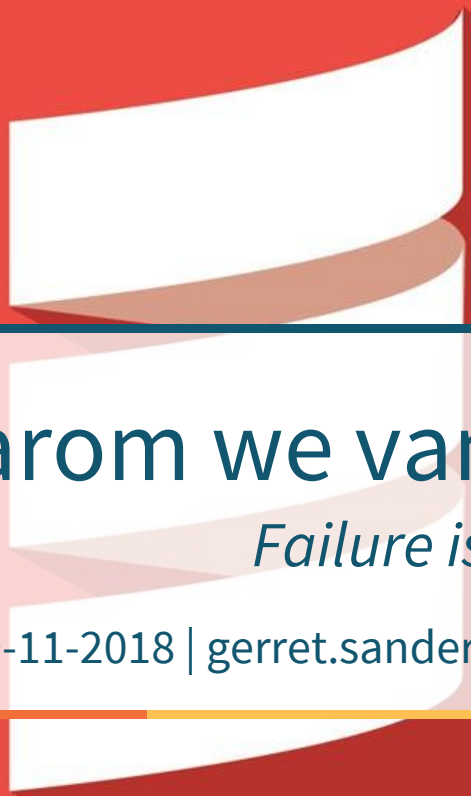
simacan



Waarom we van Scala houden

Failure is not an Option, it's a Try

29-11-2018 | gerret.sanders@simacan.com | @simacanbv

Programma

simacan

- Introductie
- Scala vs. Java
- Scala basic syntax
- Polymorphisme
- Case classes
- Functies
- Collecties framework
- Higher order functions
- Monads
- Pattern matching
- For comprehension
- Implicits
- Het Scala landschap

Introductie

simacan

- **Gerret Sanders**

  sinds 2015:
  Java developer

  sinds 2017:
  Scala developer
  bij Simacan

Simacan...

simacan

- zet verkeer en logistiek op de kaart
- integreert logistieke planningsdata met realtime verkeersinformatie en locatie-updates van de voertuigen
- heeft focus op **performance**, **data-kwaliteit**, en **gebruiksvriendelijkheid**
- biedt 24/7 SaaS dienstverlening
    - Simacan Control Tower: Frontend
    - Simacan Transport Cloud: API’s

Enkele van onze klanten:

postnl

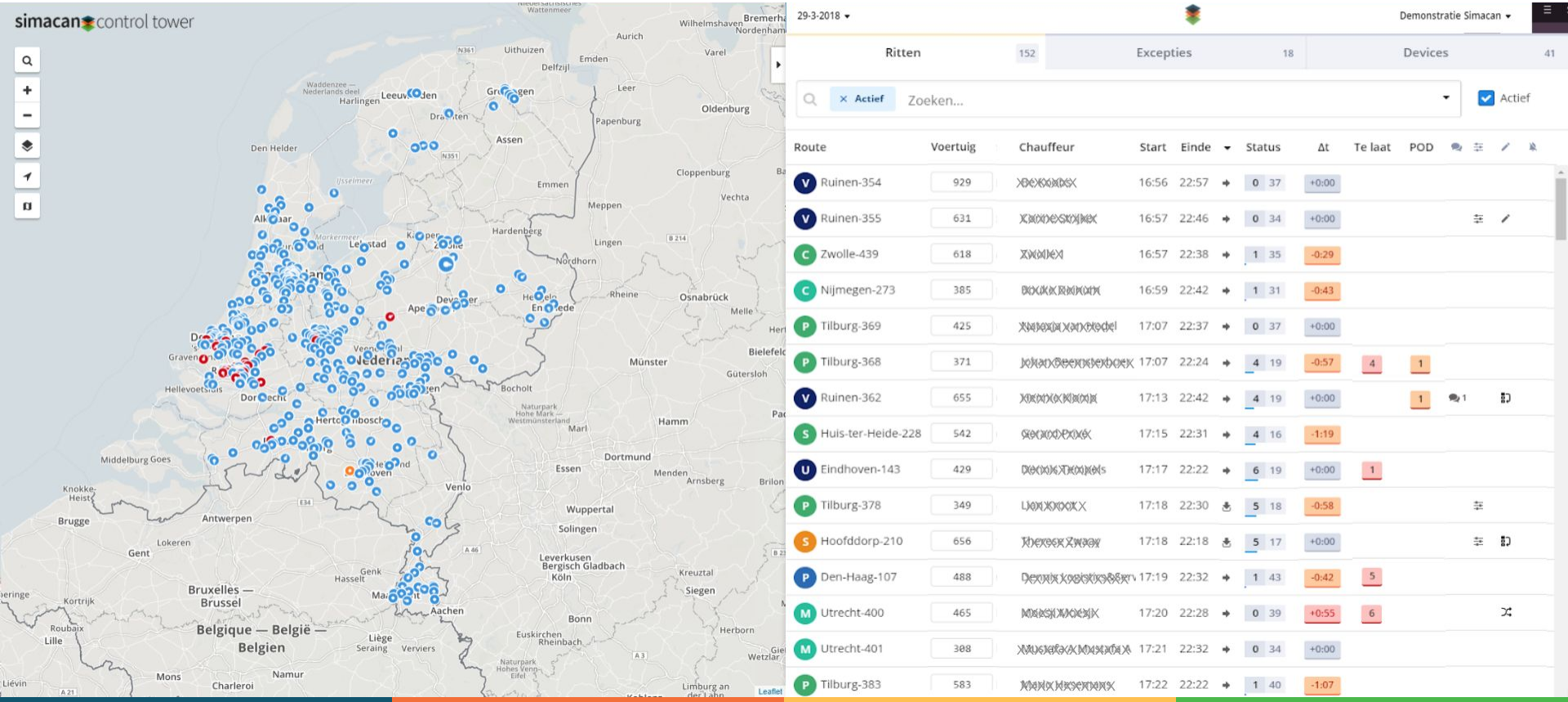
Ahold Delhaize

Port of Rotterdam

anwb

Control Tower - DEMO

simacan



Waarom we van Scala houden

simacan

Een Java POJO:

```java
public class Road {

    private String name;
    private int lengthMeters;
    private String roadType;

    public Road(String name, int lengthMeters, String
roadType) {
        this.name = name;
        this.lengthMeters = lengthMeters;
        this.roadType = roadType;

    }

    public String getName() {
        return name;
    }

    public int getLengthMeters() {
        return lengthMeters;
    }

    public String getRoadType() {
        return roadType;
    }

// ...
```

```java
// ...

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Road)) return false;
        Road other = (Road)o;
        return name.equals(other.name) && lengthMeters ==
other.lengthMeters && roadType.equals(other.roadType);
    }

    @Override
    public int hashCode() {
        // Some sensible implementation
        return 0;
    }

    @Override
    public String toString() {
        return "Road("+ name + "," + lengthMeters + "," +
roadType + ")";
    }

}
```

Waarom we van Scala houden

simacan

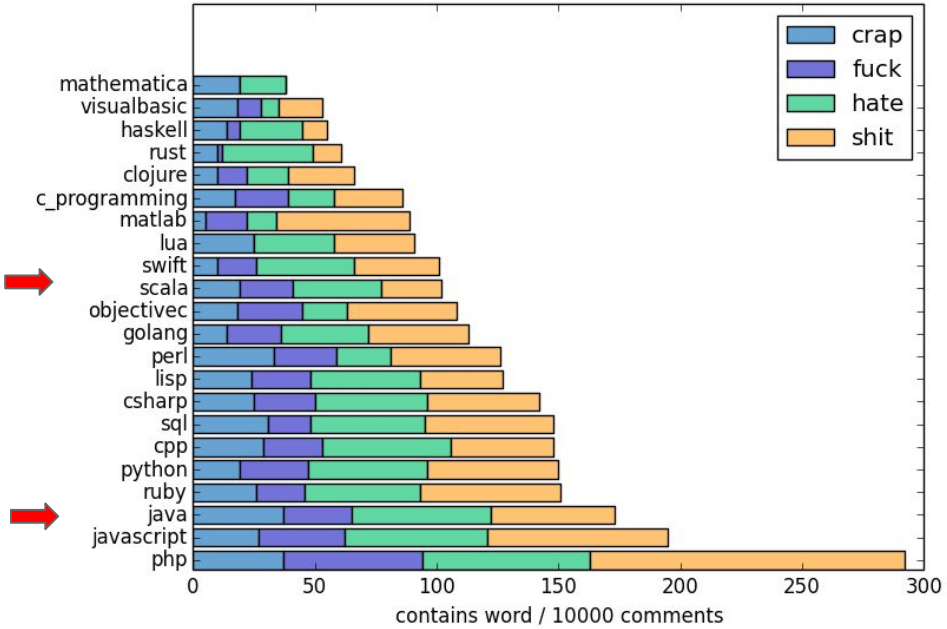
Dezelfde code maar dan in Scala

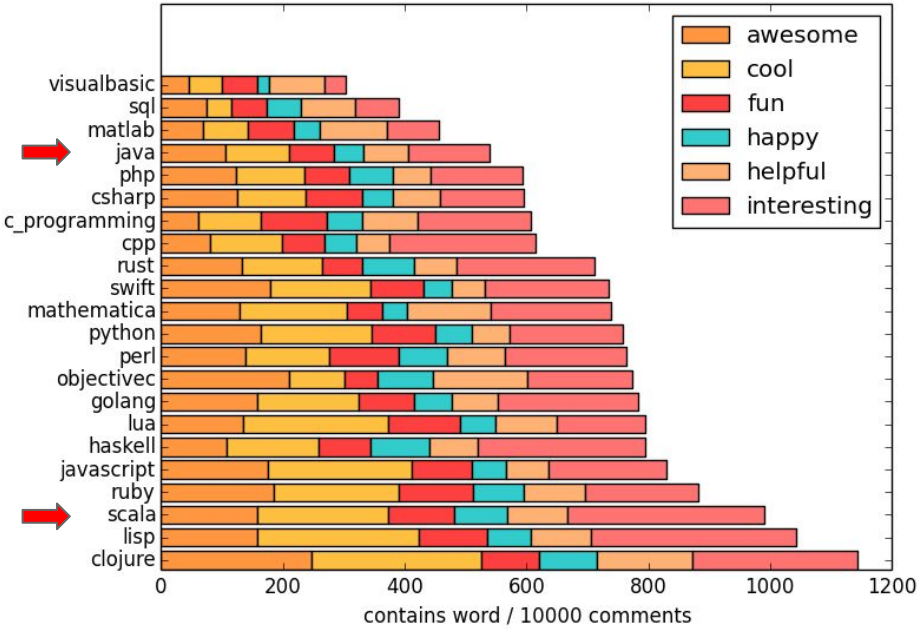
```
case class Road(name: String, lengthMeters: Int, roadType: String)
```

Waarom we van Scala houden

simacan

Onderzoek: Programming language subreddits and their choice of words





https://github.com/Dobiasd/programming-language-subreddits-and-their-choice-of-words

Scala

simacan

Tijdlijn

- 2001: eerste ontwerp door Martin Odersky
  - Schrijver van wat later *javac* werd
  - Bedenker van *Java generics*
- 2004: eerste publieke release
- 2006: Scala 2.0
- Nu: Scala 2.12.7
- 2020: Scala 3.0

Gebruikt door:

- Apache Kafka
- Apache Spark
- Apple, Twitter, LinkedIn,
  Coursera, SoundCloud, Airbnb, Zalando, Duolingo, ...

Java vs Scala

simacan

Java

- Object-georiënteerd
- Statically typed

- Vnl. imperatief
  - Functionele trekjes: Lambdas
- Backwards compatible
  - Dus: Legacy syntax

- Zeer grote community
- A library for everything

Scala

- Object-georiënteerd
- Statically typed
  - Type inference
- Vnl. functioneel
  - Imperatief te gebruiken
- Sommige versies niet backwards compatible
  - Up-to-date syntax
- Grote community
- A library for everything + more
- Volledig compatible met Java-code

Basis syntax

simacan

```
package com.simacan.scalatalk

import java.util.UUID

class Point(val x: Int, val y: Int) {

 val id: UUID = UUID.randomUUID()


 var name = ""


 override def toString: String = {
   id.toString + ", " + x + ", " + y
 }

}
```

<- Class + constructor

<- Immutable value

<- Mutable variable.

<- Functie
Retourneert resultaat van laatste expressie.

Basis syntax

simacan

```
package com.simacan.scalatalk

import java.util.UUID

class Point(x: Int = 0, y: Int = 0) {

 val id: UUID = UUID.randomUUID()

 override def toString: String = {
   s"${id.toString}, $x, $y"
 }

}
```

Default constructor values

String interpolatie

```
val point0 = new Point // (0, 0)
val point1 = new Point(1) // (1, 0)
val point2 = new Point(y = 1) // (0, 1)
val point3 = new Point(1, 1) // (1, 1)
val point4 = new Point(y = 5, x = 2) // (2, 5)
```

Het ‘object’ keyword

simacan

```
object Logger {
 private val warnString = "WARNING:"
 private val errorString = "ERROR:"

 def warn(message: String): Unit = {
   println(s"$warnString $message")
 }

 def error(message: String): Unit = {
   println(s"$errorString $message")
 }

}
```

```
Logger.error("Something went wrong")
```

Singleton object
Java termen: top-level static class

Unit return type: Java ‘void’

Companion object

simacan

```
package com.simacan.scalatalk

import java.util.UUID

class Point(x: Int = 0, y: Int = 0) {
// ...
}

object Point {

 def average(firstPoint: Point, secondPoint: Point): Point = {
   Point(average(firstPoint.x, secondPoint.x), average(firstPoint.y, secondPoint.y))
 }

 private def average(int1: Int, int2: Int) = (int1 + int2) / 2

}
```

Java: static method

Companion object

simacan

```
package com.simacan.scalatalk

import java.util.UUID

class Point(x: Int, y: Int) {
// ...
}

object Point {
 def apply(x: Int = 0, y: Int = 0): Point = {
   new Point(x, y)
 }

}
```

```
val point0 = Point() // (0, 0)
val point1 = Point(1) // (1, 0)
val point2 = Point(y = 1) // (0, 1)
val point3 = Point(1, 1) // (1, 1)
val point4 = Point(y = 5, x = 2) // (2, 5)
```

Default values werken ook voor functie params

apply(): syntactic sugar

Ons eerste Scala programma

simacan

Java:

```java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Hoe herschrijf je deze code in Scala?

Ons eerste Scala programma

simacan

```scala
object Main {

 def main(args: Array[String]): Unit = {
   println("Hello world")
 }

}
```

```scala
object Main extends App {
 println("Hello world")
}
```

Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- **Polymorphisme**
- **Case classes**
- Functies
- Collecties framework
- Higher order functions
- Monads
- Pattern matching
- For comprehension
- Implicits
- Het Scala landschap

Classes en traits

simacan

- Classes hebben we net gezien.
- Abstract classes: hetzelfde als Java abstract classes.
- Traits: het antwoord op Java interfaces.

Trait als soort ‘interface’

simacan

```
abstract class Vehicle {
 def move()
}


trait Engine {
 val power: Int
}

class Car extends Vehicle with Engine {
 override val power: Int = 100

 override def move(): Unit = {
   println(s"Ik rijd met $power kW")
 }
}
```

Trait als mixin

simacan

```
abstract class Vehicle {
 def move()
}

trait Engine {
 val power: Int
}

trait Brakes {
 def brake(): Unit = {
   println("stoppen!")
 }
}

class Car extends Vehicle with Engine with Brakes {
 override val power: Int = 100

 override def move(): Unit = {
   println(s"Ik rijd met $power kW")
 }
}
```

Lijkt op Java's default methods.
Doel is om functionaliteit toe te voegen aan een class.

Case classes

simacan

```
case class Road(
                name: String,
                lengthMeters: Int,
                roadType: String)
```

- Bedoeld om immutable data te modelleren.

Een gewone class +
- Constructor params zijn altijd *immutable* en *public*
- Een *apply()* functie
- *toString(), hashCode()* en *equals()*
- *copy()*
- *Serializable*
- *...*

- Net als gewone classes uit te breiden met extra functies.

```
val road1: Road = Road("Valutaboulevard",
100, "tweebaans")
val road2: Road = Road("Valutaboulevard",
100, "tweebaans")

println(road1 == road2) // true
```

Scala shorthand
voor equals()

```
println(road1.copy(roadType = "fietspad"))
// Road(Valutaboulevard,100,fietspad)
```

Programma

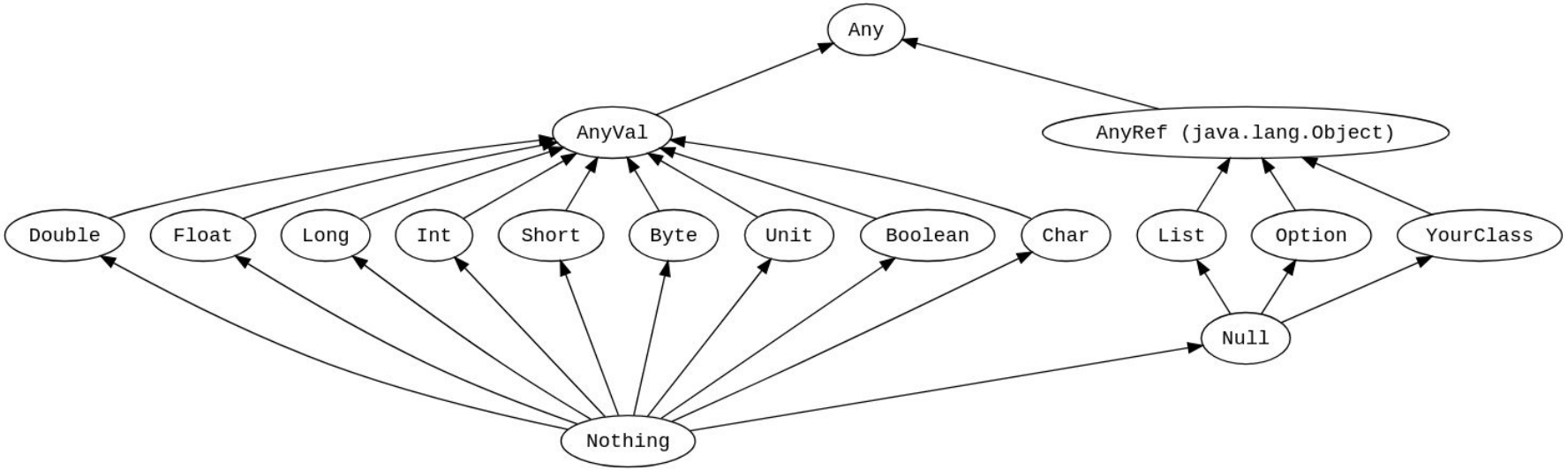
simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- **Functies**
- Collecties framework
- Higher order functions
- Monads
- Pattern matching
- For comprehension
- Implicits
- Het Scala landschap

Waarden, objecten en functies

simacan

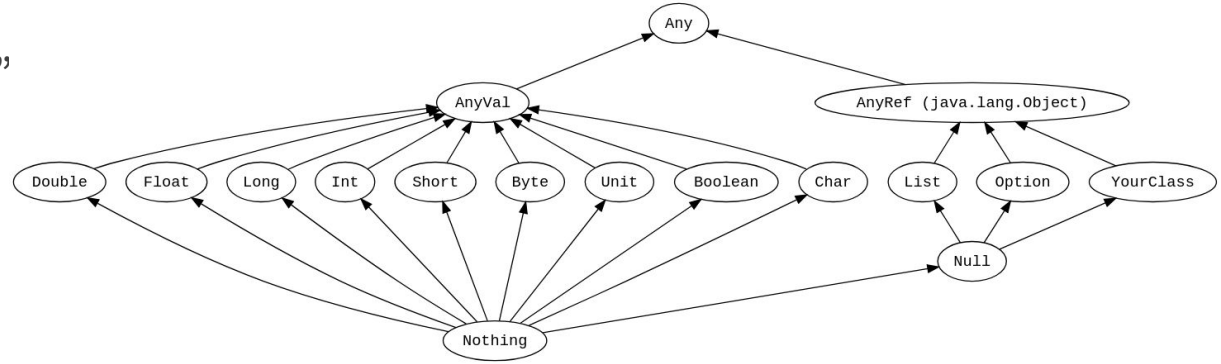
- “Every value is an object”
  - Geen primitieven



Waarden, objecten en functies

simacan

- “Every value is an object”
  - Geen primitieven
- “Every expression returns a value”




```
def someFunction(condition: Boolean) = {

 val result: AnyVal = if (condition) 1.0 else false

 println(result)
}
```

- If/else expression returns a value
- Dus scala heeft niet de *? :* ternary syntax
- Welk type heeft result?

Referential transparency

simacan

- Wiskundig pure functie
  - $f(x) = x^2 + 1$
  - Voor $x = 2$, $f(x) = 5$

```
def f(x: Int): Int = x * x + 1


var y: Int = 5
def g(x: Int): Int = x * y + 1


def h(x: Int): Int = {
  y = 3

  x * x + 1
}
```

Referential transparency

simacan

- Wiskundig pure functie
  - $f(x) = x^2 + 1$
  - Voor $x = 2$, $f(x) = 5$

```
def f(x: Int): Int = x * x + 1


var y: Int = 5
def g(x: Int): Int = x * y + 1


def h(x: Int): Int = {
  println("Hello World")

  x * x + 1
}
```

Speciale types

simacan

```
val nullValue: Null = null


def unitFunction1(): Unit = {
 println("Hello World")
}

def unitFunction2(): Unit = {
 val x = 5
}

private def doSomething() = {
 "Did it!"
}

def unitFunction3(): Unit = {
 doSomething()

 ()
}
```

Null: Java compatibiliteit, niet gebruiken in pure Scala

Unit: Vergelijkbaar met java ‘void’, Singleton object zonder data

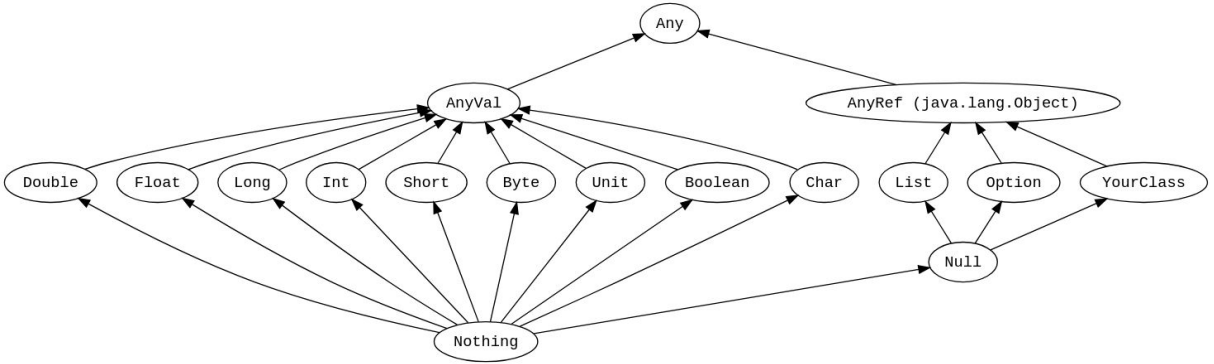
- Return type van functie met alleen side effects

- Return type van assignment expression

- Return type van ()

Waarden, objecten en functies

simacan




```
val result: String = if (condition) {
 "Success"
} else {
 throw new RuntimeException("Oops")
}
```

- *throw* retourneert nooit normaal, dus de *throw* expressie heeft het type *Nothing.*

- *Nothing* kan niet geïnstantieerd worden

- *Nothing* is een subtype van elk ander type

- Het gemeenschappelijke supertype van deze if/else is *String*.

Speciale types

simacan

```
val fun: () => String = () => "Hello"
```

Type van fun

Functie-definitie

- Een functie zonder params die string retourneert

```
println(fun()) // Hello

val double: Int => Int = (int: Int) => int * 2

println(double(3)) // 6
```

- Een functie van Int naar Int

Waarden, objecten en functies

simacan

- ‘Every value is an object’
  - Geen primitieven
- ‘Every expression returns a value’
- **‘Every function is a value’**

Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- ✓ Functies
- **Collecties framework**
- **Higher order functions**
- Monads
- Pattern matching
- For comprehension
- Implicits
- Het Scala landschap

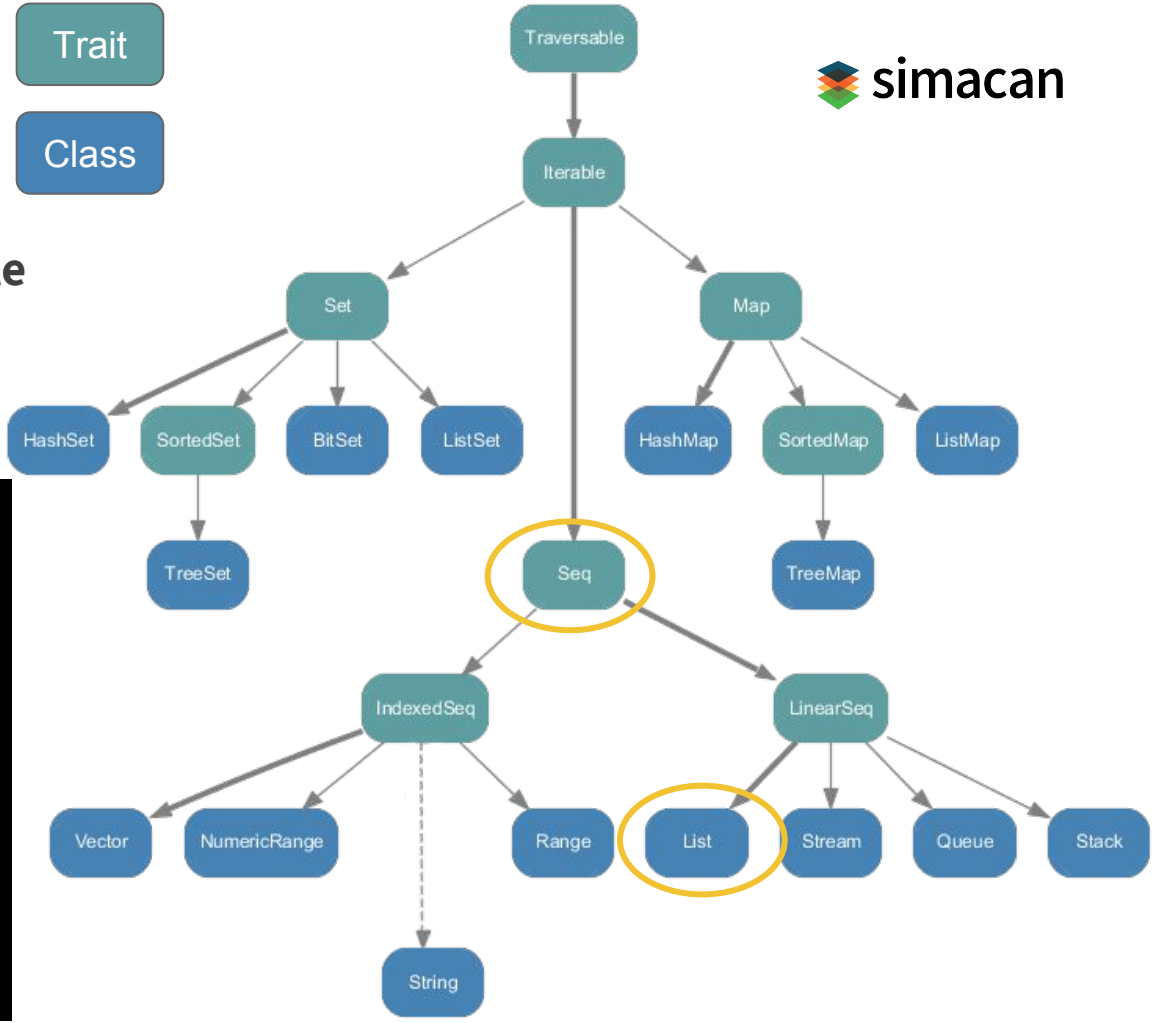
Scala collecties

- **scala.collection.immutable**
- scala.collection.mutable

```
val intSeq: Seq[Int] = Seq(1, 2, 3)

println(intSeq)

// Prints List(1, 2, 3)
```



simacan

Scala collections: map()

simacan

```scala
val intSeq: Seq[Int] = Seq(1, 2, 3)

val doubleSeq: Seq[Int] = intSeq
  .map(value => value * 2)


println(intSeq) // Prints List(1, 2, 3)
println(doubleSeq) // Prints List(2, 4, 6)

val stringSeq: Seq[String] = intSeq
  .map(int => int.toString)
```

map() is een higher order function: zijn parameter is een functie *van* het type in de list *naar* iets anders. Hier Int => Int

Maak een lijst van Strings.

In Java: een for loop waarbij je elke waarde in de list expliciet aanpast.

Scala collections: map()

simacan

```scala
val intSeq: Seq[Int] = Seq(1, 2, 3)

val doubleSeq: Seq[Int] = intSeq
  .map(value => value * 2)
```

map() is een higher order function: zijn parameter is een functie *van* het type in de list naar iets anders. Hier Int => Int

```scala
val doubleSeq2: Seq[Int] = intSeq
  .map(_ * 2)
```

Verkorte notatie anonymous function

```scala
private def double(int: Int) = int * 2

val doubleSeq3 = intSeq.map(double)
```

Een function is een value, dus een elders gedefinieerde functie kan ook worden meegegeven.

Scala collections: flatMap()

simacan

```scala
val intSeq: Seq[Int] = Seq(1, 2, 3)

val negAndPos: Seq[Seq[Int]] = intSeq
.map(value => Seq(-value, value))
// List(List(-1, 1), List(-2, 2), List(-3, 3))

val simpleNegAndPos: Seq[Int] = negAndPos.flatten
// List(-1, 1, -2, 2, -3, 3)

val negAndPos2: Seq[Int] = intSeq
.flatMap(value => Seq(-value, value))
// List(-1, 1, -2, 2, -3, 3)
```

flatten slaat een collectie van collecties plat.

map + flatten = flatMap()

Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- ✓ Functies
- ✓ Collecties framework
- ✓ Higher order functions
- **Monads**
- Pattern matching
- For comprehension
- Implicits
- Het Scala landschap

‘Monadic’ types

simacan

- Objecten in een ‘doos’
- Een operatie om een object in de Monad te stoppen.
- Een functie om het object in de doos te transformeren.

Option

simacan

- “Een lijst van 0 of 1 waarden”
- java.util.Optional

```
case class Cat(happiness: Int = 5) {
 def pet: Cat = Cat(happiness + 5)
}

val catInABox: Option[Cat] = Some(Cat())
val emptyBox: Option[Cat] = None

val happyCatInABox: Option[Cat] = catInABox
  .map(cat => cat.pet)

println(happyCatInABox) // Some(Cat(10))


val stillEmpty: Option[Cat] = emptyBox
  .map(cat => cat.pet)

println(stillEmpty) // None
```

Options en flatMap()

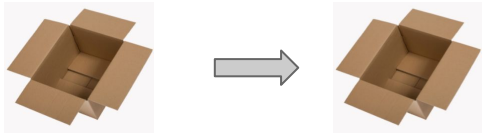
simacan

```
case class Cat(happiness: Int)
case class Toy(description: String)
case class CatWithToy(cat: Cat, toy: Toy)

def giveToyToCat(catOpt: Option[Cat],
 toyOpt: Option[Toy]): Option[CatWithToy] = {

 catOpt.map(cat =>
   toyOpt.map(toy => CatWithToy(cat, toy))
 )
}
```

Compiler error: deze dubbele map retourneert Option[Option[CatWithToy]].



Options en flatMap()

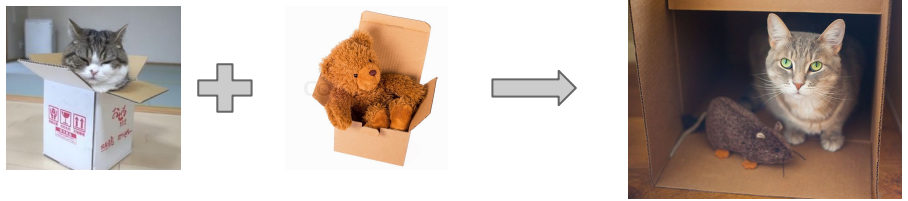
simacan

```
case class Cat(happiness: Int)
case class Toy(description: String)
case class CatWithToy(cat: Cat, toy: Toy)

def giveToyToCat2(catOpt: Option[Cat],
 toyOpt: Option[Toy]): Option[CatWithToy] = {

 catOpt.flatMap(cat =>
   toyOpt.map(toy => CatWithToy(cat, toy))
 )
}
```

Gewenste situatie: als catOpt en/of toyOpt *None* is wordt *None* geretourneerd, anders *Some(CatWithToy)*.



Options en flatMap()

simacan

- Voor een map() of flatMap() maakt het niet uit je Option *None* is of je Seq *empty*. Het zijn veilige operaties.

- Nooit meer *NullPointerExceptions*

Als je per ongeluk *map()* ipv *flatMap()* gebruikt:

‘Monadic’ types

simacan

- **Collections**: *Nil* of *Traversable[T]*
- **Option**: *None* of *Some[T]*
- **Try**: *Failure(Throwable)* or *Success[T]*
- **Either**: *Left[T]* or *Right[U]*

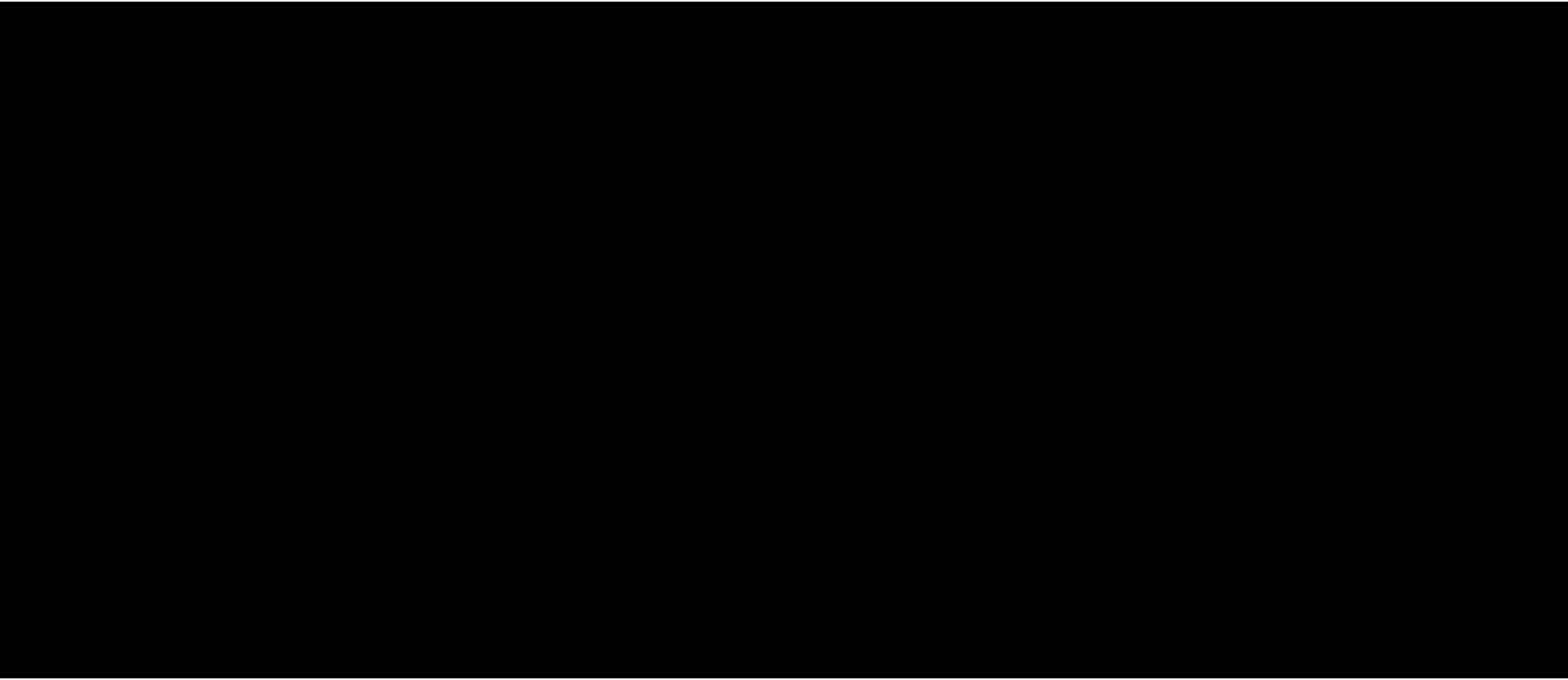
Futures

simacan

- **scala.concurrent.Future**
- Scala default voor async berekeningen/multithreading
- Monadic syntax:
    - Een *map()* op een *Future* is een volgende stap in de async berekening.
    - Een *flatMap()* op een *Future* wordt gebruikt om verschillende *Future*s als één geheel te laten afronden.
    - Bij een *exception* in een *Future* worden verdere *maps/flatMaps* niet uitgevoerd (vgl. *Try.Failure*).

Oefening

simacan



Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- ✓ Functies
- ✓ Collecties framework
- ✓ Higher order functions
- ✓ Monads
- **Pattern matching**
- For comprehension
- Implicits
- Het Scala landschap

Pattern matching

Op waarde

```
val x: Int = Random.nextInt(10)

val res: String = x match {
 case 0 => "nul"
 case 1 => "een"
 case _ => "veel"
}

println(res)
```

- Een match statement is een expressie met als type het common supertype van alle cases.
- De value van de eerste matchende case wordt geretourneerd (geen fallthrough).
- Als er geen match mogelijk is geeft de expressie een runtime exception.
- _ is een wildcard die ‘alle andere cases’ afvangt.

Pattern matching

Op type

simacan

```
abstract class Pet
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

val x: Pet = Dog("Pluto")

val res: String = x match {
 case _: Cat => "Ik ben een kat"

 case _: Dog => "Ik ben een hond"

 case _ => "Ik ben een ander huisdier"
}

val res2: String = x match {
  case _: Cat => "Ik ben een kat"

  case hond: Dog =>
    s"Ik ben een hond genaamd ${hond.name}"

   case _ => "Ik ben een ander huisdier"
}
```

Match alles (underscore) van het type Cat.

Match alles (underscore) van het type Dog.

Match alles van het type Dog en zet dat in een val genaamd ‘hond’.

Pattern matching

Op type

```
abstract class Pet
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

val x: Pet = Dog("Pluto")

val res: String = x match {
 case _: Cat => "Ik ben een kat"

 case _: Dog => "Ik ben een hond"

 case _ => "Ik ben een ander huisdier"
}
```

Wildcard case is nodig om runtime excepties te voorkomen...

Pattern matching

Op type

```
sealed abstract class Pet
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

val x: Pet = Dog("Pluto")

val res: String = x match {
 case _: Cat => "Ik ben een kat"

 case _: Dog => "Ik ben een hond"
}
```

Wildcard case is nodig om runtime excepties te voorkomen…

Behalve wanneer je de superclass/trait *sealed* maakt, want dan kunnen geen nieuwe subtypes buiten deze file gedefinieerd worden.

Pattern matching

Op type

```
sealed abstract class Pet
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet
case class Goldfish(name: String) extends Pet

val x: Pet = Dog("Pluto")

val res: String = x match {
 case _: Cat => "Ik ben een kat"

 case _: Dog => "Ik ben een hond"
}
```

Wildcard case is nodig om runtime excepties te voorkomen…

Behalve wanneer je de superclass/trait *sealed* maakt, want dan kunnen geen nieuwe subtypes buiten deze file gedefinieerd worden.

Compiler warning.

Pattern matching

simacan

Met case classes

```
case class Road(
                name: Option[String],
                roadNumber: Option[String],
                lengthMeters: Int,
                roadType: String)

val localRoad: Road = Road(
 Some("Hobbelweg"),
 None,
 1500,
 "eenbaans")


def singleCarriageWay(road: Road): Option[Road] = {
 road match {
   case Road( ,  , _, "eenbaans") => Some(road)
   case _ => None
 }
}
```

Extra functie van case classes: extract values voor pattern match.
_ is wildcard, ook in een case class match.

Pattern matching

simacan

Met case classes… realistischer voorbeeld

```
case class Road(
                name: Option[String],
                roadNumber: Option[String],
                lengthMeters: Int,
                roadType: String)

val localRoad: Road = Road(Some("Hobbelweg"), None,
  1500, "eenbaans")
val highWay: Road = Road(None, Some("A28"),
  50000, "snelweg")

val roads: Seq[Road] = Seq(localRoad, highWay)

def getRoadIdentifier(road: Road): Option[String] = road match {
 case Road(name: Some[String],  ,  ,  ) => name
 case Road( , roadNr: Some[String], _, _) => roadNr
 case _ => None
}

val roadIdentifiers: Seq[String] = roads.map(getRoadIdentifier) // List(Hobbelweg, A28)
```

Pattern matching

Met case classes… realistischer voorbeeld

simacan

```
def getRoadIdentifier(road: Road): Option[String] = road match {
 case Road(name: Some[String], _, _, _) => name
 case Road(_, roadNr: Some[String], _, _) => roadNr
 case _ => None
}

val roadIdentifiers: Seq[String] =roads.map(getRoadIdentifier) // List(Hobbelweg, A28)
```

De match expressie:

- geeft de name terug als die *Some* is
- geeft de roadNr terug als die *Some* is
- geeft None terug als ze beide None zijn

Pattern matching

Met case classes, met pattern guard

simacan

```
def getRoadLengthDescription(road: Road): String = road match {
 case Road(_, _, length, _) if length < 1000 => s"Dit is een erg korte weg"
 case Road(_, _, length, _) => s"Deze weg is best lang"
}
```

Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- ✓ Functies
- ✓ Collecties framework
- ✓ Higher order functions
- ✓ Monads
- ✓ Pattern matching
- **For comprehension**
- Implicits
- Het Scala landschap

For comprehensions

simacan

Het Scala *for* keyword is divers inzetbaar

1. for als een foreach()

```
val list: Seq[String] = Seq("We", "love", "Scala")
```

```
for (string <- list) {
 println(string)
}
```

for loopt over de list, en geeft *string* elke iteratie een volgende waarde uit de list.

```
list.foreach(println)
```

Dit doet exact hetzelfde.

Deze simpele for loop en list.foreach retourneren beide *Unit.*

For comprehensions

simacan

2. for als een map()

```
val list: Seq[String] = Seq("We", "love", "Scala")


val upperCaseList: Seq[String] =
  list.map(string => string.toUpperCase)

println(upperCaseList) // List(WE, LOVE, SCALA)


val upperCaseListWithFor: Seq[String] =
 for (string <- list)
   yield {
     string.toUpperCase
   }

println(upperCaseListWithFor) // List(WE, LOVE, SCALA)
```

toUpperCase in een normale list map.

Met *yield* krijgt een *for* een return value. Als de value in de *generator* een *Seq* is, wordt het resultaat ook weer in een *Seq* gezet.

For comprehensions

simacan

3. for met een filter

```
case class Cat(name: String, happiness: Int)

val cats: Seq[Cat] = Seq(
  Cat("Tom", 5),
  Cat("Mientje", 8),
  Cat("Oreo", 10))

val happyCats: Seq[Cat] =
  for (cat <- cats if (cat.happiness > 7))
    yield cat

println(happyCats)
// List(Cat(Mientje,8), Cat(Oreo,10))
```

De for loop itereert over alle katten, maar de if bepaalt welke kandidaten zijn voor verdere verwerking.

For comprehensions

simacan

4. for met meerdere *generators*

```
trait Toy
case object ToyMouse extends Toy
case object LaserPointer extends Toy

case class Cat(name: String, favouriteToys: Seq[Toy])

val cats: Seq[Cat] = Seq(
  Cat("Tom", Seq(ToyMouse)),
  Cat("Mientje", Seq(LaserPointer, ToyMouse)),
  Cat("Oreo", Seq(LaserPointer)))


val catsWithToys: Seq[String] =
 for (
   toy <- Seq(ToyMouse, LaserPointer);
   cat <- cats if cat.favouriteToys.contains(toy)
 ) yield s"${cat.name} likes to play with the $toy"

// List(Tom likes to play with the ToyMouse, Mientje likes to play with the ToyMouse,
Mientje likes to play with the LaserPointer, Oreo likes to play with the LaserPointer)
```

Vergelijkbaar met Java nested for loop.

For comprehensions

simacan

4. for met meerdere *generators*

```
val catsWithToys: Seq[String] =
 for (
   toy <- Seq(ToyMouse, LaserPointer);
   cat <- cats if cat.favouriteToys.contains(toy)
 ) yield s"${cat.name} likes to play with the $toy"
```

Is hetzelfde als:

```
val catsWithToys: Seq[String] =
 Seq(ToyMouse, LaserPointer)
   .flatMap(toy => cats
     .withFilter(cat => cat.favouriteToys.contains(toy))
     .map(cat => s"${cat.name} likes to play with the $toy"))
```

Met een *for comprehension* kun je diep geneste map()s op een leesbare wijze noteren.
Het is syntactic sugar.

For comprehensions werken op **alle types** die de map(), flatMap() en withFilter() functies implementeren.

Lists, Options, Futures, maar ook custom types zoals DB queries of HttpRequests uit Scala libraries.

For comprehensions

Praktijkvoorbeeld

simacan

```
case class Location(id: Option[Long])

val from: Location = Location(Some(12345))
val to: Location = Location(Some(67890))

val routeForLocationPair: Option[String] = for {
 fromId <- from.id
 toId <- to.id
 route <- tryCalculateRoute(fromId, toId)
} yield route

def tryCalculateRoute(fromId: Long, toId: Long): Try[String] = {
// In echte code: voer complex algorithme uit
 Try("Routebeschrijving")
}
```

Dit compileert niet! Alle generators moeten hetzelfde monadic type hebben (hier *Option*).

For comprehensions

Praktijkvoorbeeld

```
case class Location(id: Option[Long])

val from: Location = Location(Some(12345))
val to: Location = Location(Some(67890))

val routeForLocationPair: Option[String] = for {
 fromId <- from.id

 toId <- to.id

 route <- tryCalculateRoute(fromId, toId).toOption

} yield route
```

Nu werkt het wel.

In woorden:

1) Haal uit de *from* Location de id.

2) Als 1) bestaat, Haal uit de *to* Location de id.

3) Als 2) bestaat, bereken de route.

4) Als 3) gelukt is return het resultaat,anders None.

```
def tryCalculateRoute(fromId: Long, toId: Long): Try[String] = {
// In echte code: voer complex algorithme uit
 Try("Routebeschrijving")
}
```

For comprehensions

Praktijkvoorbeeld

```
case class Location(id: Option[Long])

val from: Location = Location(Some(12345))
val to: Location = Location(Some(67890))

val routeForLocationPair: Option[String] = for {
 fromId <- from.id

 toId <- to.id

 route <- tryCalculateRoute(fromId, toId).toOption

} yield route


def tryCalculateRoute(fromId: Long, toId: Long): Try[String] = {
// In echte code: voer complex algorithme uit
 Try("Routebeschrijving")
}
```

Voordelen:

- Alles kan in één *for comprehension.*
- Als er iets mis gaat (er mist een id of het algoritme faalt) wordt er een simpele None teruggegeven.

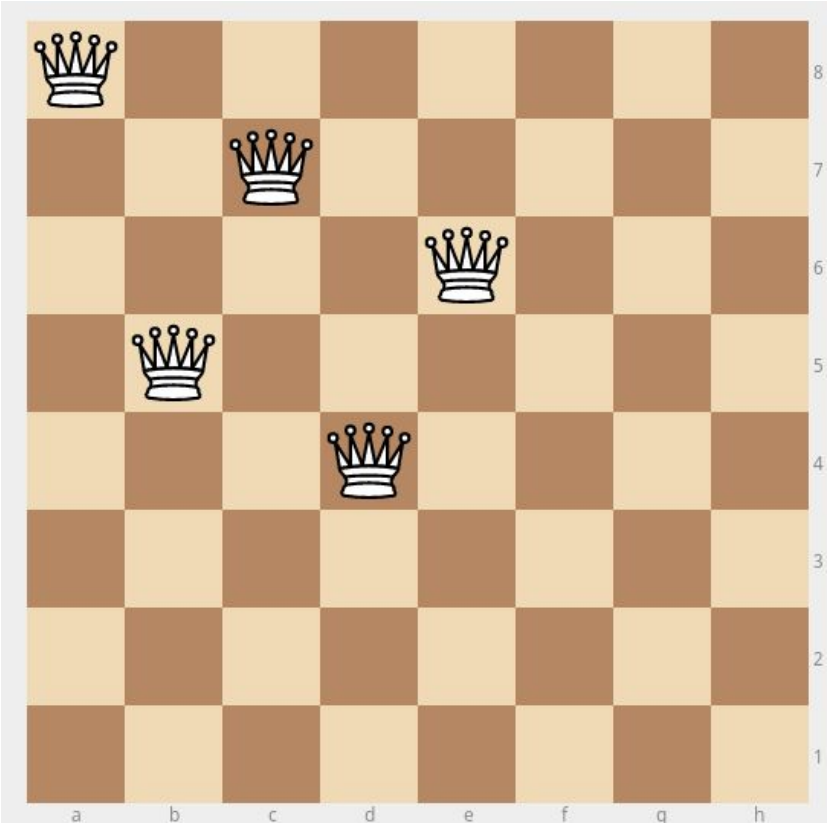
Nadeel:

- De *.toOption* transformeert een *Try.Failure* naar een *None*.

Demo: 8 queens

simacan

- Dijkstra’s Backtracking algorithm
- 92 oplossingen



Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- ✓ Functies
- ✓ Collecties framework
- ✓ Higher order functions
- ✓ Monads
- ✓ Pattern matching
- ✓ For comprehension
- **Implicits**
- Het Scala landschap

Implicit classes

simacan

```
val routeForLocationPair: Option[String] = for {
 fromId <- from.id
 toId <- to.id

 route <- tryCalculateRoute(fromId, toId).toOption

} yield route
```

Voorbeeld van eerder:

tryCalculateRoute retourneert Try maar we hadden Option nodig.

toOption gooit echter error messages in Try.Failure weg.

Implicit classes

simacan

```
val routeForLocationPair: Option[String] = for {
 fromId <- from.id
 toId <- to.id

 route <- tryCalculateRoute(fromId, toId).toOptionWithLogging

} yield route
```

Zou het niet mooi zijn om dit te kunnen doen?

Implicit classes

simacan

```
import TryUtil.TryOps

val routeForLocationPair: Option[String] = for {
 fromId <- from.id
 toId <- to.id

 route <- tryCalculateRoute(fromId, toId).toOptionWithLogging

} yield route
```

Nu werkt dit!

```
import TryUtil.TryOps

val routeForLocationPair: Option[String] = for {
 fromId <- from.id
 toId <- to.id

 route <- TryOps(tryCalculateRoute(fromId, toId)).toOptionWithLogging

} yield route
```

Onder de motorkap:

Implicit classes

simacan

```
object TryUtil {

 implicit class TryOps[T](in: Try[T]) { }
}
```

Een *implicit class* zorgt ervoor dat er een *implicit conversion* gedaan kan worden.

In dit voorbeeld: als deze class in scope is, wordt een Try automatisch omgezet naar een TryOps wanneer nodig.

Implicit classes

simacan

```
object TryUtil {

 implicit class TryOps[T](in: Try[T]) {

   val logger = LoggerFactory.getLogger("TryOps")

   def toOptionWithLogging(msg: String = "A Try failed."): Option[T] = {

     in.failed.foreach { throwable => logger.error(msg, throwable) }

     in.toOption
   }
 }

}
```

We schrijven een functie die een failed Try logt, en vervolgens de toOption aanroept.

Implicit classes

simacan

- Vnl. gebruikt om extra functies ‘toe te voegen’ aan library classes.
- Laat de compiler en IntelliJ Scala-plugin je helpen.
- Scala heeft ook andere soorten *implicits*

Programma

simacan

- ✓ Over ons
- ✓ Scala vs. Java
- ✓ Scala basic syntax
- ✓ Polymorphisme
- ✓ Case classes
- ✓ Functies
- ✓ Collecties framework
- ✓ Higher order functions
- ✓ Monads
- ✓ Pattern matching
- ✓ For comprehension
- ✓ Implicits
- **Het Scala landschap**

Hulpmiddelen

simacan

- Documentatie
  - Incl. ScalaDoc
- IntelliJ Scala-plugin
  - Toont inferred types
  - Geeft hints voor syntactic sugar
  - Toont het gebruik van implicits
  - etc.
- Scala REPL
  - Interactive shell
  - IntelliJ Scala worksheets
- Scala compiler
  - Compiler errors

Tooling - sbt

simacan

- *Scala Build Tool*
- Scala variant van Ant/Maven/Gradle
- Scala-based DSL
- Plugins geschreven in Scala
  - Bijv: *sbt-dependency-graph*
  - Simacan CI plugin

Tooling - ScalaTest

simacan

- Het meest populaire test framework voor Scala
  - Te gebruiken voor unit tests, automated integration tests, etc.
  - Compatible met sbt maar ook JUnit, ScalaMock, Mockito, etc.
  - Human-readable DSL.

```
def toUpperCaseList(list: Seq[String]): Seq[String] =
 list.map(string => string.toUpperCase)


"toUpperCaseList" should "convert each string in the list to an uppercase string" in {
 val testSeq = Seq("hello", "world")

 val result = toUpperCaseList(testSeq)

 result should have size 2
 result(0) should be "HELLO"
 result(1) should be "WORLD"
}
```

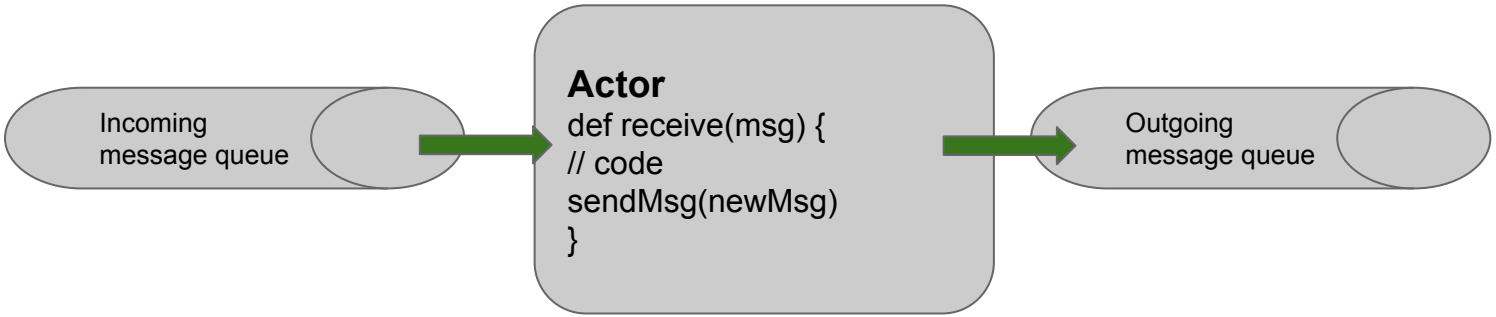
Tooling - Akka

simacan

- Een verzameling gerelateerde libraries
- Akka Actors
  - Erlang-achtig ‘actor’ concurrency systeem

Tooling - Akka Actors

simacan



akka

- Model voor concurrent computation.
- Een Actor heeft state en code.
- Een Actor kan alleen met de buitenwereld en met andere Actors communiceren via messages.
- Hierdoor heb je geen locks meer nodig.

Tooling - Akka

simacan

- Een verzameling gerelateerde libraries
- Akka Actors
- Akka Streams
  - Asynchrone non-blocking stream processing
- Akka Http
  - Asynchrone streaming-based HTTP server en client
  - REST services
- Akka Cluster
- Akka Persistence

Bronnen

simacan

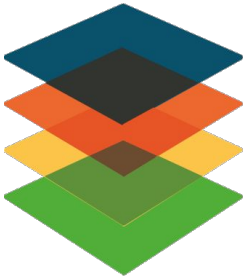
- Om te beginnen:
  - https://docs.scala-lang.org/
  - https://www.tutorialspoint.com/scala/
  - https://www.coursera.org/learn/progfun1
  - https://www.scala-exercises.org
- Scala programmeren in je browser
  - https://scalafiddle.io/
- Implicits:
  - http://www.lihaoyi.com/post/ImplicitDesignPatternsinScala.html
- SBT:
  - https://www.scala-sbt.org/1.x/docs/sbt-by-example.html
- Akka:
  - https://akka.io/docs/

Slides en code examples

simacan

https://github.com/simacan/NLJUG-sessies

Bedankt voor je aandacht!



simacan