

24 november 2025

PAL-sessie Declaratieve Talen

# Sessie II: typeclasses

Zijn er vragen vanuit de oefenzittingen?

# I. Typeclasses

- Wat zijn typeclasses?
- Hoe pas je een typeclass toe?
- Nuttige typeclasses
  - Functor
  - Foldable
  - Traversable
  - Monad
    - Nuttige Monad: State
- Waar kun je documentatie vinden?

## II. De opdracht voor vandaag

## III. Mogelijke oplossingen

# I. Typeclasses

# Wat zijn typeclasses?

- Stel dat je verschillende types hebt (Int, Point, List, Tree...)
- Je wilt een generieke `show :: a → String` die elk van deze types omzet naar een printbare string
- Of je wilt een generieke `(=) :: a → a → Bool` die voor elk type werkt

# Wat zijn typeclasses?

Een typeclass is een set van functies met regels die een type kan implementeren

- Gelijkaardige concept:
  - Interfaces in OGP

```
public interface Equals {  
    public boolean equals(Object other);  
}
```

```
class Eq t where  
    (==) :: t → t → Bool  
    (/=) :: t → t → Bool
```

# Hoe pas je een typeclass toe?

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

```
instance Eq Color where
  Red    == Red    = True
  Green  == Green  = True
  Blue   == Blue   = True
  _      == _      = False
```

OF

```
class Eq t where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool
```

```
instance Eq Color where
  Red    /= Red    = False
  Green  /= Blue   = False
  Blue   /= Blue   = False
  _      /= _      = True
```

## Nuttige typeclasses: `Functor (<$>)`

Een type `f` is een `Functor` indien het een functie `fmap` voorziet die, gegeven eenderwelke types `a` en `b`, je een functie van type `(a → b)` laat toepassen om `f a` naar `f b` te veranderen, waarbij de structuur van `f` behouden blijft.

```
class Functor f where
    fmap :: (a → b) → f a → f b
```

```
data MyList a = Empty | Cons a (MyList a)
```

```
instance Functor MyList where
```

```
    fmap _ Empty = Empty
```

```
    fmap f (Cons head tail) = Cons (f head) (fmap f tail)
```

```
>> fmap length (Cons "Hello " (Cons "World" (Cons "!" Empty)))
(Cons 6 (Cons 5) (Cons 1 Empty))
```



## Nuttige typeclasses: Foldable

Een type `t` is `Foldable` indien het een manier voorziet om alle elementen in een structuur van links naar rechts te “samenvouwen” tot één waarde. Dit gebeurt met een beginwaarde en een functie die het volgende element met de huidige accumulator combineert.

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr   :: (a -> b -> b) -> b -> t a -> b
```

## Nuttige typeclasses: Foldable

```
data MyList a = Empty | a (MyList a)
```

```
instance Foldable MyList where
```

```
    foldr _ z Empty = z
```

```
    foldr f z (Cons head tail) = f head (foldr f z tail)
```

```
-- OF
```

```
instance Foldable MyList where
```

```
    foldMap _ Empty = mempty
```

```
    foldMap f (Cons head tail) = f head <math>\diamond</math> foldMap f tail
```

```
>> foldr (++) "" (Cons "Hello " (Cons "World" (Cons "!" Empty)))  
"Hello World!"
```

```
>> foldMap id (Cons "Hello " (Cons "World" (Cons "!" Empty)))  
"Hello World!"
```

## Nuttige typeclasses: Foldable

Set dat het type `t` behoort tot de `Foldable` typeclass, dan kan je ook de volgende functies gebruiken:

- `fold` `:: Monoid m => t m -> m`
- `foldl` `:: (b -> a -> b) -> b -> t a -> b`
- `toList` `:: t a -> [a]`
- `null` `:: t a -> Bool`
- `length` `:: t a -> Int`
- `elem` `:: Eq a => a -> t a -> Bool`
- `maximum` `:: Ord a => t a -> a`
- `minimum` `:: Ord a => t a -> a`
- `sum` `:: Num a => t a -> a`
- `product` `:: Num a => t a -> a`

## Nuttige typeclasses: `Traversable` ( `<*>` )

Een type `t` is `Traversable` indien het elementen één voor één kan “doorlopen” met een effectvolle functie, en zo een gestructureerd effect `f (t b)` oplevert.

```
class (Functor t, Foldable t) => Traversable t where
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
```

```
data MyList a = Empty | Cons a (MyList a)
```

```
instance Traversable MyList where
  traverse _ Empty = pure Empty
  traverse toApp (Cons head tail) = Cons <$> toApp head <*> traverse toApp tail
```

## Nuttige typeclasses: Traversable voorbeelden

```
safeNonEmpty :: String → Maybe String
```

```
safeNonEmpty "" = Nothing
```

```
safeNonEmpty s = Just s
```

```
>> traverse safeNonEmpty (Cons "Hello " (Cons "World" (Cons "!" Empty)))  
Just (Cons "Hello " (Cons "World" (Cons "!" Empty)))
```

```
>> traverse safeNonEmpty (Cons "Hello " (Cons "" (Cons "!" Empty)))  
Nothing
```

- Alle strings niet leeg ⇒ Just (Cons ...)
- Minstens 1 lege string ⇒ Nothing

## Nuttige typeclasses: Traversable voorbeelden

```
dup :: String → [String]
dup s = [s, s ++ s]
```

```
>> traverse dup (Cons "Hello" (Cons "World" Empty))
[Cons "Hello" (Cons "World" Empty),
 Cons "Hello" (Cons "WorldWorld" Empty),
 Cons "HelloHello" (Cons "World" Empty),
 Cons "HelloHello" (Cons "WorldWorld" Empty)]
```

traverse + lijst → Cartesiaans product van mogelijkheden

## Nuttige typeclasses: Traversable voorbeelden

```
askQuestion :: String → IO String  
askQuestion q = putStrLn q >> getLine
```

```
>> traverse askQuestion ["What are your names?", "Favourite language?"]
```

```
What are your names?  
Simeon and Jonas  
Favourite language?  
Haskell
```

```
["Simeon and Jonas", "Haskell"]
```

## Nuttige typeclasses: `Monad`

Een type `m` is een `Monad` indien het een manier voorziet om berekeningen met context na elkaar uit te voeren. Het resultaat van de eerste stap (in context `m`) kan de volgende stap bepalen.

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a    -- vaak: return = pure
```

```
data Maybe a = Just a | Nothing
```

```
instance Monad Maybe where
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
  return  = Just
```



## Nuttige typeclasses: **Monad** voorbeeld

```
data User      = User { address :: Maybe Address }
data Address   = Address { city :: Maybe String }

getCity :: User → Maybe String
getCity user =
    address user >>= \addr →
    city addr
-- OF met do-notatie (syntactic sugar)
getCity :: User → Maybe String
getCity user = do
    addr ← address user
    city addr
```

Als `address user` of `city addr` **Nothing** is, dan wordt het geheel **Nothing**

## Nuttige monad: State

De `State`-monad modelleert toestandsvolle berekeningen op een zuiver functionele manier.

Intuïtief: een `State s a` is een berekening die

- een huidige toestand `s` binnenkrijgt,
- een resultaat `a` en een nieuwe toestand `s` teruggeeft.

```
newtype State s a =  
  State { runState :: s -> (a, s) }
```

Monad-idee:

- `return x` : verander de toestand niet, maar geef `x` terug
- `m >>= f` : voer `m` uit, gebruik het resultaat om `f` te kiezen, en geef de nieuwe toestand door

## Nuttige monad: **State** voorbeelden

```
import Control.Monad.State

data Door = Open | Closed
          deriving (Show)

step :: String → State Door ()
step "open"   = put Open
step "close"  = put Closed
step "toggle" = do
    state ← get
    case state of
        Open   → put Closed
        Closed → put Open
step _ = return ()
```

```
>> runState (step "open") Closed
((), Open)
>> runState (step "open") Open
((), Open)

>> runState (step "close") Closed
((), Closed)
>> runState (step "close") Open
((), Closed)

>> runState (step "toggle") Closed
((), Open)
>> runState (step "toggle") Open
((), Closed)
```

```
import Control.Monad.State

step :: String -> State Int Int
step "inc" = do
    modify (+1)           -- teller verhogen
    return 1              -- 1 increment geteld
step "dec" = do
    modify (subtract 1)   -- teller verlagen
    return -1             -- 1 decrement geteld
step "reset" = do
    put 0                  -- teller resetten
    return 0               -- gereset
step _ = return 0
```

```
runCounter :: [String] → State Int Int
runCounter cmds = do
    counts ← mapM step cmds    -- [Int] met 0/1/-1 per commando
    return (sum counts)
```

## **II.** De opdracht voor vandaag

# Schattenjacht in het doolhof

```
data Cell
  = Empty
  | Wall
  | Treasure Int      -- Waarde van de schat
  deriving (Show, Eq)

data Grid a = Grid [[a]]
  deriving (Show, Eq)
```

## Schattenjacht in het doolhof: Functor

Voeg `Grid` toe aan de `Functor` class door instance te implementeren, we willen dat `fmap` de functie op elke cel van de grid toepast:

```
instance Functor Grid where  
  fmap ... = ...
```

Definieer een helperfunctie `increaseTreasure` die de waarde van een `Treasure` verhoogt met een gegeven waarde:

```
increaseTreasure :: Cell → Int → Cell  
increaseTreasure ... = ...
```

Gebruik `fmap` en de helperfunctie om alle schatten in de grid te verhogen met 5:

```
increaseAllTreasures :: Grid Cell → Int → Grid Cell  
increaseAllTreasures ... = ...
```



## Schattenjacht in het doolhof: Foldable

Voeg `Grid` toe aan de `Foldable` class door instance te implementeren, we willen dat er over alle elementen van het grid rij per rij wordt gefold:

```
instance Foldable Grid where
  foldMap ... = ... -- Kies 1 van de 2 opties
  foldr ... = ...
```

Gebruik een functie die door de `Foldable` class wordt voorzien naar keuze om `totalTreasure` te implementeren:

```
totalTreasure :: Grid Cell → Int
totalTreasure ... = ...
```

## Schattenjacht in het doolhof: Traversable + Maybe

Voeg `Grid` toe aan de `Traversable` class door instance te implementeren, we willen dat alle elementen van het grid rij per rij worden overlopen:

```
instance Traversable Grid where  
  traverse ... = ...
```

Definieer een helperfunctie `validateCell` die nakijkt of een schat geen negatieve waarde bevat en anders `Nothing` terug geeft:

```
validateCell :: Cell → Maybe Cell  
validateCell ... = ...
```

Gebruik een `traverse` om het hele bord te validaten:

```
validateGrid :: Grid Cell → Maybe (Grid Cell)  
validateGrid ... = ...
```

## Schattenjacht in het doolhof: Traversable + Either

De implementatie met `Maybe` laat ons niet weten waarom het hele grid faalt, dus we gaan nu hetzelfde doen maar met `Either` waarbij `validate` oftewel een `CellError` of een `Cell` terug geeft:

```
data CellError = TreasureTooSmall Int
  derive (Show)
```

Definieer een nieuwe helperfunctie `validateCellE` die een `CellError` teruggeeft (`Left` constructor) of de `Cell` zelf terug geeft (`Right` constructor):

```
validateCellE :: Cell → Either CellError Cell
validateCellE ... = ...
```

Gebruik een `traverse` om het hele bord te validaten:

```
validateGridE :: Grid Cell → Either CellError (Grid Cell)
validateGridE ... = ...
```

# Schattenjacht in het doolhof: `State` monad

We gaan de `State` monad gebruiken om een speler door het doolhof te laten lopen en de verzamelde hoeveelheid treasure bij te houden.

```
data Move = MoveUp | MoveDown | MoveLeft | MoveRight
          deriving (Show, Eq)
```

```
type Pos      = (Int, Int)
type Score    = Int
type PlayerState = (Pos, Score)
```

Schrijf een hulperfunctie `getCell` die safely een cell opvraagt, return `Nothing` als de cel buiten het veld ligt:

```
getCell :: Grid a → Pos → Maybe a
getCell ... = ...
```

Definieer een `step` functie die de speler verplaatst:

```
step :: Grid Cell → Move → State GameState (Maybe Int)
step ... = ...
```

Verwacht gedrag:

- Speler stapt buiten het veld → Return `Nothing`
- Speler stapt naar een muur → Speler beweegt niet
- Speler stapt naar een schat → Speler verplaatst en verzamelt de schat
- Speler stapt naar een leeg vakje → Speler verplaatst

Schrijf een functie die een speler een lijst van stappen in het grid laat volgen:

```
runSteps :: Grid Cell → [Move] → State GameState [Maybe Int]
runSteps ... = ...
```

**Mogelijke oplossingen**

# Schattenjacht in het doolhof: Functor - Oplossing

```
instance Functor Grid where  
    fmap f (Grid rows) = Grid (map (map f) rows)
```

```
increaseTreasure :: Cell → Int → Cell  
increaseTreasure (Treasure n) v = Treasure (n + v)  
increaseTreasure c _ = c
```

```
increaseAllTreasures :: Grid Cell → Int → Grid Cell  
increaseAllTreasures grid v = fmap (`increaseTreasure` v) grid
```

## Schattenjacht in het doolhof: Foldable - Oplossing

```
instance Foldable Grid where
  foldMap f (Grid rows) = foldMap (foldMap f) rows
  -- OF
  foldr f z (Grid rows) = foldr (\row acc → foldr f acc row) z rows
```



```
totalTreasure :: Grid Cell → Int
totalTreasure = foldr (\cell acc → getVal cell + acc) 0
  where
    getVal (Treasure n) = n
    getVal _            = 0
-- OF
totalTreasure = sum . map getVal . toList
  where
    getVal (Treasure n) = n
    getVal _            = 0
```

## Schattenjacht in het doolhof: Traversable + Maybe - Opl.

```
instance Traversable Grid where
  traverse f (Grid rows) = Grid <$> traverse (traverse f) rows
```

```
validateCell :: Cell → Maybe Cell
validateCell Treasure n
  | n ≥ 0      = Just (Treasure n)
  | otherwise = Nothing
validateCell c = Just c
```

```
validateGrid :: Grid Cell → Maybe (Grid Cell)
validateGrid = traverse validateCell
```

## Schattenjacht in het doolhof: Traversable + Either - Opl.

```
data CellError = TreasureTooSmall Int
  derive (Show)
```

```
validateCellE :: Cell → Either Cell
validateCellE Treasure n
  | n ≥ 0    = Right (Treasure n)
  | otherwise = Left  (TreasureTooSmall n)
validateCellE c = Right c
```

```
validateGridE :: Grid Cell → Either CellError (Grid Cell)
validateGridE = traverse validateCellE
```

# Schattenjacht in het doolhof: State monad - Oplossing

```
data Move = MoveUp | MoveDown | MoveLeft | MoveRight
    deriving (Show, Eq)
```

```
type Pos      = (Int, Int)
type Score    = Int
type PlayerState = (Pos, Score)
```

```
getCell :: Grid a → Pos → Maybe a
getCell (Grid rows) (row, col)
    | row < 0                = Nothing
    | row ≥ length rows      = Nothing
    | col < 0                = Nothing
    | col ≥ length (rows !! row) = Nothing
    | otherwise              = Just (rows !! row !! col)
```

```
step :: Grid Cell → Move → State PlayerState (Maybe Int)
step grid dir = do
  ((row, col), score) ← get
  let newPos = case dir of
    MoveUp    → (row - 1, col)
    MoveDown  → (row + 1, col)
    MoveLeft  → (row, col - 1)
    MoveRight → (row, col + 1)
  case getCell grid newPos of
    Nothing → return Nothing
    Just cell → do
      let (pos', score', treasure) = case cell of
        Empty    → (newPos, score, Nothing)
        Wall     → ((row, col), score, Nothing)
        Treasure n → (newPos, score + n, Just n)
      put (pos', score')
      return treasure
```

```
runSteps :: Grid Cell → [Move] → State PlayerState [Maybe Int]
runSteps grid steps = mapM (step grid) steps
```