Scala for all

property-based testing met ScalaCheck ⚡

Philippus Baalman

Wehkamp - 10 maart 2017

scala>

An order with order lines


```
scala>
```

```scala
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine]) {
  def addOrderLine(orderLine: OrderLine): Order = {
    this.copy(orderLines = orderLine :: orderLines)
  }
  def total: Int = orderLines.map(x => x.total).sum
}

sealed abstract class CustomerType(override val toString: String)
case object NewCustomer extends CustomerType("new")
case object ExistingCustomer extends CustomerType("existing")

case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int) {
  def total: Int = quantity * netValue
}
```


scala>

```
val testOrderLine = OrderLine(Option("a"), "b", "000", 1, 2)

val testOrder = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List(testOrderLine))

"Order" should {
  "be able to add an order line to an order" in {
    testOrder.addOrderLine(testOrderLine).orderLines.length
    shouldEqual testOrder.orderLines.length + 1
  }
}
```


scala>

```
"Order" should {
  "calculate total correctly" in {
    testOrder.total shouldEqual testOrderLine.total
  }
}
```


scala>

```
val testOrderWithoutOrderLines = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List())
```


scala>

```
"Order" should {
  "be able to add an order line to an order" in {
    testOrder.addOrderLine(testOrderLine).orderLines.length
    shouldEqual testOrder.orderLines.length + 1
  }

  "be able to add an order line to an order without order lines" in {
    testOrderWithoutOrderLines.addOrderLine(testOrderLine).orderLines.length
    shouldEqual 1
  }

  "be able to add an order line to an order" in {
    testOrder.addOrderLine(testOrderLine).orderLines.length
    shouldEqual testOrder.orderLines.length + 1
  }

  "calculate total correctly for orders without order lines" in {
    testOrderWithoutOrderLines.total shouldEqual 0
  }
}
```



```
scala>
```

```
val testOrderLine = OrderLine(Option("a"), "b", "000", 1, 2)

val jsonOrderLine =
  """
    {"id":"a","productId":"b","productCategory":"000","quantity":1,"netValue":2}
  """.parseJson

"OrderLineMarshalling" should {
  "correctly serialize an order line" in {
    jsonOrderLine shouldEqual testOrderLine.toJson
  }
  "correctly deserialize an order line" in {
    jsonOrderLine.convertTo[OrderLine] shouldEqual testOrderLine
  }
}
```


scala>

```
val testOrder = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List(testOrderLine))

val jsonOrder =
  """
    {"orderLines":[{"id":"a","productId":"b","productCategory":"000","quantity":1,
    "netValue":2}],"id":"a","orderTime":"20161209T10:50:24.091Z",
    "orderState":"ordered","customerType":"new"}
  """.parseJson

"OrderMarshalling" should {
  "correctly serialize an order" in {
    jsonOrder shouldEqual testOrder.toJson
  }
  "correctly deserialize an order" in {
    jsonOrder.convertTo[Order] shouldEqual testOrder
  }
}
```



```
scala>
```

```
val testOrder = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List(testOrderLine))

val jsonOrder =
  """
    {"orderLines":[{"id":"a","productId":"b","productCategory":"000","quantity":1,
    "netValue":2}],"id":"a","orderTime":"20161209T10:50:24.091Z",
    "orderState":"ordered","customerType":"new"}
  """.parseJson

val testOrderWithoutOrderLines = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List())

val jsonOrderWithoutOrderLines =
  """
    {"orderLines":[],"id":"a","orderTime":"20161209T10:50:24.091Z",
    "orderState":"ordered","customerType":"new"}
  """.parseJson
```



```
scala>
```

```
"OrderMarshalling" should {
  "correctly serialize an order" in {
    jsonOrder shouldEqual testOrder.toJson
    }
  "correctly deserialize an order" in {
    jsonOrder.convertTo[Order] shouldEqual testOrder
  }
  "correctly serialize an order without order lines" in {
    jsonOrderWithOrderLines shouldEqual testOrderWithoutOrderLines.toJson
  }
  "correctly deserialize an order without order lines" in {
    jsonOrderWithOrderLines.convertTo[Order] shouldEqual testOrderWithoutOrderLines
  }
}
```


scala>

Bij property-based testing definieer je eigenschappen die gelden voor de output van je code, gebaseerd op de input.

Properties

scala>

```scala
import org.scalacheck.Prop._

val propInt = forAll {
  i: Int =>
    i + 0 == i
}
```


scala>

```
propInt.check
```



```
scala>
```

```
forAll {
  (a: String, b: String) =>
    (a + b).endsWith(b)
}.check
```


scala>

```
forAll {
  l: List[Int] =>
    l.reverse == l
}.check
```



```
scala>
```

```
forAll {
  l: List[String] =>
    l.reverse.reverse == l
}.check
```


scala>

```
forAll {
  (i: Int, j: Int) =>
    i + j == j + i
}.check
```



```
scala>
```

```
forAll {
  s: String =>
    s.toUpperCase.toLowerCase == s.toLowerCase
}.check
```


scala>

```
forAll {
  i: Int =>
    Math.abs(i) >= 0
}.check
```


scala>

Bij property-based testing definieer je eigenschappen die gelden voor de output van je code, gebaseerd op de input, deze worden geverifieerd voor heel veel verschillende gegeneerde input.

Generators

scala>

```
forAll {
  i: Int =>
    collect(i) {
      Math.abs(i) >= 0
    }
}.check
```


scala>

```
import org.scalacheck.Arbitrary.arbitrary
import org.scalacheck.Gen

arbitrary[String].sample
```



```
scala>
```

```
arbitrary[Int].sample
```



```
scala>
```

```
arbitrary[(Int, String)].sample
```



```
scala>
```

```
Gen.listOfN(3, arbitrary[Int]).sample
```



```
scala>
```

```
Gen.posNum[Int].sample
```



```
scala>
```

```
Gen.alphaStr.sample
```



```
scala>
```

```
forAll(Gen.alphaStr) {
  s: String =>
    s.toUpperCase.toLowerCase == s.toLowerCase
}.check
```



```
scala>
```

Bij property-based testing definieer je eigenschappen die gelden voor de output van je code, gebaseerd op de input, deze worden geverifieerd voor heel veel verschillende gegeneerde input, als er een fout gevonden wordt dan wordt er geprobeerd een minimaal tegenvoorbeeld te vinden.

Shrinking


```
scala>
```

```
forAll {
  l: List[Int] =>
    l.reverse == l
}.check
```


scala>

```
forAll {
  s: String =>
    println(s)
    s.toUpperCase.toLowerCase == s.toLowerCase
}.check
```



```
scala>
```

An order revisited


```
scala>
```

```scala
case class Order(
    id: String,
    orderTime: org.joda.time.DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine]) {
  def addOrderLine(orderLine: OrderLine): Order = {
    this.copy(orderLines = orderLine :: orderLines)
  }
  def total: Int = orderLines.map(x => x.total).sum
}

sealed abstract class CustomerType(override val toString: String)
case object NewCustomer extends CustomerType("new")
case object ExistingCustomer extends CustomerType("existing")

case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int) {
  def total: Int = quantity * netValue
}
```


scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

import org.scalacheck.Gen._

def genOrderLineId: Gen[String] = for {
  left ← Gen.listOfN(1, Gen.numChar).map(_.mkString)
  middle ← Gen.const("x")
  right ← Gen.listOfN(5, Gen.numChar).map(_.mkString)
} yield List(left, middle, right).mkString
```


scala>

genOrderLineId.sample

scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

```
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

genOrderLine.sample

scala>

```scala
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

import org.joda.time.{DateTime, DateTimeZone}

def genOrderTime: Gen[DateTime] = {
  val start = new DateTime(2017, 1, 1, 0, 0, 0, 0, DateTimeZone.UTC).getMillis
  val end = new DateTime(2018, 1, 1, 0, 0, 0, 0, DateTimeZone.UTC).getMillis
  Gen.choose(start, end).map(new DateTime(_))
}
```


scala>

```
genOrderTime.sample
```



```
scala>
```

```
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

def genOrderState = oneOf("ordered", "packaged", "shipped", "delivered")
```


scala>

genOrderState.sample


```
scala>
```

```scala
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

sealed abstract class CustomerType(override val toString: String)
case object NewCustomer extends CustomerType("new")
case object ExistingCustomer extends CustomerType("existing")

def genCustomerType: Gen[CustomerType] = oneOf(NewCustomer, ExistingCustomer)
```


scala>

genCustomerType.sample


```
scala>
```

```scala
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

def genOrder: Gen[Order] = for {
  id ← Gen.listOfN(8, Gen.numChar).map(_.mkString)
  orderTime ← genOrderTime
  orderState ← genOrderState
  customerType ← Gen.option(genCustomerType)
  orderLines ← Gen.resize(5, listOf(genOrderLine))
} yield Order(id, orderTime, orderState, customerType, orderLines)
```


scala>

genOrder.sample


```
scala>
```

```
forAll(genOrder, genOrderLine) {
  (o: Order, ol: OrderLine) =>
    o.addOrderLine(ol).orderLines.length == o.orderLines.length + 1
}.check
```


scala>

```
forAll(genOrder, genOrderLine) {
  (o: Order, ol: OrderLine) =>
    o.total + ol.total == o.addOrderLine(ol).total
}.check
```



```
scala>
```

```
val propSerializationRoundtrip = forAll {
  order: Order =>
    order == order.toJson.convertTo[Order]
}
```


scala>

"Program testing can be used to show the presence of bugs,
but never to show their absence!" -- Edsger W. Dijkstra

scala>