whitehorses

member of the human network

WHITEBOOK

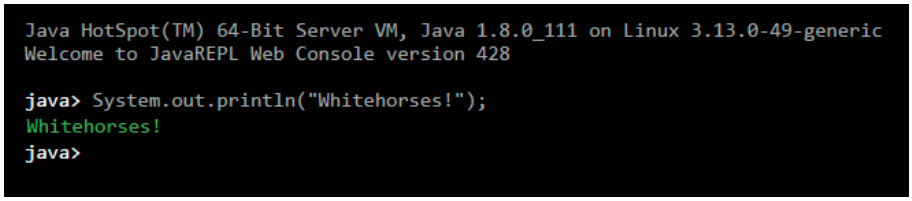
JShell in Java 9 - De eerste officiële Java REPL

Augustus 2017

Auteur:
Martijn van de Goor
INTEGRATIESPECIALIST

Inleiding

Het is bijna zover! Java SE 9 wordt op 21 september uitgebracht. Java 9 heeft, naast diverse kleinere aanpassingen, een belangrijk nieuw onderdeel: JShell (in het verleden ook wel Project Kulla genoemd).
JShell is de eerste officiële Java REPL (Read-Eval-Print-Loop) command-line tool die Java statements kan draaien zonder ze te moeten bundelen in klassen of methoden. De gebruiker voert één of meer expressies in, de REPL evalueert dit, en toont de uitkomst.
Veel programmeertalen, en dan vooral scripting-talen hebben al een REPL-tool. Voor Java was de programmeur tot nu toe aangewezen op niet-officiële tools wanneer je REPL wilde gebruiken. Een goed voorbeeld hiervan is Java REPL (www.javarepl.com) waarvan hieronder een schermafdruk is afgebeeld.



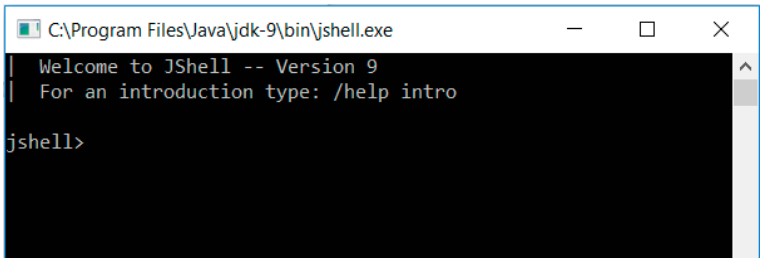
In dit Whiteboek worden de mogelijkheden van *JShell* uiteengezet aan de hand van diverse voorbeelden. Daarna worden een aantal *usecases* getoond waarin je *JShell* zou kunnen inzetten. Denk hierbij aan de educatieve mogelijkheden die het heeft of de mogelijkheid om snel met API's te kunnen experimenteren.

Aan de slag met JShell

Om aan de slag te gaan met *JShell* moet er een *Early Access Build* van JDK 9 geïnstalleerd worden. Deze is te vinden op http://jdk.java.net/9/. *jshell.exe* is eenvoudig te vinden in de *bin* folder van JDK 9. Om met *JShell* te experimenteren is het aan te raden om het in *verbose* mode te draaien zodat er extra informatie verschijnt over wat er onder water in *JShell* gebeurt. Type het volgende in op de command-line om *JShell* in *verbose* te starten (Alle invoer in dit Whitebook wordt dikgedrukt weergegeven):

```
> jshell -v
```

Na het starten van jshell.exe verschijnt een eenvoudige command-line tool:



Je kunt verbose weer uitzetten met het volgende commando:

```
> /set feedback normal
```

Het is nu mogelijk om statements in te voeren alsof je midden in een Java-klasse aan het programmeren bent. Het is dus niet nodig om een project of een *main*-methode aan te maken om snel een stuk code uit te proberen. Wanneer je *JShell* start krijg je standaard een aantal veelvoorkomende *imports*:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

JShell accepteert statements, variabelen, methodes en klassen, *imports* en expressies.

Zo kun je een statement invoeren en *JShell* geeft je vervolgens informatie over wat je hebt ingetoetst. Wanneer *verbose* aanstaat krijg je een uitgebreidere omschrijving die voorafgegaan wordt door een verticale streep:

```
jshell> int x = 56
x ==> 56
|  created variable x : int
```

In *JShell* is het niet nodig een puntkomma te gebruiken om een statement af te sluiten. Op zich is dit logisch aangezien de REPL het toestaat dat statements regel voor regel worden ingevoerd. Na deze korte introductie wordt nu verder gekeken naar de uitgebreide mogelijkheden die *JShell* biedt. Het is verstandig om de voorbeelden zelf te proberen in *JShell*, dan wordt het nog duidelijker. Aan de slag!

Expressies met tijdelijke variabele

JShell kan zelf expressies evalueren. Dit betekent dat *JShell* alles kan doen wat je normaal gesproken in *System.out.println()* zou zetten, zoals aanroepen naar methodes of berekeningen:

```
jshell> 5+8
$2 ==> 13
|  created scratch variable $2 : int
```

Het is dus niet nodig een variabele aan te maken. *JShell* maakt zelf een tijdelijke *scratch* variabele aan waardoor je later naar deze variabele kunt verwijzen.

Opslaan, laden en wijzigen van code

Binnen *JShell* is het mogelijk om de code die je hebt ingevoerd op te slaan en later weer in te laden. Hierdoor lijkt Java wat meer op een scripttaal. Opslaan van eerder uitgevoerde code gaat eenvoudig met het volgende commando:

```
jshell> /save D://test-JShell//whitehorses.txt
```

De ingevoerde statements zijn nu als script opgeslagen. Ze zijn vervolgens weer in te laden met het volgende commando:

```
jshell> /open D://test-JShell//whitehorses.txt
```

Met het commando /list wordt het script getoond:

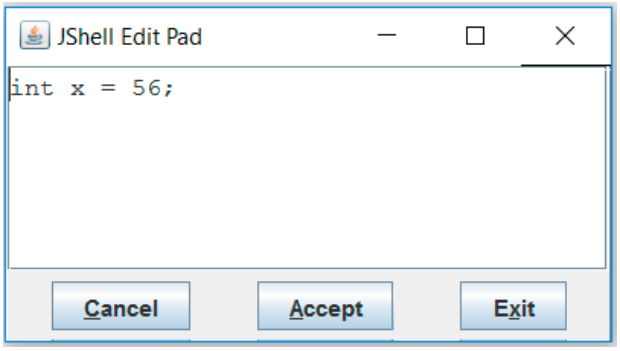
```
jshell> /list

  1 : int x = 56;
  2 : 5+8
```

Het is ook mogelijk om delen van een script, zoals bijvoorbeeld de variabele int x, te wijzigen:

```
jshell> /edit x
```

Een meegeleverde editor wordt dan geopend en men kan dan aanpassingen doen:



Nadat de code wordt gewijzigd in bijvoorbeeld int x = 68; verschijnt de volgende output:

```
x ==> 68
|  modified variable x : int
|    update overwrote variable x : int
```

Checked exceptions

Een leuke eigenschap van *JShell* is dat het niet verplicht is om *checked exceptions* af te vangen. Zoals bekend worden *checked exceptions* gecontroleerd tijdens het compileren. Dit betekent dat wanneer een methode een *checked exception* gooit, er wordt nagegaan of die uitzondering wordt afgevangen met een *try-catch block* of dat de methode wordt gedeclareerd met het *throws* keyword. In onderstaande voorbeeld kan *FileInputStream* een *FileNotFoundException* gooien. De compiler controleert of dit in de code is afgevangen:

```
import java.io.*;

public class test {
 public static void main(String[] args){
FileInputStream fis = null;
fis = new FileInputStream("D:/test-JShell/whitehorses.txt");
 }
}
```

Na compilatie krijgen we dan ook de volgende output:

```
D:\test-JShell>javac test.java
test.java:6: error: unreported exception FileNotFoundException; must be
caught or declared to be thrown
       FileInputStream fis = new FileInputStream("D:/test-JShell/
whitehorses.txt");
                             ^
1 error
```

We kunnen hetzelfde proberen in *JShell*:

```
jshell> import java.io.*;

jshell> FileInputStream fis = null;
fis ==> null
|  created variable fis : FileInputStream

jshell> fis = new FileInputStream("D:/test-JShell/whitehorses.txt");
fis ==> java.io.FileInputStream@3e9b1010
|  assigned to fis : FileInputStream
```

JShell klaagt niet dat *checked exception* niet is afgevangen. Dit wordt door *JShell* zelf in de achtergrond gedaan.

Er zijn uitzonderingen waarbij een *checked exception* wel moet worden afgevangen. In het onderstaande voorbeeld wordt een klasse met een methode gedeclareerd waarin *Thread.sleep()* wordt aangeroepen:

```
jshell> class MyThread extends Thread { public void run() { Thread.
sleep(2000);}}
|  Error:
|  unreported exception java.lang.InterruptedException; must be caught or
declared to be thrown
|  class MyThread extends Thread { public void run() { Thread.
sleep(2000);}}
|                                                      ^----------------^
```

Aangezien hier een gehele methode is gedeclareerd en niet een los *statement* moet de methode een valide Java *statement* zijn.

Verwijzing naar nog niet gedefinieerde code

Het is mogelijk in *JShell* om een functie te declareren met een verwijzing naar code die pas later wordt gedefinieerd. Dit heet *forward reference*. Stel voor dat je een methode wil declareren die de oppervlakte van een cirkel berekent. De formule komt er dan zo uit te zien:

```
jshell> double oppervlakte(double radius) { return PI * kwadraat(radius); }
|  created method oppervlakte(double), however, it cannot be invoked until
variable PI, and method kwadraat(double) are declared
```

JShell staat de definitie toe maar waarschuwt wel dat een variabele en een methode nog gedefinieerd dienen te worden. Wanneer *PI* gedefinieerd wordt en de methode *oppervlakte()* wordt aangeroepen, faalt de aanroep:

```
jshell> double PI = 3.14159265
PI ==> 3.14159265
|  created variable PI : double

jshell> oppervlakte(3)
|  attempted to call method oppervlakte(double) which cannot be invoked
until method kwadraat(double) is declared
```

Wanneer methode *kwadraat()* wordt gedefinieerd zal de aanroep van *oppervlakte()* wel slagen:

```
jshell> double kwadraat(double y) { return y * y; }
|  created method kwadraat(double)
|    update modified method oppervlakte(double)

jshell> oppervlakte(3)
$10 ==> 28.27433385
|  created scratch variable $10 : double
```

Aanvullen met de tab-toets

Veel *IDE*'s hebben een handige functie waarbij een woord dat je invoert wordt aangevuld wanneer je op de tab-toets drukt. *JShell* heeft ook zo'n functie. We hebben al eerder de methode oppervlakte gedeclareerd. *JShell* kan dit nu aanvullen:

```
jshell> opp<tab>
oppervlakte(

jshell> oppervlakte(
```

Het is ook mogelijk dat er meer dan één mogelijke aanvulling is, dan zal *JShell* ze allemaal tonen. Vervolgens kan de gebruiker dit verder aanvullen:

```
jshell> System.g<tab>
gc()                    getLogger(              getProperties()
getProperty(            getSecurityManager()
getenv(

jshell> System.g
```

Wanneer je na het haakje openen van de methode de tab-toets indrukt toont *JShell* informatie over de mogelijke type parameters. Wanneer je daarna weer tab indrukt, wordt de documentatie van de eerste methode getoond. Druk je vervolgens weer op tab dan wordt de documentatie van de volgende methode getoond:

```
System.getProperty(<tab>
Signatures:
String System.getProperty(String key)
String System.getProperty(String key, String def)

<press tab again to see documentation>

jshell> System.getProperty(<tab>
String System.getProperty(String key)
Gets the system property indicated by the specified key.
First, if there is a security manager, its checkPropertyAccess method is
called with the key as
its argument. This may result in a SecurityException.
If there is no current set of system properties, a set of system properties
is first created
and initialized in the same manner as for the getProperties method.

Parameters:
key - the name of the system property.

Returns:
the string value of the system property, or null if there is no property
with that key.

<press tab to see next documentation>

jshell> System.getProperty(
```

Importeren en transformeren

Met de tab-toets kun je nog een tweetal handige dingen doen. Stel dat je een *JButton* wil declareren, dan zal *JButton* geïmporteerd moeten worden. Dit kan worden gedaan met *<shift-tab>i*. In het begin is het even lastig om dit voor elkaar te krijgen. Houd de shift-toets ingedrukt en druk vervolgens de tab-toets in. Laat ze vervolgens los en druk de i-toets in. Er verschijnt een menu waarin je in dit geval de keuze krijgt om de *JButton* te importeren.

```
jshell> new JButton<shift-tab>i
0: Do nothing
1: import: javax.swing.JButton
Choice: 1
Imported: javax.swing.JButton
```

Na het aanvullen met een expressie is het mogelijk om de expressie te transformeren naar een variabele met *<shift-tab>v*. De cursor verschijnt op de plaats waar de naam van de variabele moet komen. In dit geval hebben we de variabele de naam *button* gegeven:

```
jshell> new JButton("Whitehorses")<shift-tab>v
jshell> JButton button = new JButton("Whitehorses")
button ==> javax.swing.JButton[,0,0,0x0,invalid,alignmentX=0 ...
orses,defaultCapable=true]
|  created variable button : JButton
```

Soms komt het voor dat het resultaat van de transformatie naar een variabele nog niet geïmporteerd is. Dan krijg je de mogelijkheid om de variabele zowel te importeren als te creëren. Dit lijkt ingewikkeld, maar met onderstaande voorbeeld wordt het duidelijk:

```
jshell> button.getGraphics()<shift-tab>v
0: Do nothing
1: Create variable
2: import: java.awt.Graphics. Create variable
Choice: 2
Imported: java.awt.Graphics

jshell> Graphics graphics = button.getGraphics()
graphics ==> null
|  created variable graphics : Graphics
```

Waarom JShell?

We hebben nu een aantal mogelijkheden van *JShell* gezien. De vraag is nu wat het nut van *JShell* is, daarom worden hier een aantal *usecases* besproken.

Verlagen van de leercurve van Java

Ik kan me herinneren dat toen ik begon te programmeren in Java ik werd geconfronteerd met een stukje code zoals dit:

```
public class TestClass {
public static void main(String[] args) {
      System.out.println(“Whitehorses”);
  }
}
```

Dit zorgde meteen voor verwarring. Je wilt bij het leren van Java niet meteen beginnen met de uitleg van packages, klassen en methoden. Je wilt je kunnen focussen op het onderwerp waar je iets over wilt leren, in dit geval het wegschrijven een stuk tekst naar het scherm. Met *JShell* heb je aan de volgende regel code dan voldoende:

```
System.out.print(“Whitehorses”)
```

JShell maakt de leercurve van Java een stuk minder steil.

Experimenteren met API’s

Regelmatig heb je als programmeur te maken met slecht gedocumenteerde *API*’s. Dan is het omslachtig om een volledige applicatie te schrijven om te achterhalen wat voor methoden en attributen de API bevat en wat ze precies doen. *JShell* maakt het mogelijk een willekeurige methode aan te roepen en meteen te zien wat het resultaat is. Dit kan je blijven doen totdat je een goede indruk hebt van de mogelijkheden van de *API*.

Als voorbeeld gebruiken we de *Time Package* van *Apache Commons*, met daarin de *StopWatch*-klasse. Deze is redelijk goed gedocumenteerd, maar als voorbeeld is het prima te gebruiken. Deze stappen kun je volgen om met de *StopWatch*-klasse te experimenteren:

1. Download de *commons-lang JAR* (https://commons.apache.org/proper/commons-lang/download_lang.cgi)

2. Voeg de *commons-lang JAR* toe aan de *classpath* van *JShell*:

```
jshell> /env --class-path D://test-JShell/commons-lang3-3.6.jar
|  Setting new options and restoring state.
```

3. Importeer de *StopWatch*-klasse en maak een *StopWatch* aan:

```
jshell> import org.apache.commons.lang3.time.StopWatch
jshell> StopWatch.createStarted()
$3 ==> 00:00:00.000
|  created scratch variable $3 : StopWatch
```

   JShell maakt nu zelf een tijdelijke variabele aan van het type *StopWatch*.

4. Je kunt nu experimenteren met het *StopWatch*-object door zijn diverse methoden aan te roepen. Dit scheelt een hoop tijd in vergelijking met de oude methode waarbij je telkens een Java-bestand opnieuw zou moeten opslaan en compileren. Je zou bijvoorbeeld het volgende kunnen doen:

```
jshell> $3.suspend()

jshell> $3.isSuspended()
$6 ==> true
|  created scratch variable $6 : boolean

jshell> $3.getTime()
$9 ==> 521550
|  created scratch variable $9 : long

jshell> $3.resume()

jshell> $3.isStarted()
$8 ==> true
|  created scratch variable $8 : boolean
```

Zoals uit bovenstaand voorbeeld blijkt, maakt *JShell* het experimenteren met *API*'s erg eenvoudig.

Snelle *prototyping* van een stuk code

Soms moet je een lastig stuk code schrijven die je regelmatig moet testen en compileren voordat het helemaal goed is. Een goed voorbeeld hiervan is het schrijven van *regular expressions*. Met *JShell* kun je dit soort stukken code eenvoudig testen. Stel voor dat je in een gescand document wilt controleren of een geboortedatum van een persoon de juiste formattering heeft. Dit zou je kunnen nabootsen in *JShell*. Je simuleert de naam en geboortedatum door deze als String te declareren:

```
jshell> "Martijn Goor 27-06-1975"
$12 ==> "Martijn Goor 27-06-1975"
|  created scratch variable $12 : String
```

Daarna kun je er eenvoudig een *regular expression* op los laten.

```
jshell> $12.matches("\\w*\\s\\w*\\s*(\\d+)-\\d+-\\d+")
$13 ==> true
|  created scratch variable $13 : boolean
```

Wanneer je niet het gewenste resultaat krijgt druk je gewoon op de pijltje-toets-omhoog, past de code aan en drukt weer op enter. Op deze manier is Prototyping van code erg eenvoudig.

Conclusie

In dit Whitebook heb ik uiteen gezet hoe *JShell* werkt, welke commando's je in *JShell* kunt gebruiken en in wat voor soort situaties het gebruik van *JShell* handig kan zijn. Dit stuk is bij lange na niet uitputtend. Zo is er een *JShell* Java API die integratie van *JShell* in diverse populaire IDE's mogelijk maakt. Daarnaast zou je *JShell* kunnen gaan gebruiken voor bijvoorbeeld geautomatiseerd testen. Dit soort mogelijkheden moeten verder onderzocht worden wanneer *JShell* officieel uit is en door de community gebruikt gaat worden.
Met de release van *JShell* in Java 9 heeft Java zijn eerste officiële REPL gekregen, een command-line tool die Java statements kan draaien zonder ze te moeten bundelen in klassen of methoden. Door het toevoegen van *JShell* heeft Java wederom meer eigenschappen gekregen die men associeert met de meer functionele programmeertalen. In Java 8 gebeurde dit al door de toevoeging van *Lambdas* en *Streams*. Het is interessant om te volgen in hoeverre deze weg in de komende jaren verder wordt ingeslagen tijdens de ontwikkeling van Java 10.