Thuisplan

Tijdens J-Spring 2009 is al enige aandacht besteed aan de dit jaar officieel beschikbaar gestelde Java versie 7. Toen waren er nog aannames over de features die in deze nieuwe JDK beschikbaar zouden komen maar inmiddels is Java 7 'General Available' en dus kunnen we definitieve functionaliteiten uitlichten.

Java 7 Project Coin

Thijs Volders
Senior Java/SOA Consultant
& Architect bij Yenlo B.V.

Begin 2009 is een RFP programma opgestart waarin voorstellen voor wijzigingen in de syntax van de java-taal konden worden gedaan. De projectnaam van dit RFP programma is 'Project Coin'. Er zijn ongeveer 70 voorstellen aangeboden, waarvan een selectiecommissie er vijf heeft geselecteerd om officieel opgenomen te worden in Java 7. Deze syntax-wijzigingen zijn gebundeld in 'JSR334: Small Enhancements to the Java Programming Language'.
In dit artikel worden de features die onder de JSR334-vlag in de JDK terecht zijn gekomen behandeld.

'Strings in switch'

In 2004 is een bug ingediend op de bugtracker van Sun's Java 1.4.x release. Deze bug beschreef een feature om String literals in switch(...) statements te kunnen gebruiken.
Project Coin was het ideale podium om deze feature meer aandacht te kunnen geven. Het 'Strings in switch' proposal is door een Sun medewerker ingediend en met succes ontvangen.
Switch statements waren voorheen alleen mogelijk met de primitieve types van Java; byte, short, char, int maar ook enums als parametertype. Door de compiler een hashcode te laten genereren op basis van de String-literal die in de case-clausules is opgenomen, kan tijdens runtime de hashcode van de switch-waarde worden gegenereerd en vergeleken met de gecompileerde hashcode. In feite is dus string tijdens compileren als constant in de code opgenomen.
In pre-java 7 is complexe conditional-constructie noodzakelijk:

```
...
String game = myService.classifyGame();
if ("Grand Theft Auto".equals(game) || "Call of Duty".
equals(game)) {
         rating = VoilenceRating.HARD;
} else if ("Angry Birds".equals(game)) {
         rating = VoilenceRating.MEDIUM;
} else if ("Dora - The Explorer".equals(game)) {
         rating = VoilenceRating.LOW;
} else {
         rating = VoilenceRating.UNCLASSIFIED;
}
...
```

Met de komst van Java 7 kan dit worden vereenvoudigd in een Switch statement:

```
...
String game = myService.classifyGame();

switch (game) {
case "Grand Theft Auto" :
case "Call of Duty":
         rating = ViolenceRating.HARD;
         break;
case "Angry Birds":
         rating = ViolanceRating.MEDIUM;
         break;
case "Dora - The Explorer":
         rating = ViolanceRating.LOW;
         break;
default:
         rating = ViolanceRating.
UNCLASSIFIED;
}
...
```

Naast het feit dat het voornamelijk de code makkelijker leesbaar maakt is er verder weinig verschil met de pre-java 7 implementatie.
De ontwikkelaars van de JDK hebben rekening gehouden met bepaalde gevallen zoals strings waarin alleen unicode 0 characters (\u0000) zijn opgenomen.
Onderstaande code illustreert hoe wordt omgegaan met dergelijke unicode nul karakters.

```
private static int zeroHashes(String s) {
         int result = Integer.MAX_VALUE;
         switch(s) {
         case "": result = 0; break;
           case "\u0000": result = 1; break;
           case "\u0000\u0000": result = 2; break;
           case "\u0000\u0000\u0000": result = 3; break;
           default: result = -1;
     }
     return result;
}

public static void main(String[] args) {
         String zero = "";
     for(int i = 0; i <= 4; i++, zero += "\u0000") {
         int result = zeroHashes(zero);
           if (result != i) {
                    System.err.printf("Error: String
\"%s\" resulteerde in %d ipv %d.", zero, result, i);
                    }
         }
     if (zeroHashes("foo") != -1) {
         System.err.println("Failed to get -1 for input
string.");
         }
}
```

Daarnaast is het mogelijk om een gedeclareerde String constante te gebruiken in de switch statements in plaats van een literal.

ARM

Advanced Resource Management (ARM) is de volgende feature die met Project Coin in Java JDK 7 terecht is gekomen.
Om een resource correct af te sluiten moeten alle excepties die tijdens het afbouwen van de resource kunnen optreden worden afgehandeld. Ook in het geval dat er een onverwachte exceptie optreed is het noodzakelijk dat de resource wordt afgesloten. Dit wordt veelal geïmplementeerd als een try-finally statement. De finally clausule wordt immers ook uitgevoerd als een niet opgevangen exceptie optreed.
Neem onderstaand code-voorbeeld:

```
…
Connection connection;
Statement statement;
try {
         connection = dataSource.
getConnection();
         statement = connection.
createStatement();
         …
         } finally {
                   statement.close();
                   connection.close();
         }
...
```

Op meerdere plaatsen in dit code voorbeeld kunnen excepties optreden. NullPointerExceptions als het statement of de connection variabele niet correct zijn geïnitialiseerd en Exceptions door vele andere redenen. Als er een exceptie binnen het try-block optreedt en vervolgens treedt er een exceptie op bij het sluiten van de resources dan zal deze laatste overleven en als root-cause worden doorgegeven. Alleen uitgebreide code die detecteert dat er al een exceptie was opgetreden, zal ervoor kunnen zorgen dat de echte rootCause als exceptie wordt opgegooid. Veel voorkomende implementaties sluiten de connecties binnen het try-block en bij het optreden van een exceptie wordt deze genegeerd.
Het resultaat kan zijn dat het resource-object alsnog in de JVM achterblijft en als de JVM wordt gesloten, voordat alle resources door de garbage collector kunnen worden gesloten. Dan kunnen connecties open blijven of files worden niet netjes gesloten.

In Java 7 kan dit probleem worden aangepakt met ARM of ook wel try-with-resources genoemd. Door het openen van resources als compound-statement in een try-block te definiëren kan aan de compiler worden aangegeven dat resources gesloten moeten worden.
Resource classes implementeren nu een AutoClosable interface waardoor de Compiler de resources automatisch kan sluiten.

```
static void copy(String src, String dest) throws
IOException {
         try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dest))
{
         byte[] buf = new byte[8192];
         int n;
         while ((n = in.read(buf)) >= 0) {
                   out.write(buf, 0, n);
         }
     }
}
```

In dit voorbeeld worden de InputStream alsook de OutputStream automatisch gesloten door de JVM.
Overigens is er ook een oplossing voor het dubbele exceptie probleem. Het is nog steeds mogelijk dat binnen het try-block een exceptie optreedt en tijdens het sluiten van de resource (binnen de compiler-gegenereerde code) nog een.
De compiler regelt hiervoor echter dat de originele exceptie (die was opgetreden binnen het try-block) als een 'suppressedException' op de Throwable instance beschikbaar wordt gesteld.
De ontwikkelaar kan nog steeds op excepties reageren die in het try-block of binnen de compiler toegevoegde code gegenereerd worden.
Door het try-block met een catch aan te vullen kan een exceptie opgevangen worden. De exceptie die tijdens het sluiten van een InputStream gegenereerd wordt kan dan in het catch-block afgehandeld worden.
Ter illustratie een voorbeeld:

```
static void copy(String src, String dest) throws
IOException {
         boolean done = false;
         try (InputStream in = new FileInputStream(src)
{
      try(OutputStream out = new FileOutputStream(dest))
{
                   byte[] buf = new byte[8192];
                   int n;
                   while ((n = in.read(buf)) >= 0)
                        out.write(buf, 0, n);
              }
              done = true;
         } catch(IOException e) {
              if (!done)
                   throw e;
         }
     }
```

In bovenstaand voorbeeld wordt de exceptie doorgegeven als het try-block nog niet klaar was met verwerking.

Type-inference for Generic Instance Creation

Een hele mond vol maar in feite betekend het 'laat de compiler type-parameters afleiden indien een instance-creatie met generics-definities plaatsvindt' ofwel de 'Diamond operator'. Toen Generics in de Java-syntax werd opgenomen werd er gemengd gereageerd op deze functionaliteit. Generics zijn enerzijds lastig te doorgronden maar anderzijds heel krachtig. Inmiddels is de werkwijze met Generics voor de gemiddelde javaan duidelijk en kan men ermee werken en er het voordeel van inzien.

Ook ARM is met Project Coin in Java JDK 7 terecht gekomen.

Een nadeel van Generics in Java is dat het om uitgebreide code vraagt.

Een nadeel echter aan Generics in de Java-taal is dat het een nogal uitgebreide manier van coderen teweeg brengt. Een voorbeeld:

```
Map<String, Collection<String>> myMap = new HashMap<String,
Collection<String>>();
```

In deze code-regel moeten de object-types twee keer worden aangegeven in de Generics type-parameters.
Dit lijkt overbodig waardoor een voorstel is ingediend om de compiler automatisch de type-parameters te laten bepalen wanneer dat mogelijk is.
De Diamond-operator functionaliteit zorgt ervoor dat deze code-regel korter kan worden gebruikt

```
Map<String, Collection<String>> myMap = new HashMap<>();
```

Tijdens het definiëren van het voorstel is gesproken over het geheel weglaten van de <> karakters.
Probleem hiermee echter is dat raw-types nog steeds bestaan in huidige code en het weglaten van deze definitie suggereert een raw-type. Afleiding van het type is in dat geval niet mogelijk.
Een test die is uitgevoerd door een ontwikkelaar van de java-compiler heeft aangetoond dat in ongeveer 80 à 90 procent van de gevallen de dubbele notatie overbodig is.

Versimpelde varargs methode invocatie

Ook dit is een verbetering welke voortkomt uit problemen die vaak door ontwikkelaars worden omzeild. Ditmaal aan de manier waarop met varargs wordt omgegaan.
In bepaalde gevallen in pre-java 7 code wordt een compiler waarschuwing gegeven omdat de compiler niet juist kan bepalen van welke type de parameters zijn en er mogelijk een type-conflict optreed.
Bijvoorbeeld:

```
...
List<String> orders = Lists.newArrayList("orderId",
"orderId2");
List<String> adresregels = Lists.newArrayList("Straat 1",
"1000XX Amsterdam");
Lists.newArrayList(orders, adresregels);
...
```

De compiler zal voor deze code-regels op de laatste regel een waarschuwing geven.

```
TestClass.java:23: warning: [unchecked] unchecked generic
array creation of type java.util.List<java.lang.String>[]
for varargs parameter Lists.newArrayList(orders,
adresregels);
```

De melding is wat vreemd omdat, kijkend naar de source-code, geen 'generic array' gecreëerd wordt.
Door de toevoeging van Generics-functionaliteit aan de Java-syntax is het niet toegestaan om arrays te creëren van object-types waarvan de inhoud niet vast-gedefinieerd is, zogeheten non-reifiable object-types.
Een array definiëren als ArrayList<String>[] lijst = new ArrayList<String>[10]; is hierom niet toegestaan. Als dit zou worden toegestaan dan kan op runtime (door het toepassen van type-erasure) niet gegarandeerd worden dat het verwachte object-type in de lijst aanwezig is.
In het bovenstaand voorbeeld zou de compiler strict genomen een foutmelding moeten geven. De ontwikkelaar kan echter niets veranderen aan hoe de compiler de code aanpast waardoor de compiler dit als een waarschuwing mededeelt aan de ontwikkelaar.
De compiler genereert namelijk voor bovenstaand voorbeeld de volgende code:

```
List<String> orders = Lists.newArrayList(new String[]
{"orderId", "orderId2"});
List<String> adresregels = Lists.newArrayList(new String[]
{"Straat 1", "1000XX Amsterdam"});
Lists.newArrayList(new java.util.List<String>[] {orders,
adresregels});
```

De laatste regel is niet toegestaan volgens de Java-syntax regels en de compiler genereert de waarschuwing.
Het voorstel voor 'versimpelde vargargs methode invocatie' verplaatst deze waarschuwing naar de declaratie van de potentieel problematische varargs constructie. Een annotatie is toegevoegd om deze waarschuwing zelfs op de nieuwe locatie te onderdrukken.
Onderdrukking van deze waarschuwing was ook al mogelijk met @SuppressWarnings("unchecked")
Het voordeel van deze nieuwe feature is echter dat @SafeVarargs kan worden aangebracht op de plaats waar de varags constructie gedefinieerd wordt in plaats van bij ieder gebruikt.

```
@SafeVarargs
public static void determineSmthn(List<String>...
stringLists) {
…
}
```

Door de @SafeArgs annotatie op de methode te plaatsen zorgt de ontwikkelaar ervoor dat de waarschuwing onderdrukt wordt. Veelal gaat dit gepaard met een extra controle binnen de methode op het juiste type van de varargs inhoud.

Multi-catch

In Pre-java 7 code is het een probleem om meerdere excepties op eenzelfde wijze af te handelen zonder concessies te doen. Deze concessies zijn enerzijds code-duplicatie en anderzijds een gedeelde super-exceptie afhandelen.
Dit laatste is vooral problematisch omdat meer dan de bedoelde excepties worden opgevangen en de afhandel-code moet alle niet-interessante excepties opnieuw opgooien.

```
try{
} catch (LastOwnerException e) {
        // exceptie afhandeling
} catch (NotOwnerException e) {
```

```
        // dezelfde exceptie afhandeling
}
```

of

```
try{
} catch (Exception e) {
        if (e instanceof LastOwnerException || e
instanceof NotOwnerException) {
// exceptie afhandeling
} else {
        throw e;
    }
}
```

Door in een try-catch block meerdere exceptie-types in de catch-definitie te kunnen opnemen wordt het mogelijk om eenzelfde afhandeling voor meerdere exceptie-types te maken.

```
try{
} catch (LastOwnerException | NotOwnerException e) {
        // exceptie afhandeling
}
```

Naast deze code-style verbetering is de bepaling van het exceptie-type ook verfijnd. Indien een exceptie in pre-java 7 code opnieuw doorgegooid werd, dan werd ook het Exceptie-type aangepast.
Als een super-class van een exceptie werd opgevangen in een catch en opnieuw werd doorgegooid dan werd het type van de exceptie aangepast naar dit super-type.

```
try{
        owner.deleteOwner(caller, owner); // genereert
LastOwnerException
} catch (Exception e) {
        throw e;
}
```

In dit voorbeeld wordt in pre-java7 een Exception doorgegooid en niet de subclass LastOwnerException. In Java7 zal nu een LastOwnerException worden doorgegooid waardoor betere afhandeling in super-classes mogelijk is. De super-class kan nu immers ook alleen de LastOwnerException afhandelen en de rest negeren.

Binaire literals en underscores in nummers

In Java was het voorheen mogelijk om decimale, hexadecimale of octale getallen als literals te definiëren.
Het gebruik van bitmasks werd hierdoor bemoeilijkt omdat deze bitmasks eerst naar een van de andere notatievarianten moest worden omgezet.
Java 7 voegt een notatie toe voor binaire getallen.

```
// Een 8-bit 'byte' literal.
byte eenByte = (byte)0b00100001;
byte andereByte = (byte) 0B01011110;
```

Naast de binaire notatie is ook een kleine verbetering doorgevoerd in het leesbaar maken van getallen. Grote getallen zijn vaak lastig leesbaar.

```
Long bignumber = 10998234989823498L;
```

Door de scheidingstekens toe te voegen wordt het mogelijk om nummers op te delen. Omdat punten in nummers een andere betekenis heeft moest er een andere oplossing gevonden worden om nummers te scheiden.
In constante definities worden vaak underscores gebruikt om woorden van elkaar te scheiden en het gevolg is dat dit zijn weg heeft gevonden naar nummer literals. Bijvoorbeeld:

```
Long bignumber = 10_998_234_989_823_498L;
```

Het is overigens op dit moment niet ondersteund om Strings met underscores naar hun getal-equivalent te converteren.
De classes in java.lang kunnen hier (nog) niet mee omgaan.

```
Integer.valueOf("1234_0001"); // genereert
NumberFormatException
```

Conclusie

Project Coin was een community gedreven proces. Iedereen kon vorrstellen indienen om uitbreidingen op de syntax van Java aan te dragen. Om het mogelijk te maken dat een voorstel werd opgenomen in Java7 was het noodzakelijk dat de implementatie ervan relatief eenvoudig mogelijk moest zijn.
Een aantal voorstellen kreeg vele reacties in de mailinglist, anderen werden direct van tafel geveegd.

De voorstellen die uiteindelijk zijn geselecteerd en geïmplementeerd werden in dit artikel besproken. Deze voorstellen dragen allemaal bij aan meer flexibele implementaties. Het zijn duidelijk geen pogingen om drastische wijzigingen in de syntax van Java te implementeren.
Er zijn wijzigingen in nieuwe Java versies geweest die lastiger te begrijpen of toe te passen waren en de wijzigingen die via Project Coin aan Java7 zijn toegevoegd zullen denk ik snel door ontwikkelaars gebruikt worden.

Wijzigingen in Project Coin zullen snel door de ontwikkelaar worden gebruikt.

Referenties:

http://openjdk.java.net/projects/coin/