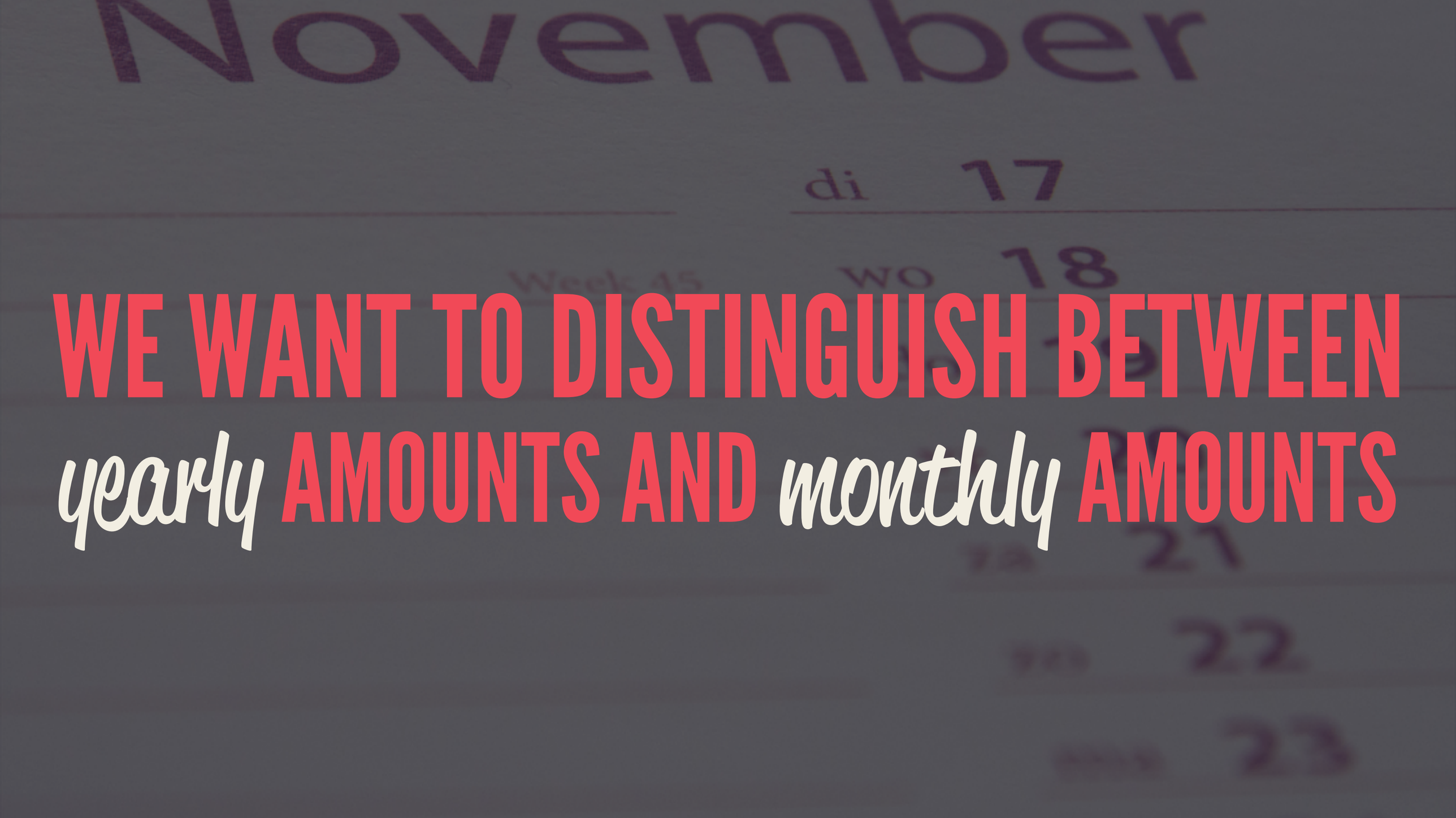
TO SUMMARISE:

- We have a `*` overload in `Percentage`
- We have a `*` overload in an implicit `Quantity`
- These 2 methods handle *all* combinations commutatively

Easy, right?

Time to use another

COOL TRICK!



WE WANT TO DISTINGUISH BETWEEN *yearly* AMOUNTS AND *monthly* AMOUNTS

Let's define a PERIOD

```
sealed trait Period
case object Month extends Period
case object Year extends Period
```

Let's also define a "PER"

```
case class Per[T, P <: Period](value: T, period: P) {

  def * (n: BigDecimal)(implicit ev: Quantity[T]): Per[A, P] =
                        Per(ev.multiply(value, n), period)
}

val p = Per(10.euro, Month)
```

READABILITY

```
val p = Per(10.euro, Month)
p: Per[Amount,Month] = Per(10.euro,Month)
```

We want the type to read `Per[Amount, Month]`

Not `Per[Amount, Month.type]`

LET'S TRY THAT AGAIN

```
sealed trait Period
class Month extends Period
class Year extends Period
val Month = new Month
val Year = new Year

scala> val p = Per(10.euro, Month)
p: Per[Amount,Month] = Per(10.euro,Month)
```

THAT'S BETTER

AND NOW, *la pièce de résistance*

```
val p: Per[Amount, Month]
```

IS EQUIVALENT TO

```
val p: Amount Per Month
```

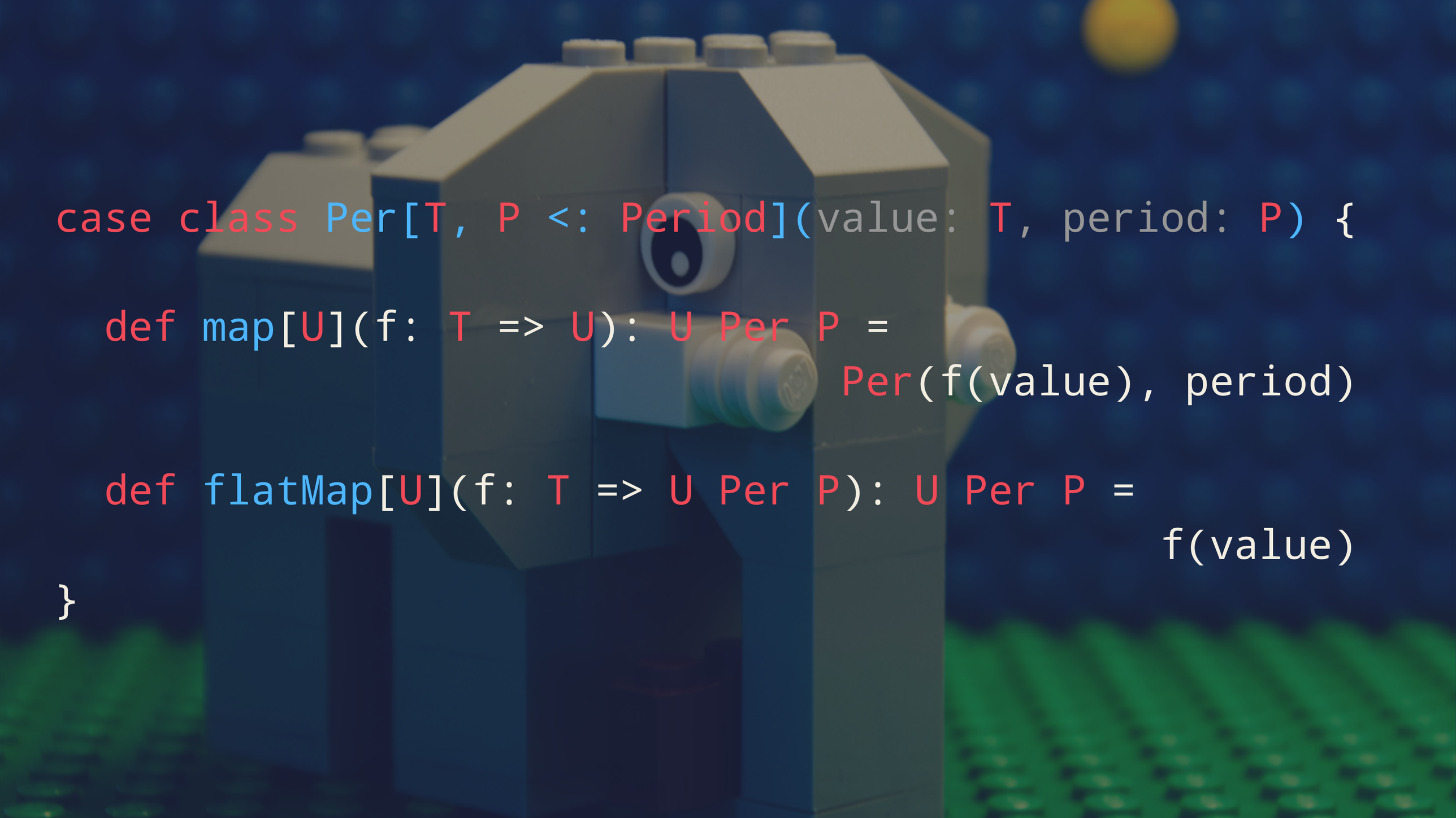
LET'S ADD SOME MORE IMPLICIT SUGAR

```
implicit class ToPer[T](value: T) {
  def per[P <: Period](p: P): T Per P = Per(value, p)
}
```

```
val p: Amount Per Year = 10.euro per Year
```

MAKING *Per* MORE USEFUL

- Access values inside Per
- Maybe in a *for* expression?



```
case class Per[T, P <: Period](value: T, period: P) {

  def map[U](f: T => U): U Per P =
                                Per(f(value), period)

  def flatMap[U](f: T => U Per P): U Per P =
                                       f(value)
}
```

PER STILL NEEDS A NUMERIC AND A QUANTITY

An exercise for the reader 😉

A LOOK AT SOME CODE

```
def canIAffordThisHouse(price: Amount,
        incomes: List[Amount]): Boolean = ...
```

VS

```
def canIAffordThisHouse(price: Amount,
        incomes: List[Amount Per Month]): Boolean = ...
```

SUMMARY

- Implicit conversions → constructors
- Implicit objects → type classes
- Infix notation for types → pretty

2. Evaluation

A *final* LOOK AT THE CODE

FROM Java...

```java
public class Annuity {
    private static final BigDecimal interest = new BigDecimal(6);
    private static final int numberOfTerms = 30 * 12;
    private static final BigDecimal costOfLiving = new BigDecimal(25);

    private final MathContext MC = MathContext.DECIMAL128;

    public BigDecimal calculateAnnuity(BigDecimal capital, BigDecimal interest, int numberOfTerms, int termCode) {
        BigDecimal monthlyInterest = interest.divide(new BigDecimal(1200).divide(new BigDecimal(termCode), MC), MC);
        BigDecimal pow = BigDecimal.ONE.add(monthlyInterest).pow(-numberOfTerms, MC);
        BigDecimal annuityFactor = monthlyInterest.divide(BigDecimal.ONE.subtract(pow), MC);
        return capital.multiply(annuityFactor);
    }

    public boolean canIAffordThisHouse(BigDecimal price, List<BigDecimal> incomes) {
        BigDecimal burden = calculateAnnuity(price, interest, numberOfTerms, 1);
        BigDecimal totalIncome = new BigDecimal(0);
        for (BigDecimal b : incomes) {
            totalIncome = totalIncome.add(b);
        }
        BigDecimal estimate = (costOfLiving.divide(new BigDecimal(100), MC)).multiply(totalIncome);
        return burden.compareTo(estimate) < 0;
    }
}
```

... TO Scala ...

```
object Annuity {
  val interest = BigDecimal(6)
  val numberOfTerms = 30 * 12
  val costOfLiving = BigDecimal(25)

  def calculateAnnuity(capital: BigDecimal, interest: BigDecimal,
                       numberOfTerms: Int, termCode: Int): BigDecimal = {
    val monthlyInterestFactor = interest / (1200 / termCode)
    val pow = (1 + monthlyInterestFactor).pow(-numberOfTerms)
    val annuityFactor = monthlyInterestFactor / (1 - pow)
    capital * annuityFactor
  }

  def canIAffordThisHouse(price: BigDecimal, incomes: List[BigDecimal]): Boolean = {
    val burden = calculateAnnuity(price, interest, numberOfTerms, 1)
    val totalIncome = incomes.sum
    burden < (costOfLiving / 100) * totalIncome
  }
}
```

... TO typesafe SCALA

```
object Annuity {
  val interest = 6.percent
  val numberOfTerms = 30.years
  val costOfLiving = 25.percent

  def calculateAnnuity[P <: Period](capital: Amount, interest: Percentage,
                                    numberOfTerms: Int, period: P): Amount Per P = {
    val monthlyInterestFactor = interest / period.frequency
    val pow = (1 + monthlyInterestFactor).pow(-numberOfTerms)
    val annuityFactor = monthlyInterestFactor / (1 - pow)
    (capital * annuityFactor) per period
  }

  def canIAffordThisHouse(price: Amount, incomes: List[Amount Per Month]): Boolean = {
    val burden = calculateAnnuity(price, interest, numberOfTerms, Month)
    val totalIncome = incomes.sum
    burden < costOfLiving * totalIncome
  }
}
```

THE DSL CONTAINS *much, much* MORE,

FOR INSTANCE, AUTOMATIC CONVERSIONS BETWEEN FISCAL '*boxes*',

BUT I'M ALMOST OUT OF TIME

DID WE ACHIEVE OUR GOALS?

- Easy to use for developers
- Readable for the business
- Correct

WE STILL HAD TO CONVINCE PEOPLE

Can't we do this in Java too?

- Lose type safety
- Lose readability

But

It inspired a new project

A *rule engine* THAT LOOKS LIKE THIS

```
Gegeven (NHG is true) Bereken
    MeerwaardeVerbouwingNHG is KostenVerbouwing * MeerwaardeVerbouwingsPercentageNHG
,
Gegeven (NHG is false) Bereken
    MeerwaardeVerbouwingGeenNHG is KostenVerbouwing * MeerwaardeVerbouwingsPercentageGeenNHG
,
Gegeven (altijd) Bereken
    MarktwaardeVoorVerbouwing is Koopsom - KostenRoerendeGoederen en
    MarktwaardeNaVerbouwing is MarktwaardeVoorVerbouwing + MeerwaardeVerbouwing en
    MeerwaardeVerbouwing is eerste(MeerwaardeVerbouwingNHG, MeerwaardeVerbouwingGeenNHG) en
    Marktwaarde is eerste (MarktwaardeNaVerbouwing, TaxatieWaarde, MarktwaardeVoorVerbouwing)
```

NOW THE BUSINESS ANALYSTS *write* CODE

We had to teach them Git though 😅

IT'S *open source*!

https://github.com/scala-rules/finance-dsl

Thank you

QUESTIONS?

Twitter: @jqno