Talen

Domeinspecifieke talen zijn in, maar niet nieuw. Talen als SQL en CSS bestaan al enige tijd, en zijn geschreven voor één specifiek doel. Wel nieuw is de grote belangstelling voor het zelf schrijven van domeinspecifieke talen. Dat is met de komst van nieuwe dynamische talen op de JVM veel eenvoudiger geworden. We laten zien hoe je snel zelf een domeinspecifieke taal kunt schrijven. We schrijven onze taal bovenop Groovy, zodat we een groot aantal standaard constructies niet meer zelf hoeven te schrijven. En we laten zien hoe je je eigen taal kunt integreren in een Java-project.

Snel en eenvoudig resultaat met Groovy

Schrijf een domeinspecifieke taal met Groovy

Michel Vollebregt
is software-ontwikkelaar bij VX Company. Daarnaast is hij actief lid van de Groovy en Grails User Group (NLGUG).

Domeinspecifieke talen zijn heel geschikt om specifieke problemen op te lossen. Met EasyB schrijven we leesbare tests. Met Gradle schrijven we elegante build scripts. Omdat deze talen op één doel zijn toegespitst, kunnen we problemen er korter en leesbaarder in opschrijven. EasyB en Gradle zijn beide voorbeelden van embedded talen en zijn gebouwd bovenop een bestaande taal. Embedded talen nemen automatisch alle constructies van de moedertaal over. Dit maakt dergelijke talen heel krachtig. Het alternatief, het van de grond opbouwen van een taal met een eigen parser, biedt meer controle, maar kost ook aanzienlijk meer tijd.

De mogelijkheden om in Java embedded talen te schrijven zijn beperkt. De syntax van Java is weliswaar eenvoudig, maar ook erg strikt. Andere talen op de JVM zijn expressiever en bieden meer mogelijkheden voor het schrijven van domeinspecifieke constructies. Een voorbeeld van zo'n taal is Groovy. Groovy biedt veel syntactic sugar, waardoor domeinspecifieke code erg aantrekkelijk oogt. Groovy is vanwege de syntax bovendien erg laagdrempelig voor Java-ontwikkelaars. Daarom is Groovy uitstekend geschikt om snel tot een resultaat te komen.

Groovy syntax

We gaan uit van een denkbeeldige Java-library. De library heeft een Java-API zoals we die gewend zijn. Zo'n API wordt ook wel command query API genoemd. We schrijven een dun laagje Groovy-code om de command query API heen. Dit dunne laagje implementeert de domeinspecifieke taal. In het algemeen is het aan te bevelen om de domeinspecifieke taal op deze manier los te koppelen van de onderliggende logica.
In de onderstaande code is een klassedefinitie uit de library gegeven. Onder de definitie staat een stuk voorbeeldcode.

```
class Money {
  Money(double amount, String currency);
  double getAmount();
  String getCurrency();
  Money plus(Money other);
  Money convertTo(String currency);
}

class Example {
  public static void main(String[] args) {
    Money am1 = new Money(2, "euro");
    Money am2 = new Money(3, "dollar");
    Money cv1 = am1.convertTo("dollar");
    System.out.println(cv1.plus(am2)) ;
  }
}
```

Bovenstaande code is zowel geldig in Java als in Groovy. Groovy bestaat uit de syntax van Java plus een aantal extra's. In Groovy mogen we de code uit de main-method direct in een script plaatsen, zonder de klassedefinitie daaromheen. Typedefinities en puntkomma's zijn optioneel. En ook System.out.println mag korter. De klasse met voorbeeldcode kunnen we meteen al reduceren tot een script van vier regels:

```
am1 = new Money(2, “euro”)
am2 = new Money(3, “dollar”)
cv1 = am1.convertTo(“dollar”)
println cv1 + am2
```

In de laatste regel zien we hoe Groovy operator overloading ondersteunt. a.plus(b) en a + b betekenen exact hetzelfde. We kunnen iedere klasse dus een +-operator geven, simpelweg door plus() te implementeren. Voor andere operatoren gelden soortgelijke methodenamen.

Klassedefinities in Java zijn statisch. Eenmaal vastgelegd, kunnen ze niet meer worden veranderd. In Groovy kunnen we bestaande klassen uitbreiden met nieuwe methoden. Dat kan met onze eigen klassen, maar ook met standaard klassen als String, Integer en Number.
We schrijven een methode getEuro() om eurobedragen te creëren. We voegen deze toe aan de Number-klasse. In plaats van:

```
am1 = new Money(2, “euro”)
```

schrijven we dan:

```
am1 = 2.getEuro()
```

Met Groovy’s speciale notatie voor getters en setters, verkorten we dat nog verder tot:

```
am1 = 2.euro
```

2.getEuro() en 2.euro zijn in Groovy volstrekt identiek.

We kunnen methoden toevoegen aan een klasse met behulp van een categoryklasse. Een category is een speciaal geprepareerde klasse die extra functionaliteit vastlegt op bestaande klassen. De categoryklasse die wij nodig hebben ziet er als volgt uit:

```
class MoneyArithmetic {
  static getEuro(Number amount) {
    return new Money(amount, “euro”)
  }
}
```

Onze categoryklasse definieert één methode getEuro(). Deze methode wordt niet aangeroepen op instanties van de categoryklasse zelf, maar op instanties van een andere klasse. Daarom is een categorymethode altijd static. De feitelijke klasse waaraan de methode moet worden gekoppeld, wordt gegeven door het eerste argument van de methode. Eventuele overige argumenten zijn de normale methodeargumenten.

De MoneyArithmetic-klasse is het eerste stukje van onze eigen DSL-implementatie. Deze kunnen we steeds opnieuw gebruiken. Om de DSL in het script te gebruiken, voegen we een use-clausule toe:

```
use (MoneyArithmetic) {
    cv1 = 2.euro.convertTo(“dollar”)
    am2 = new Money(3, “dollar”)
    println cv1 + am2
}
```

Het is ook mogelijk om getEuro() beschikbaar te maken zonder de use-clause. Maar hoewel dat niet veel moeilijker is, valt het buiten de scope van dit artikel.

Dynamische properties

Groovy kent meerdere manieren om aan de waarde van een property te komen. Wanneer getEuro() expliciet is gedefinieerd, gebruikt Groovy die uiteraard. Maar wanneer dat niet het geval is, probeert Groovy het op andere manieren. Dynamische properties zijn zo’n manier. Deze worden geïmplementeerd door een get()-methode. Wanneer getEuro() niet gedefinieerd is, probeert Groovy get(“euro”) aan te roepen om alsnog aan een waarde te komen.
We gebruiken dit principe om op een uniforme manier verschillende muntsoorten te creëren. We hoeven onze MoneyArithmetic maar een klein beetje aan te passen om hem generieker te maken:

```
class MoneyArithmetic {
    static get(
        Number amount, String property) {
      return new Money(amount, property)
    }
}
```

Binnen de use-clausule roept Groovy nu MoneyArithmetic.get(2, “euro”) aan.

Method missing

We kunnen nu met minimale syntax bedragen creëren. In de volgende stap gaan we ook het converteren van valuta mooier schrijven. Dat zal leiden tot de volgende syntax:

```
use (MoneyArithmetic) {
    println 2.euro.toDollar() + 3.dollar
}
```

We kunnen onze Money-klasse uitbreiden met een toDollar()-methode. We willen echter niet alleen naar dollars converteren, maar ook naar andere valuta. En we willen niet voor iedere valuta een aparte methode schrijven. Gelukkig kent Groovy niet alleen dynamische properties, maar ook dynamische methoden. Wanneer we een methode aanroepen die Groovy niet kan vinden, zoekt Groovy naar een speciale methodMissing()-methode. Binnen deze methode kunnen we zelf bepalen wat er gebeurt.
We breiden onze Money-klasse uit met een methodMissing(). Wanneer de naam van de ontbrekende methode begint met to, converteren we de muntsoort. Wanneer dat niet het geval is, gooien we een MissingMethodException.

```
class GroovyMoney extends Money {
```

Groovy kent meerdere manieren om aan de waarde van een property te komen.

Groovy ondersteunt closures, uitvoerbare code om als object aan een variabele toe te wijzen.

```
  GroovyMoney(
      double amount, String currency) {
    super(amount, currency);
  }

  def methodMissing(
      String methodName, args) {
    if (methodName.startsWith("to")) {
      return convertTo(
          methodName.substring(2));
    } else {
        throw new MissingMethodException(
              methodName, this.class, args)
    }
  }
}
```

Money is een Java-klasse. GroovyMoney is daarvan een subklasse in Groovy. Dat is geen enkel probleem. De Groovy-compiler is een joint compiler. Hij combineert moeiteloos Groovy met Java.

Een eigen scripttaal

We hebben Groovy uitgebreid met syntax voor het opschrijven van bedragen. In de volgende stap rekenen we scenario's door met deze bedragen. De scenario's schrijven we op in een eigen scripttaal. Die scripttaal zullen we in de rest van het artikel implementeren. We gaan daarbij uit van het volgende voorbeeldscript:

```
MoneyDsl.calculate {

  jan = 20.euro
  slijter = 100.euro

  betaal (10.euro, btw: 19, accijns: 20) {
    van jan
    aan slijter
  }
}
```

Het script lijkt nauwelijks op programmacode. Het is echter wel degelijk uitvoerbare Groovy-code. Laten we één voor één naar de constructies kijken, die de bovenstaande code mogelijk maken.

Knutselen met closures

Groovy ondersteunt closures. Een closure is een stuk uitvoerbare code dat we als object aan een variabele kunnen toewijzen. Ook kunnen we het als parameter aan een functie meegeven. Closures zijn vergelijkbaar met functiepointers in C of delegates in C#.

We creëren een closure door een stuk code tussen accolades te plaatsen. Bijvoorbeeld:

```
def myClosure = { println "my closure" }
myClosure()
```

De eerste regel definieert een closure en kent die toe aan een variable. Pas in de tweede regel wordt de code daadwerkelijk aangeroepen en wordt de waarde 'my closure' afgedrukt.

In Groovy zijn haakjes niet verplicht. De eerste regel van het voorbeeldscript roept dus een statische methode MoneyDsl.calculate() aan. De rest van het script definieert een closure. De closure is de parameter voor MoneyDsl.calculate(). De volgende code is een eerste aanzet voor onze scripttaal. Calculate() voert simpelweg de closure uit:

```
class MoneyDsl {
  static calculate(closure) {
    use (MoneyArithmetic) {
      closure()
    }
  }
}
```

Doorverwijzen met delegates

We zijn er nog niet. Aanroepen van het script geeft een melding dat de methode script.betaal() niet bestaat. Wat is er aan de hand? Regel 6 van het script roept een betaal()-methode aan, met drie 'normale' parameters en een closure. De betaal()-methode is echter niet gedefinieerd. Daarom geeft Groovy een foutmelding.

We kunnen de foutmelding verhelpen door een betaal()-methode aan het script toe te voegen. De betaal()-methode is echter een onderdeel van de scripttaal, en niet van het script zelf. We moeten Groovy dus vertellen waar hij de betaal()-methode kan vinden.

Standaard voert Groovy closures uit in de context waarin ze zijn gedefinieerd. Dat betekent dat Groovy de betaal()-methode zoekt in het voorbeeldscript. We kunnen functieaanroepen op closures eenvoudig omleiden door een delegate te zetten:

```
  closure.delegate = new MoneyDsl()
  closure()
```

Methodeaanroepen binnen de closure worden nu uitgevoerd op de instantie van het MoneyDsl-object. We kunnen de betaal()-methode nu dus implementeren binnen MoneyDsl. En dat is precies waar hij thuis hoort.

Named parameters

Betaal() heeft twee parameters die er wellicht raar uit zien. BTW en accijns zijn zogenaamde named parameters. De naam van zulke parameters staat voor de dubbele punt. De waarde staat erachter. We kunnen willekeurige aantallen named parameters aan een methode meegeven.

De named parameters worden door Groovy verzameld in een HashMap. Deze HashMap is altijd de eerste methodeparameter. De overige parameters volgen daarna. We kunnen onze betaal()-methode op de MoneyDsl dus als volgt vastleggen:

```
def betaal(percentages, bedrag, closure) {
  def tebetalen =
    bedrag * ((100.0 + percentages.btw +
            percentages.accijns) / 100.0)
  closure()
  vanpersoon.subtract tebetalen
  aanpersoon.add tebetalen
}
```

percentages.btw is een korte notatie voor percentages.get("btw").

Binnen de betaal()-methode roepen we de meegegeven closure aan. In ons voorbeeldscript leidt dat tot het aanroepen van methoden van(jan) en aan(slijter). Deze methoden leggen we vast binnen MoneyDsl. De implementatie is eenvoudig:

```
class MoneyDsl {

    private vanpersoon;
    private aanpersoon;

    ...

    def van(persoon) {
        this.vanpersoon = persoon
    }

    def aan(persoon) {
        this.aanpersoon = persoon
    }

}
```

Closure bindings

De calculate()-methode voert nu de gewenste berekening uit. Na het uitvoeren van het script bevatten jan en slijter de uitkomsten van de berekening. We willen deze uitkomsten graag teruggeven als returnwaarde. De vraag is, hoe we deze waarden tevoorschijn kunnen halen uit de closure.
Gelukkig is dit eenvoudig. Iedere closure slaat zijn variabelen op in een Binding-object. Binding heeft een methode getVariables(). Deze geeft een HashMap terug van alle variabelenamen en waarden. De API-documentatie voor Binding is te vinden onder: http://groovy.codehaus.org/api.

Onze code wordt nu:

```
closure.delegate = new MoneyDsl()
closure()
return closure.binding.variables
```

Integratie met Java

Kunnen we de DSL gebruiken binnen een Java-project? Jazeker. We gebruiken de DSL dan specifiek voor het vastleggen van ingewikkelde geldberekeningen. De geldberekeningen staan opgeschreven in een taal die daar speciaal voor bedoeld is. Dat maakt de berekeningen veel leesbaarder. We kunnen de rekenscripts schrijven samen met een domeindeskundige. Die domeindeskundige kan onze scripttaal immers prima begrijpen. Maar ook onze eigen build-scripts en test-scripts worden in een DSL veel leesbaarder.

We blijven in alle gevallen zelf verantwoordelijk voor de inhoud van de scripts. Iedere Groovy-expressie is immers ook geldig in de scripttaal. En we willen niet dat verkeerd geschreven scriptcode de werking van onze software negatief beïnvloedt.

Om het script vanuit Java uit te voeren, hebben we slechts een enkele regel code nodig:

```
HashMap values = (HashMap) ((Script)
  new GroovyClassLoader().parseClass( new
  File("script.dsl")).newInstance()).run();
```

De code voert het Groovy-script uit en stopt de berekende waarden in een lokale Java-variabele. Deze waarden kunnen we zonder meer in het Java-project gebruiken. We hoeven alleen maar groovy-all.jar aan ons classpath toe te voegen. Deze jar wordt standaard meegeleverd met de Groovy-distributie.

Conclusie

Met Groovy schrijven we bijzonder snel een domeinspecifieke taal. De uitdrukkingsmogelijkheden in Groovy zijn erg groot. Daardoor kunnen we domeinspecifieke scripts zowel voor onszelf als voor een domeindeskundige zeer leesbaar opschrijven. Integratie in een Java-project is bovendien eenvoudig. DSL's in Groovy kunnen in de praktijk hun nut dus goed bewijzen. «

Integratie in een Java-project is bovendien eenvoudig met Groovy.