Bij de start van een nieuw project kiezen ontwikkelaars soms, om redenen van snelheid en flexibiliteit, liever voor een scripting-taal. Tegenwoordig kan men kiezen uit een hele reeks scripting-talen die gebruik maken van de Java Virtual Machine en de Java-API. In dit artikel bespreekt Willem Koppenol de Java scripting-standaarden die in ontwikkeling zijn rond `javax.scripting` en de scripting-talen BeanShell en Groovy en geeft en passant een nadere beschouwing van het fenomeen scripting.

Java scripting talen

Standaards volop in ontwikkeling

Bij de start van een nieuw project zien ontwikkelaars zich vaak gesteld voor de vraag welke taal zich het best leent voor de gestelde taak. De keuze voor een robuuste, gecompileerde, type-safe taal als Java, ligt niet voor alle projecten voor de hand. Soms kiezen ontwikkelaars om redenen van snelheid en flexibiliteit liever voor een scripting-taal. In vroeger tijden was de keuze bij een scripting-taal beperkt tot standalone scripting-talen, zoals Ruby, Perl, Python, JavaScript en Tcl. Het voornaamste verschil tussen een scripting-taal en een gecompileerde taal is gelegen in het type systeem: de manier waarop je data-elementen definieert en ermee werkt. Mogelijk springt het verschil tussen het interpreteren of compileren van code eerder in het oog. Dit verschil is echter minder fundamenteel van aard. Interpretatie of compilatie heeft vooral invloed op de snelheid van code en flexibiliteit van codering en niet zozeer op de manier van coderen.

Sterke typering van variabelen is belangrijk bij het ontwikkelen van grote robuuste applicaties. Het gebruik van een sterk getypeerde taal maakt het immers gemakkelijker een correct werkende applicatie te krijgen. Door het type systeem van een taal kan een compiler de structuur van de applicatie checken op correctheid.

Het fenomeen scripting Maar het werken met types is ook een belasting voor de ontwikkelaar. Het steeds vastleggen van de types van variabelen is bij bepaalde vormen van ontwikkeling een belemmering. Soms is niet programmastructuur, maar zijn flexibiliteit, eenvoud en gebruiksgemak de belangrijkste criteria. Dit is het geval bij projecten die bijvoorbeeld te maken hebben met prototyping, dynamische configuratie of het programmeren van unit-tests. De inzet van een scripting-taal is dan een goede oplossing. Scripting-talen kunnen gebruikt worden voor stand-alone applicaties. Het is echter ook mogelijk een scripting-taal in een applicatie te embedden. De gebruikers van de applicatie krijgen dan de mogelijkheid de applicatie uit te breiden door er scripts voor te schrijven. Een script-interpreter moet aan de applicatie worden gekoppeld voor het uitvoeren van de scripts.

Scripting in Java Een scripting-taal die de Java Virtual Machine gebruikt heeft het voordeel van portabiliteit. De scripting-taal heeft toegang tot alle platformen waarvoor er een virtuele machine is. Dergelijke

Scripting taal	Beschrijving
Jacl	Java-implementatie van de Tcl interpreter. Bestaat al geruime tijd. Wordt nog actief aan ontwikkeld.
Jython	Java-implementatie van de Python interpreter. Een nieuwe release laat al geruime tijd op zich wachten.
Rhino	Java-implementatie van de JavaScript interpreter. Ondersteunt XML en compilatie tot class files
JRuby	Java-implementatie van de Ruby interpreter. Wordt actief aan ontwikkeld.
BeanShell	Java-broncode interpreter. Version 2.0 ondersteunt gewone Java volledig.
Groovy	Voegt kenmerken van Python en Ruby aan een Java-achtige syntax. Groovy plugins zijn beschikbaar voor vele IDE's.
JudoScript	JavaScript-achtige syntax. Doel is scripting te leveren op applicatie, OS en object niveau.
Pnuts	Java-achtige syntax. Je kunt de script direct tot class files compileren. Wordt actief aan gewerkt.

Tabel 1. Een overzicht van scripting-talen voor Java.

scripting-talen zijn over het algemeen interoperabel met Java en kunnen gebruik maken van de rijk geschakeerde Java-API. Er bestaan een hele reeks scripting-talen voor Java (zie tabel). De talen worden over het algemeen ontwikkeld volgens een open source-model. De talen BeanShell en Groovy die verheven zijn tot JSR zullen we nader bekijken.

Momenteel zijn er een drietal Java Specification Requests (JSR's) op het gebied van Java Scripting onderweg. 'Scripting for the Java Platform' (JSR223), 'The Groovy Programming Language' (JSR241) en 'Beanshell' (JSR274). De achtergronden van deze JSR's worden in het vervolg nader besproken.

SCRIPTING FOR THE JAVA PLATFORM 'Scripting for the Java Platform' definieert de nieuwe `javax.script`-API om vanuit Java met een reeks scripttalen te werken. Deze API is met name bedoeld om scripts te kunnen embedden in Java-applicaties. De API is ontstaan uit het IBM/Apache Bean Scripting Framework en zal onderdeel gaan uitmaken van Mustang (Java 6.0). De default script-taal van de referentie-implementatie is PHP.

Een krachtig aspect eraan is dat je heel gemakkelijk een script-engine kunt inpluggen. Als je bijvoorbeeld een Ant-taak met een nieuwe scripting-taal wilt scripten, hoef je alleen het JAR-file voor die scripting-taal op de juiste plaats neer te zetten. Het maakt voor de Ant-taak niet uit welke taal je voor het script gebruikt, mits de Ant-taak de `javax.script`-API begrijpt.

De `javax.script`-API schermt de ontwikkelaar of van mechanische details als het opzoeken van een scripting-engine, het binden van waarden van variabelen aan een script en het uitvoeren van een script. De `javax.script`-API gaat tamelijk ver in de ondersteuning van scripting. Variabelen uit de applicatie kun je bijvoorbeeld via een simpel `Map`-interface bekend maken aan scripts. De scripts zullen deze variabelen zien alsof ze in het script zelf zijn gedeclareerd. In onderstaande code fragment wordt eerst een `php` scripting-engine geladen en vervolgens een `php`-script uitgevoerd:

```
import javax.script.*;
import java.io.*;

public class HelloWorld {
   public static void main(String[] args)
throws Exception {
   //Initiate ScriptEngineManager
   ScriptEngineManager manager = new
ScriptEngineManager();

   //Return Javascript engine by name
   ScriptEngine jsengine = manager.
getEngineByName("php");

   //Execute the HelloWorld script
   jsengine.eval(new FileReader("./
HelloWorld.php"));
   }
}
```

Het aangeroepen `php`-script:

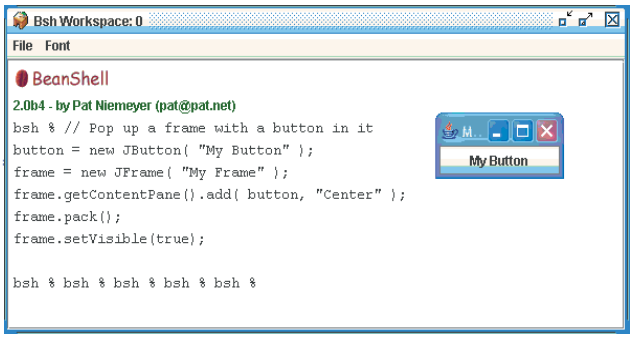
```
<? print("\nHello World!\n"); ?>
```

BEANSHELL Eén van de bekendere Java scripting-talen is de BeanShell. Deze taal heeft een hele schare volgelingen gekregen sinds Patrick Niemeyer begon te experimenteren met Java-scripting. Vandaag de dag is de BeanShell een gedreven open-source project en is Beanshell-scripting opgenomen in een reeks commerciele en open source-projecten, zoals BEA's WebLogic, de NetBeans IDE, Sun's Java Studio en vele anderen. De BeanShell is onlangs verheven tot Java Specification Request (JSR 274) en is daarmee op weg onderdeel te worden van de Java-standaard.

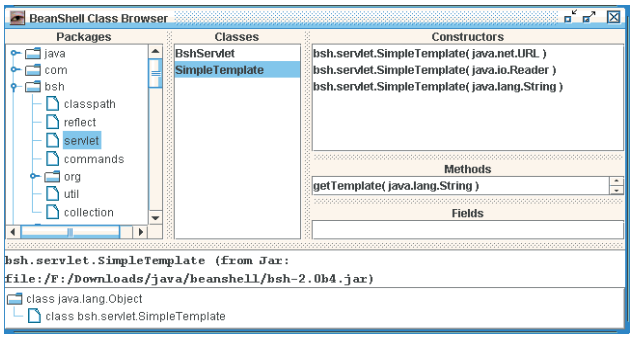
DYNAMIC SCRIPTING Op het basale niveau is de BeanShell een Java source-interpreter die de aangeboden Java-code dynamisch, tijdens runtime, interpreteert. De Beanshell-syntax is gebaseerd op de Java-syntax en is daar een uitbreiding op. De Java-syntax is echter versoepeld zodat Java in Beanshell op scriptachtige wijze gebruikt kan worden. Een voorbeeld hiervan is dat ongetypeerde, niet-gedeclareerde variabelen zijn toegestaan. Ook kun je simpele ongestructureerde scripts schrijven die bestaan uit willekeurige statements en expressies. Je kunt gescripte of gecompileerde 'commands' aanroepen, die zich gedragen als Java-methoden, maar die pas bij aanroep geladen worden. Verder kun je het classpath dynamisch wijzigen en classes *on-the-fly* aanmaken.

BEANSHELL MODES De BeanShell-distributie komt zowel met een grafische als een command line-console.

FIGUUR 1. Beany, de mascotte van de BeanShell.



FIGUUR 2. Grafische BeanShell console met script en het resultaat van het script.



FIGUUR 3. BeanShell's class browser.

Scripts kunnen vanuit beide consoles worden ingegeven en uitgevoerd. Ook kun je een bestand met scripting-commando's aan BeanShell meegeven en laten uitvoeren. Tenslotte kan BeanShell ook vanuit een Java-applicatie, een servlet of zelfs applet worden aangesproken, mits de BeanShell-interpreter wordt meegeleverd. Een voorbeeld van een script dat een `JFrame` met `JButton` aanmaakt, is te zien in de afbeelding van de grafische BeanShell-console. Het `JFrame` wordt *on-the-fly* aangemaakt en is meteen zichtbaar.

BEANSHELL BASIS-SYNTAX BeanShell kent de standaardzaken die in Java-methoden worden gebruikt, zoals variabele declaraties, assignments, aanroepen van methodes, iteraties en condities. Je kunt deze zaken in BeanShell net zo gebruiken als in standaard-Java. Een functie die twee, sterk getypeerde, integers optelt is:

```
int addTwoNumbers( int a, int b ) {
    return a + b;
}
sum = addTwoNumbers( 9, 6 );  // result : 15
```

In BeanShell bestaat echter ook de optie om te werken met 'loosely typed'-variabelen. Je kunt simpelweg de types weglaten van de variabelen die je gebruikt. En dit geldt zowel voor primitieve types als voor objecten. BeanShell zal een fout melden als het actuele type van een variabele wordt misbruikt. Een 'loosely typed'-versie van de functie kan zowel gebruikt worden voor het optellen van integers als voor het concateneren van strings:

```
add( a, b ) {
    return a + b;
}

foo = add(1, 2);                        //
result : 3
foo = add("Hello", " Scripting");   // result
: "Hello Scripting"
```

Het `switch`-statement werkt in BeanShell niet alleen met integers, zoals in Java, maar ook met objecten. Zo kun je bijvoorbeeld switchen op `Date` en `String`-objecten die vergeleken worden met de `equals()`-methode. BeanShell ondersteunt automatische boxing en unboxing. Deze termen duiden aan dat primitieve typen waar nodig worden omhuld met een wrapper-object of een primitief type uit een wrapper-object wordt gehaald. De gebruikelijke Java-core en extension packages worden in Beanshell automatisch geïmporteerd. Hieronder vallen de `swing, awt, net, util` en `io` packages. Verder komt BeanShell met een class-browser waarmee je packages, classes, methoden en velden kunt inspecteren.

BEANSHELL'S DYNAMISCHE KENMERKEN BeanShell kent naast standaard en iets afwijkende Java syntax ook een aantal syntax elementen die nogal afwijken van wat in Java gebruikelijk is. Een tweetal daarvan, maar zijn er meer, worden in het navolgende besproken. Zo kent BeanShell een 'invoke' meta-methode:

```
invoke(name, args) {....}
// Of, equivalent met alle type information
void invoke( String name, Object[] arg)
{....}
```

Als `invoke()` in een script wordt opgenomen, handelt hij alle aanroepen naar niet bestaande methoden af. Dit is handig als je bijvoorbeeld te maken hebt met een interface met veel methoden en je bent maar geïnteresseerd in één daarvan. Met een `invoke()`-implementatie in je script, hoef je niet alle methoden van het interface op te nemen. Een voorbeeld is een `MouseListener` interface (vijf methoden) implementatie die alleen geïnteresseerd is in `mousePressed`:

```
<< begin kader met computercode >>
mouseHandler = new MouseListener() {
    mousePressed( event ) {
        print("mouse button pressed");
    }
```

```
    invoke( method, args ) {
        print("Undefined MouseListener
method:"
             + name +", with args: "+args
        );
    }
};
```

Een ander nuttig onderdeel van BeanShell is de `this.caller` referentie, waarmee je in een methode variabelen kunt benaderen die zich in de scope van de aanroeper van de methode bevinden. In onderstaand codefragment heeft de aanroep van de `varSetter()` methode het effect dat de variabele `var` in de scope van de aanroeper wordt gezet:

```
varSetter() {
    this.caller.var=11;
}
varSetter();
print( var );        // result : 11
```

Dit is met name van belang bij het schrijven van BeanShell commands, waarvan je vaak wilt dat ze expliciete gevolgen hebben in de scope van de aanroeper. Door deze mogelijkheid lijken nieuwe commands dan onderdeel te zijn van de taal. Een goed voorbeeld is het `eval()` BeanShell command, dat een string evalueert door deze naar een instantie van de BeanShell interpreter te sturen. Met de `this.caller` referentie geeft het `eval()` command aan dat de string in de scope van de aanroeper moet worden geëvalueerd:

```
eval("a=5");
print( a );        // result : 5
```

De implementatie van het `eval()` command ziet er als volgt uit:

```
eval( String text ) {
    this.interpreter.eval(text, this.caller.
namespace);
}
```

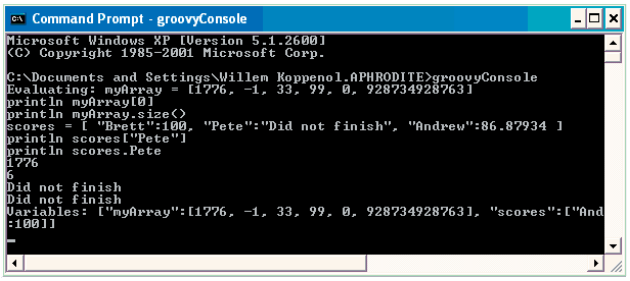
Groovy Groovy is een open-source scripting taal die geïmplementeerd is in Java en er nauw mee is geïntegreerd. Groovy voegt een aantal kenmerken van de Ruby en Python scripting talen aan Java toe. De Groovy syntax lijkt sterk op de Java syntax, maar is hier en daar wat compacter. De reeds besproken Beanshell blijft dichter bij de Java syntax. Kenmerken van Groovy zijn dynamic typing, closures, eenvoudige object navigatie en een meer compacte syntax bij het werken met Lists en Maps.

Figuur 4. Het Groovy-logo.

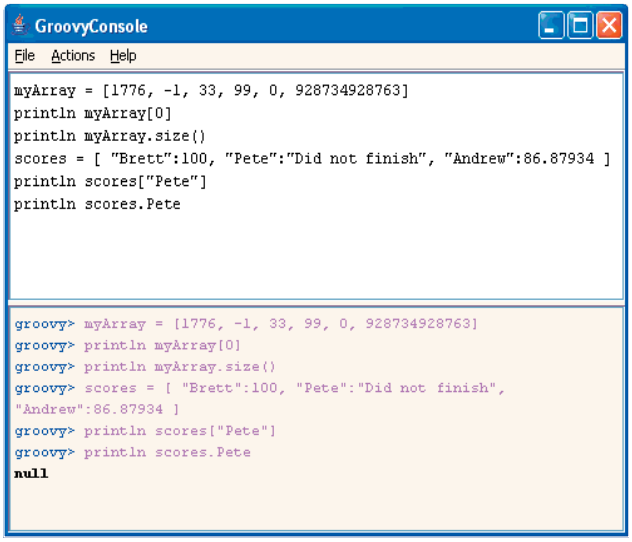
Groovy-scripts kunnen alle Java-classes uit de Java-API gebruiken. Ze kunnen worden gecompileerd tot Java-bytecode (.class files) en kunnen worden aangeroepen uit normale Java classes. De Groovy compiler, groovyc, vertaalt zowel Groovy-scripts als Java source files, hoewel niet de gehele Java-syntax wordt ondersteund. Groovy onderscheidt zich van andere scripting-talen zoals Ruby, Python en Beanshell, omdat hele applicaties in Groovy kunnen worden geschreven. Groovy is dus een nieuwe taal voor het Java-platform. De performance-karakteristiek van Groovy-applicaties is te vergelijken met die van Java applicaties. De reden dat Groovy vooralsnog wat langzamer is dan Java zelf is gelegen in het feit dat de gegenereerde bytecode reflectie gebruikt om constructors van private en protected methoden aan te roepen. Dit zal in volgende releases worden opgelost.

Groovy Consoles De Groovy-distributie komt, net als BeanShell, met zowel een grafische als een command-line console. In beiden kunnen Groovy-commands kunnen worden ingegeven en uitgevoerd. Ook is het mogelijk Groovy-scripts die in aparte bestanden zijn gedefinieerd aan te roepen en uit te voeren. De bijgevoegde schermafbeelding toont de grafische Groovy-console waarin een Groovy array en map worden gedeclareerd en worden benaderd. Let op de aparte, van Java afwijkende syntax, voor het declareren en vullen van een map. De output verschijnt in de karakter-console van waaruit de grafische console werd opgestart.

Scriptcompilatie Een Groovy script wordt opgeslagen met de extensie `.groovy` en kan vertaald wor-



Figuur 5. Grafische Groovy-console.



FIGUUR 6. Groovy-output in de karaktergeoriënteerde Groovy-console.

den tot een Java `.class` file met het commando:

```
groovyc script-name.groovy
```

Als het script bestaat uit losse statements, zal dit `.class` file een `main` methode bevatten, zodat het kan worden uitgevoerd als Java applicatie met het commando `java script-name`. Het classpath moet hiertoe de `groovy*.jar` en de `asm*.jar` files bevatten.

GROOVY BASIS-SYNTAX De Groovy-syntax komt in belangrijke mate overeen met de Java-syntax. Het doel is om in de toekomst alle legale Java-syntax te ondersteunen, maar zover is het nog niet. Opvallende verschillen in de basissyntax zijn, dat in Groovy puntkomma's aan het einde van statements en haakjes om parameters voor methoden optioneel zijn. Bij constructor-aanroepen zijn haakjes echter weer verplicht en return-waarden zijn soms optioneel. Verder zijn property's en methoden in Groovy public en niet protected zoals in Java.

Groovy ondersteunt dynamische typering in de zin dat types voor variabelen, property's en parameters en return-types van methoden optioneel zijn. Zij krijgen het type dat het meest recent is aan hen toegekend. Later kan weer een ander type worden toegekend. Type conversies tussen `Strings`, primitieve types (zoals `int`) en wrapper classes (zoals `Integer`) gebeuren automatisch. Dit betekent onder meer dat primitieve types en collecties kunnen worden toegevoegd.

Groovy ondersteunt in tegenstelling tot Java operator-overloading voor een vaststaande set van operatoren. Iedere operator correspondeert met een bepaalde methode-naam. Door deze methoden in classes te implementeren kun je de corresponderende operator gebruiken met objecten van deze classes. Een aantal voorbeelden zijn:

`a == b` correspondeert met `a.equals(b)`
`a < b` correspondeert met `a.compareTo(b) < 0`
`a * b` correspondeert met `a.multiply(b)`

GROOVY-CLOSURES Een Groovy-closure is een stukje code dat optioneel parameters accepteert. Iedere closure wordt gecompileerd tot een nieuwe class die is afgeleid van `groovy.lang.Closure`. De closure wordt aangeroepen met de `call`-methode van deze class. Er kunnen dan parameters aan meegegeven worden. Closures kunnen de variabelen benaderen die in scope zijn als de closure wordt gecreëerd. Anderzijds zijn variabelen die in een closure worden gecreëerd zichtbaar in de scope van waaruit de closure wordt aangeroepen. Een closure kan zelf in een variabele worden gestopt en als parameter worden doorgegeven. De syntax voor het definiëren van een closure is:

```
{ parameter lijst gescheiden door comma's |
statements }
```

Een code fragment waarin een closure wordt aangeroepen is:

```
closure = { bill, tipPercentage | bill * tip-
Percentage / 100 }
tip = closure.call(25.19, 15)
tip = closure(25.19, 15)       // equivalent
aan vorige regel
```

GROOVY BEANS Een Groovy Bean is een Groovy-class met een aantal property's. Groovy-classes en de property's en methoden van Groovy-classes zijn default `public`. `Public` en `protected` property's resulteren in Groovy in `private`-velden waarvoor automatisch `public` of `protected` getter- en setter-methoden worden gegenereerd. Deze methoden kunnen desgewenst worden hergedefinieerd met een eigen implementatie. Een eenvoudige Groovy-bean is:

```
class Car {
    String make
    String model
}
```

Deze Groovy-code komt overeen met de volgende Java-code:

```
public class Car {
    private String make;
    private String model;
```

```
    public String getMake() {
        return make;
    }
    public String getModel() {
        return model;
    }
    public void setMake(String make) {
        this.make = make;
    }
    public void setModel(String model) {
         this.model = model;
    }
}
```

De classes die door Groovy Beans worden gegenereerd zijn afgeleid van `java.lang.Object` en implementeren `groovy.lang.GroovyObject`. Aan Groovy Beans kunnen tijdens run-time methoden worden toegevoegd. Groovy Beans kunnen ook met named parameters worden gecreëerd. Het volgende code fragment roept de `Car` constructor zonder parameters aan en vervolgens een set methode voor iedere gespecificeerde property:

```
myCar = new Car(make:'Toyota', model:'Camry')
```

SLOTWOORD Java scripting-talen hebben zonder meer bestaanrecht naast Java. Enerzijds voor embedded toepassingen in applicaties, anderzijds voor standalone applicaties. Java is immers voor bepaalde projecten een overkill. Java scripting-talen zijn met 3 JSR's, `javax.scripting`, BeanShell en Groovy helemaal in ontwikkeling. BeanShell is vooral gericht op het scripten van applicaties. Groovy kan ook gebruikt worden voor op zichzelf staande applicaties. In feite is Groovy daarmee een alternatieve taal voor het Java-platform. De opkomst van de scripting-talen is onderdeel van een trend gericht op het vereenvoudigen van applicaties. De groeiende interesse in frameworks als Spring en Hibernate kan daar ook toe gerekend worden. Vooralsnog ziet het er echter niet naar uit dat scripting-talen de toonaangevende positie van Java bedreigen. Scripting talen zullen zich daartoe eerst in de praktijk bij grootschalige projecten moeten bewijzen.

drs. Willem Koppenol is Senior Trainer en Product Specialist Software Development bij Twice IT Training (e-mail: wkoppenol@twice.nl).

De opkomst van de scripting-talen is onderdeel van een trend gericht op het vereenvoudigen van applicaties.