



ONDERZOEKSRAPPORT

JAVA 8

Lars Nijveld
Jennes Rutten
Mick Soudant
Rachèl Heimbach
Lex van de Laak
Brendan Steijn

Versie	Datum	Wijzigingen	Verantwoordelijke
0.1	13-11-2014	Concept versie	Lars
0.2	02-12-2014	Opzetten nieuwe alpha versie	Lars
0.3	08-12-2014	Verwijderen van alle content vanwege nieuwe versie op Google Drive	Lars
0.4	09-12-2014	Samenvoegen van Google Drive en Word document	Lars, Mick
1.0	06-01-2015	Verwerking feedback	Lars, Mick, Lex, Rachèl

Managementsamenvatting

Probleemstelling

Onderwerp

Java 8 biedt veel nieuwe functionaliteiten. Veel van deze nieuwe functionaliteiten zijn onder vuur komen te liggen door de softwarecommunity. Hier worden sceptische vragen gesteld zoals: “Zijn ze van toevoeging?” en “Horen ze wel thuis in Java?”. De Hogeschool van Arnhem en Nijmegen wil een onderzoek uitvoeren om deze vragen te beantwoorden en te bepalen of Java 8 geschikt lesmateriaal is voor het onderwijs.

Doelstelling

Na afloop van de projectperiode wil Java 8 Research Initiative een product opleveren waarmee informatica studenten en/of docenten inzicht kunnen krijgen in de voor- en nadelen van Java 8, in vergelijking met Java 7. Daarnaast moeten de studenten en/of docenten na het lezen van dit document de nieuwe functionaliteiten van Java 8 kunnen toepassen.

Vraagstelling

De hoofdvraag die in dit onderzoeksrapport beantwoord zal gaan worden luidt: “Wat is de toegevoegde waarde van de nieuwe functionaliteiten van Java 8 tegenover Java 7?”

Gebruikte onderzoeksmethoden & onderzoekstechnieken

Er is voor dit project gekozen om gebruik te maken van scrum. Hierbij is er besloten om vrijwel alle aspecten van scrum over te nemen, met uitzondering van een aantal onderdelen. Wel voeren we de sprint review met de opdrachtgever uit. De keuze is op scrum gevallen omdat iteratief werken aansluit op de door het projectteam beoogde onderzoekswijze, voornamelijk in de latere fase van het project. Bepaalde aspecten van scrum, zoals de daily stand-ups, bieden daarnaast veel structuur en bewaken de voortgang.

Daarnaast zijn er verschillende onderzoekstechnieken toegepast. Voor het beantwoorden van de verschillende vragen is er ten eerste gebruik gemaakt van code demo's en enquêtes. Daarnaast zal er regelmatig gerefereerd worden naar externe bronnen zoals boeken, artikelen en internetbronnen.

Resultaten

Op dit moment niet van toepassing.

Conclusie

Algemeen

Java 8 heeft veel nieuwe functionaliteit gekregen die, mits goed toegepast, de leesbaarheid en productiesnelheid bevorderen. Elke functionaliteit blijkt echter ook zijn nadelen te hebben. Lambda expressies zijn bijvoorbeeld moeilijk te testen en streams winnen niet altijd performance. Het is van belang om deze functionaliteit niet blind toe te passen zonder bekend te zijn met de nadelen die ze mee kunnen brengen, anders zal de winst waarschijnlijk verloren gaan.

Annotations

De meerwaarde van repeatable annotations bestaat uit een kleine verbetering in leesbaarheid en gebruiksgemak. Een gebruiker hoeft niet zoals bij elke toepassing van repeatable

annotations in Java 7 opnieuw de wrapper ervoor te declareren. Ook kunnen deze met één functieaanroep uitgelezen worden. Tegenover het uitlezen van de gedeclareerde container en hier de repeatable annotations van uit te lezen. Type annotations kunnen gebruikt worden om compile time validatie toe te voegen met behulp van een framework en/of een eigen implementatie.

Bij het onderdeel performance is vastgelegd dat het uitlezen van repeatable annotations d.m.v reflection nog steeds een erg dure operatie is, maar dat komt meer vanwege het verplichte gebruik van reflection. Repeatable annotations of Annotations in het algemeen, zijn alleen uit te lezen met reflection wanneer ze een `@Retention(RUNTIME)` boven hun declaratie hebben. (Java reflection annotations, 2015)

Lambdas

Lambda brengt behoorlijk veel voor- en nadelen met zich mee. Zo is het dankzij de introductie van lambda expressies niet langer nodig om uitgebreide code te schrijven om uiteindelijk een enkele abstracte methode te implementeren. Dit kan nu worden opgelost met een enkele regel code.

Daarnaast heeft lambda een aantal grote nadelen, waaronder bijvoorbeeld de beperking van testbaarheid of het kunnen hergebruiken van een lambda. Gelukkig heeft Oracle hier goed over nagedacht en hebben zij ervoor gezorgd dat deze nadelen grotendeels weggenomen kunnen worden door gebruik te maken van Method References.

Streams

Streams kunnen handig zijn, maar hebben een paar regels die gehanteerd moeten worden. De code is korter ten opzichte van voorgaande alternatieven, wat de leesbaarheid enorm bevordert, maar de performance is niet altijd beter. Bij gebruik van streams is het dus belangrijk om te kijken of de performance acceptabel is voor de machine waar deze toegepast wordt, voor deze in de realiteit worden toegepast.

Functional Interfaces

De nieuwe functionaliteit van Java 8 wat betreft functional interfaces zorgt voor aanzienlijk minder benodigde code. De SAM interfaces uit de voorgaande versies zijn compatible gemaakt met de nieuwe lambda notatie en zijn ook verkort wat betreft de syntax. De functionaliteit van de functional interfaces heeft de taal meer in de richting van functional programming gestuurd.

Optionals

Optionals zijn een waardevolle toevoeging aan Java, deze mening kan echter verschillen per persoon. Wat leesbaarheid betreft is het bijvoorbeeld niet per definitie duidelijker of minder duidelijk, maar meer een persoonlijke voorkeur. Zowel het gebruik van Optionals en nested if-statements heeft namelijk zijn voor- en nadelen.

De voordelen van het daadwerkelijk toepassen van een Optional worden pas duidelijk wanneer deze gebruikt wordt als returnwaarde van een functie. Op die manier hoeft er namelijk geen rekening gehouden te worden met het terugkrijgen van een null-waarde en kan het aantal NullPointerExceptions daadwerkelijk verminderd worden. Helaas is juiste implementatie van Optionals wel wat aan de lastige kant, aangezien deze in eerste instantie redelijk lastig te begrijpen zijn.

Hoofdvraag

Betreft de vraag of deze functionaliteiten een goede toevoeging geven en of deze wel thuis horen in Java 8, kan er worden gezegd dat deze wel degelijk goed bevonden worden. Vrijwel alle huidige talen ondersteunen over het algemeen één of meer van de nieuwe features in Java 8 waardoor Java in eerste instantie een achterstand had op de andere talen. Met de introductie van Java 8 en zijn features wordt de taal in positieve zin uitgebreid. Zo hebben veel mensen gewacht op de lambda introductie in Java en is dat met deze release goed meegenomen.

De implementatie lijkt echter af en toe aan de wensen over te laten. Zo heerst er binnen de Java community nog wel twijfels over de manier waarop bijvoorbeeld lambda is geïmplementeerd, maar werkt het op de manier waarop het zou moeten.

Inhoudsopgave

Managementsamenvatting	2
Probleemstelling.....	2
Vraagstelling.....	2
Gebruikte onderzoeksmethoden & onderzoekstechnieken	2
Resultaten	2
Conclusie	2
1 Inleiding	8
Probleemstelling.....	8
Onderzoeksvraag	8
Deelvragen	8
Strategie en methoden	10
Opbouw rapport.....	10
2 Algemeen.....	11
Wat is er allemaal toegevoegd met Java 8?	11
Op welke factoren wordt de toegevoegde waarde van functionaliteit gebaseerd?	11
Welke afwegingen zijn interessant voor het implementeren van Java 8 in een course?	12
Hoe hebben andere programmeertalen Java beïnvloed?	12
Hoelang zal Java 8 relevant blijven?	12
Waarom zijn de nieuwe functionaliteiten toegevoegd?	13
Welk commentaar is er op Java 8 vanuit de community?	13
Welke voordelen of nadelen heeft Java 8 m.b.t. de leesbaarheid van de code?	14
Welke invloed heeft Java 8 op applicaties die nog op Java 7 of ouder draaien?	14
Wat zijn de verschillen in performance op oudere systemen?	14
3 Annotations	15
3.1 Algemeen	15
3.2 Performance	19
4 Lambda & Method References.....	22
4.1 Algemeen	22
Wat is lambda?	22
Hoe is een lambda opgebouwd?	22
Wat zijn method references?	23
4.2 Performance	24

4.3 Onderhoud & testbaarheid.....	24
Wat zijn de voor- en/of nadelen van het onderhoud van lambda?	24
In welke situaties komt lambda wel goed van pas?	26
Unit testen met lambda	28
Variabelen in lambda	28
4.4 Syntax & Leesbaarheid.....	29
Op welke manieren kan de duidelijkheid van een functie bewaard worden, ondanks het onbreken van de functienaam?	29
4.5 Overig.....	30
Welke aanpassingen zijn er nodig geweest voor de ondersteuning van lambda's?	30
Hoe is lambda te vergelijken met andere programmeertalen?	30
4.6 Conclusie.....	31
5 Streams	32
5.1 Algemeen	32
5.2 Performance.....	33
Hoe stabiel zijn streams?	33
5.3 Onderhoud & Testbaarheid	36
5.4 Syntax & Leesbaarheid.....	36
5.5 Conclusie.....	37
6 Functional Interfaces.....	38
6.1 Algemeen	38
6.2 Functional interfaces.....	38
7 Optionals.....	45
7.1 Algemeen	45
Wat is een Optional?	45
Wanneer wordt een Optional gebruikt?	45
7.3 Leesbaarheid & Syntax.....	46
Hoe wordt een Optional toegepast?	46
7.2 Performance.....	47
Wat is het performanceverschil tussen Optionals en nested if-statements?	47
7.4 Overige.....	47
Waarom is het gebruiken van Optionals beter dan het controleren op null waarden?	47
7.5 Conclusie.....	48

8 Conclusie	49
Literatuurlijst	51
Bijlagen.....	54
Bijlage A - Lambda performance test eerste versie.....	54
Bijlage B - Lambda performance test aangepaste versie	54
Bijlage C - Lambda performance test resultaat	54
Bijlage D – Specificaties testsysteem M	58
Bijlage E - Annotations performance experiment 1	59
Bijlage F – Annotations performance experiment 2.....	60
Bijlage G – Opzet toepassen repeatable annotations voor unit testing	61
Bijlage H – Lambda expressies	61

1 Inleiding

Probleemstelling

Onderwerp

Java 8 biedt veel nieuwe functionaliteiten. Veel van deze nieuwe functionaliteiten zijn onder vuur komen te liggen door de softwarecommunity. Hier worden sceptische vragen gesteld zoals: “Zijn ze van toevoeging?” en “Horen ze wel thuis in Java?”. De Hogeschool van Arnhem en Nijmegen wil een onderzoek uitvoeren om deze vragen te beantwoorden en te bepalen of Java 8 geschikt lesmateriaal is voor het onderwijs.

Doelstelling

Na afloop van de projectperiode wil Java 8 Research Initiative een product opleveren waarmee informatica studenten en/of docenten inzicht kunnen krijgen in de voor- en nadelen van Java 8, in vergelijking met Java 7. Daarnaast moeten de studenten en/of docenten na het lezen van dit document de nieuwe functionaliteiten van Java 8 kunnen toepassen.

Onderzoeksvraag

“Wat is de toegevoegde waarde van de nieuwe functionaliteiten van Java 8, tegenover de functionaliteiten van Java 7?”

Deelvragen

Algemeen

- Wat is er allemaal toegevoegd aan Java 8?
- Op welke factoren wordt de toegevoegde waarde van functionaliteit gebaseerd?
- Welke afwegingen zijn interessant voor het implementeren van Java in een course?
- Hoe hebben andere programmeertalen Java beïnvloed?
- Hoelang zal Java 8 relevant blijven?
- Waarom zijn de nieuwe functionaliteiten toegevoegd?
- Welk commentaar is er op Java 8 vanuit de community?
- Welke voordelen of nadelen heeft Java 8 m.b.t. de leesbaarheid van de code?
- Welke invloed heeft Java 8 op applicaties die nog op Java 7 of ouder draaien?
- Wat zijn de verschillen in performance op oudere systemen?

Annotations

- Wat zijn repeatable annotations?
- Wat zijn type annotations?
- Hoe maak je repeatable annotations?
- Wat voor een verschil in performance is er tussen het gebruik van repeatable annotations in combinatie met reflection tegenover een alternatieve methode om data uit te lezen?

Lambda & Method references

- Wat is lambda?
- Hoe is een lambda opgebouwd?
- Wat zijn method references?

- Welk effect heeft het gebruik van lambda's tegenover publieke functies op de uitvoertijd van de code?
- Wat voor een effect heeft het gebruik van method references op de leesbaarheid van de code?
- Uit naslagwerk blijkt dat lambda lastig te onderhouden is. Wat zijn dan de aspecten hiervan die het lastig te onderhouden maken?
- Gezien alle bovenstaande 'negatieve' onderdelen van lambda, in welke situaties komt lambda dan toch goed van pas?
- Welke invloed heeft lambda op het unit testen van functionaliteiten?
- Bij het veranderen van lambda scope variabelen is er een kans dat de compiler een fout gooit op final en effectively final, wat is het verschil tussen deze twee?
- Op welke manieren kan de duidelijkheid van een functie bewaard worden, ondanks het ontbreken van de functienaam?
- Welke aanpassingen zijn er nodig geweest aan de taal, compiler en interpreter om lambda's te ondersteunen?
- Hoe is lambda binnen Java 8 te vergelijken met andere programmeertalen?

Streams

- Waarin onderscheiden streams zich van collecties?
- Welke sorteeralgoritmes gebruiken streams?
- Hoe stabiel zijn streams?
- Wat is het verschil in performance tussen parallel streams en serial streams?
- Wat is het verschil in performance tussen de Java 7 gebruikelijke join forks en/of threads tegenover parallel streams?

Functional Interfaces

- Wat is het verschil ten opzichte van de standaard Java interface instantiatie?
- Wat is het verschil ten opzichte van de standaard Java interface functionaliteit?
- Hoe kan men parameters meegeven aan een functional interface?
- Hoe worden waardes buiten de functional interface gebruik in een functional interface?
- Hoe werken de functional interfaces die zijn toegevoegd aan de Java 8 API?
- Hoe is de werking van de functional interfaces vergelijkbaar met functionele programmeertalen?

Optionals

- Wat is een Optional?
- Wat is het verschil in het gebruik van geheugen en/of processorkracht tussen optionals en nested if-statements?
- Optionals worden voornamelijk gebruikt om nullpointers weg te werken, echter wanneer is het wel of niet handig om een Optional te gebruiken?
- Wat is het effect op leesbaarheid op de syntax benodigd voor het afhandelen van een simpele nullpointer met Optional tegenover if-statements?
- Waarom is het gebruik van Optionals, die een NullPointerException kunnen geven, beter dan het controleren op null waarden?
- Waarom moet een Optional de waarde Optional.empty() krijgen en kan hij niet gewoon leeg gelaten worden?

Strategie en methoden

Er is voor dit project gekozen om gebruik te maken van scrum. Hierbij is er besloten om vrijwel alle aspecten van scrum over te nemen, met uitzondering van een aantal onderdelen. Een voorbeeld hiervan is de sprint review met de opdrachtgever. De keuze is op scrum gevallen omdat iteratief werken aansluit op de door het projectteam beoogde onderzoekswijze, voornamelijk in de latere fase van het project. Bepaalde aspecten van scrum, zoals de daily stand-ups, bieden daarnaast veel structuur en bewaken de voortgang.

Daarnaast zijn er verschillende onderzoekstechnieken toegepast. Voor het beantwoorden van de verschillende vragen is er ten eerste gebruik gemaakt van code demo's en enquêtes. Daarnaast zal er regelmatig gerefereerd worden naar externe bronnen zoals boeken, artikelen en internetbronnen.

Opbouw rapport

Dit rapport zal beginnen met een managementsamenvatting waarin een samenvatting zal komen van het document waarbij de belangrijkste punten van de resultaten zullen worden toegelicht met de daarbij behorende conclusies.

Gevolgd door een inleiding waarbij de deelvragen worden getoond die in dit rapport worden behandeld. Tevens wordt er vermeld met behulp van welke strategie en methode deze vragen onderzocht zullen worden.

Vervolgens zullen er diverse Java 8 aspecten afzonderlijk worden behandeld, waarbij er rekening is gehouden met elementen waaronder performance en leesbaarheid. Deze hoofdstukken zijn in een logische wijze ingedeeld zodat andere hoofdstukken op een voorgaand hoofdstuk kunnen voortbouwen. Zo wordt lambda zo vroeg mogelijk in het document behandeld omdat hoofdstukken zoals streams en functional interfaces gebruik maken van dit onderwerp.

Tenslotte is er een hoofdstuk voor conclusies (en eventuele aanbevelingen). In dit hoofdstuk zullen de resultaten worden bekeken waarbij er een afzonderlijke eindconclusie per onderwerp aanwezig zal zijn.

2 Algemeen

Wat is er allemaal toegevoegd met Java 8?

Java 8 heeft een aantal nieuwe features. Dit onderzoek behandelt de belangrijkste nieuwe features. Dit zijn:

1. Lambdas
2. Method references
3. Default methods
4. Repeating annotations
5. Type annotations
6. Streams
7. Functional interfaces
8. Optionals

Deze onderdelen worden verder in dit document gedetailleerd toegelicht en behandeld.

Op welke factoren wordt de toegevoegde waarde van functionaliteit gebaseerd?

Aan het begin van het onderzoek is besloten dat er specifieke factoren gehanteerd moesten worden om de toegevoegde waarde van een functionaliteit te kunnen bepalen. Uit overleg is gebleken dat de belangrijkste nieuwe functionaliteiten zullen worden afgewogen tegenover de volgende drie factoren:

1. Performance

Bij performance zal er worden bekeken welke performancewinst of -verlies er behaald is. Dit wordt onderzocht met behulp van demo projecten die, indien nodig, gebruik maken van frameworks. Tevens wordt er gebruik gemaakt van websites waarop diverse aspecten worden toegelicht.

2. Onderhoud en testbaarheid

Testbare code schrijven die tevens goed onderhoudbaar is, is een belangrijk aspect van goede softwareontwikkeling. Je bent in de meeste gevallen niet de enige software ontwikkelaar die met die specifieke code aan de slag gaat. Om deze reden is er besloten om deze factor mee te nemen in het onderzoek. De mate van onderhoud en testbaarheid zal worden aangetoond met behulp van codevoorbeelden en eventueel naslagwerk zoals websites en boeken.

3. Leesbaarheid en syntax

Om de verbeteringen van de leesbaarheid en syntax aan te kunnen tonen is er gekozen om dit te doen met behulp van codevoorbeelden, naslagwerk en enquêtes.

Welke afwegingen zijn interessant voor het implementeren van Java 8 in een course?

De waarde van het implementeren van Java 8 in een course is afhankelijk van persoonlijke leerdoelen en de leerduur van onderdelen. Daarom is aan te raden om deze vast te stellen en met behulp van dit rapport de functionaliteiten te selecteren die het meeste aansluiten op het leerdoel.

Veel onderdelen van Java 8 zijn erg taal specifiek. Zo zal veel verworven kennis niet veel toegevoegde waarde hebben buiten Java 8. Echter kunnen sommige onderdelen wel toegepast worden als middelen om een hogere abstractie aan kennis te over te dragen. Bijvoorbeeld *lambdas* en *method references* kunnen toegepast worden om functional programming te leren aan studenten. Streams kunnen gebruikt worden in combinatie met *lambda*'s, met als voordeel de mogelijkheid om data te manipuleren met weinig syntax. *Method references* worden voornamelijk gebruikt in combinatie met *lambdas*. Hierdoor is het wellicht interessant om een opdracht op te zetten waar deze functionaliteiten allemaal gebruikt dienen te worden om het gewenste resultaat te bereiken.

Repeatable annotations zijn een kleine toevoeging waarbij hooguit vermeld hoeft te worden dat het mogelijk is om ze te gebruiken. Hetzelfde geldt voor default methods, optionals en type annotations. De leerduur van elk van deze onderdelen is relatief tegenover *lambdas*, streams en constructor references aanzienlijk lager (Op basis van de ervaringen van het onderzoeksteam). Dit omdat deze onderdelen geheel op zichzelf staan en het gebruik van een van deze onderdelen geen kennis over de andere onderdelen vereist.

Tot slot hebben we de date/time API. Dit onderdeel is niet behandeld in dit onderzoek vanwege de lage prioriteit. Echter is het zeker interessant om aan te geven dat deze bestaat.

Hoe hebben andere programmeertalen Java beïnvloed?

De programmeertaal die de grootste invloed heeft gehad op Java is C. Dit komt doordat Java oorspronkelijk gebaseerd is op C zodat software ontwikkelaars onder andere de syntax als bekend aan zouden voelen (Once Upon an Oak, 2014).

Daarnaast hebben andere talen een indirecte invloed op Java, een voorbeeld hiervan is Scala waarop *lambdas* gedeeltelijk gebaseerd zijn. Dit komt omdat de community altijd een verzoek kan indienen voor een specifieke functionaliteit bij de Java Community Project (The Java Community Process(SM) Program, 2014). Als er vervolgens een serieuze vraag is naar een bepaalde functionaliteit, zal dit door het Java team worden toegevoegd aan de backlog. De community kan vervolgens worden geraadpleegd over de belangrijkste functionaliteiten.

Deze manier van werken lijkt bekend te zijn binnen Oracle. Tijdens JFall in november 2014, bleek dat Java 8 SE functionaliteiten op deze manier tot stand waren gekomen.

Hoelang zal Java 8 relevant blijven?

Java versies blijven over het algemeen nog een paar maanden (Teruggaan naar Java 7 na installatie van Java 8, 2014) relevant wanneer er een nieuwe versie is uitgebracht. Zo is met de release van Java 8, Java 6 niet langer een valide versie en zal deze alleen ondersteunt worden

door Oracle als een bedrijf bereid bent om voor bug fixes te gaan betalen. Java 7 is daarentegen nog tijdelijk relevant, dit komt doordat Oracle begin 2015 een automatische update uitvoeren zodat alle Java versies gelijk zijn met de laatste release (Java Tester - What Version of Java Are You Using?, 2014).

Met de release van Java 9, zal Java 8 dan ook nog maar een paar maanden relevant blijven. Het advies is dan ook om een upgrade uit te voeren naar de laatste versie vanwege de 'security bug fixes' die doorgevoerd zullen worden in deze versie.

Waarom zijn de nieuwe functionaliteiten toegevoegd?

Bij een release van een nieuwe Java versie, wordt er niet alleen gekeken naar de benodigde reparaties die doorgevoerd moeten worden, maar ook naar wat de community graag terug ziet komen in de volgende versie. Hiervoor bestaat er een groep genaamd JCP (Java Community Process).

De leden van JCP, waar overigens iedereen lid van kan worden, kunnen naast het feedback geven op feature requests, meerdere features aanvragen door middel van een JSR (Java Specification Request) (The Java Community Process(SM) Program, 2014).

Wanneer er vervolgens een grote vraag is naar een functionaliteit, zal de community geraadpleegd worden om te gaan stemmen welke functionaliteiten het belangrijkste worden bevonden en dus een hogere prioriteit hebben qua implementatie.

Op basis van deze werking, bleek er dan ook onder andere een grote vraag naar lambda expressies binnen Java. Waarbij Oracle vervolgens de benodigde aanpassingen heeft gemaakt aan de JSR om het juist te kunnen implementeren naar hun visie.

Welk commentaar is er op Java 8 vanuit de community?

Het commentaar op Java 8 vanuit de community is zowel positief als verward en negatief. Dit komt onder andere door de impact op de leesbaarheid van de code. De geïntroduceerde stream is een zeer handige functionaliteit met betrekken tot onder andere parallel streams dat beter gebruik maakt van de beschikbare processoren. Deze wordt geweldig bevonden vanwege het optimale processorgebruik, maar blijkt dat de community ook vindt dat doordat je 'chaining' krijgt, je de leesbaarheid van je code opoffert.

Daarnaast blijkt er veel liefde voor lambda omdat het de benodigde aantal regels voor een functionaliteit verkleint en voor meer leesbaarheid kan zorgen. Er is echter wel ontevredenheid over de mate waarop lambda is geïmplementeerd. Zo hebben veel mensen dat het een slecht geïmplementeerde versie van lambda in Scala is. Deze ontevredenheid komt doordat de syntax bijna hetzelfde is als scala, maar met gebruik van `->` in plaats van `=>`. Daarbij komt dat de lambda in Java 8 matig is toegelicht (Love and hate for Java 8, 2014).

Overigens is uit mondelinge communicatie met diverse ontwikkelaars gebleken dat het mee kunnen geven van functies nogal matig is geïmplementeerd in Java 8. Deze twijfels komen voort uit vergelijking met andere talen zoals JavaScript waarbij het doorgeven van functies een 'dagelijkse activiteit' is.

Welke voordelen of nadelen heeft Java 8 m.b.t. de leesbaarheid van de code?

De impact van het gebruik van Java 8 functionaliteiten op de leesbaarheid van de code verschilt uiteraard per functionaliteit. Zo is er bij Annotaties slechts een wrapper verdwenen waardoor ze niet meer in deze context geplaatst hoeven te worden. Bij lambda wordt het echter snel onleesbaar wanneer de lambda te complex begint te worden.

Elke belangrijke functionaliteit zal afzonderlijk in dit rapport zo goed mogelijk worden behandeld op alle mogelijke vlakken. Vanwege deze diversiteit zal er momenteel dan ook niet verder worden ingegaan op de voor- en nadelen van de leesbaarheid.

Welke invloed heeft Java 8 op applicaties die nog op Java 7 of ouder draaien?

In principe ondersteunt de Java 8 runtime ook applicaties die gebaseerd zijn op oudere versies van Java, waaronder Java 7. Dit op een enkele uitzondering na. Oracle heeft een lijst beschikbaar gesteld met alle functionaliteiten die niet meer (volledig) worden ondersteund. Deze lijst is te vinden in de *Compatibility Guide for SDK 8* (Compatibility Guide for JDK 8, 2014).

Wat zijn de verschillen in performance op oudere systemen?

Het meest opmerkelijke op dit gebied is dat Windows XP met de komst van Java 8 niet wordt ondersteund (Oracle, 2014). Dit komt mede doordat Microsoft zelf ook is gestopt met de ondersteuning voor deze versie van Windows. Dit betekent dat Oracle niet meer garant staat voor dingen die eventueel niet meer zullen werken op deze verouderde versie van Windows.

De systeemeisen voor Java 7 en Java 8 zijn ook grotendeels hetzelfde (Wat zijn de systeemvereisten voor Java?, 2014). Java 8 vraagt echter in het geval van MAC OS X en Linux om een latere versie van het besturingssysteem. Daarnaast worden er latere versies aanbevolen van browsers, zoals Internet Explorer en Firefox.

3 Annotations

3.1 Algemeen

Wat zijn repeatable annotations?

Repeatable annotations is één van de nieuwe functionaliteiten van Java 8. Het biedt de mogelijkheid aan om een annotatie meerdere keren te declareren bij een klasse of functie. (Repeating annotations, 2014) (Repeating annotations in java 8, 2014) Als een ontwikkelaar in het verleden meerdere Authors wou aangeven bij een functie dan moesten deze binnen in een container annotatie verwerkt worden (Zie afbeelding 3.1.1)

```
@Authors({  
    @Author(name = "John"),  
    @Author(name = "George")  
})  
public class Book { ... }
```

Afbeelding 3.1.1 - Oude variant

Met de toevoeging van repeatable annotations kan deze declaratie zonder container annotatie. Zie (afbeelding 3.1.2)

```
@Author(name = "John")  
@Author(name = "George")  
public class Book { ... }
```

Afbeelding 3.1.2 - Nieuwe variant

Hoe maak je repeatable annotations?

Een Repeatable annotation bestaat uit twee annotation interfaces: Een interface van de annotatie zelf en een interface om meerdere annotaties in te bewaren. Deze tweede klasse wordt een container annotatie genoemd (Repeating annotations, 2014). Hieronder volgen twee codevoorbeelden. Het eerste voorbeeld is de Schedule annotatie (Afbeelding 3.1.3). Het tweede voorbeeld is de container annotatie.

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Schedules.class)
public @interface Schedule {
    String dayOfMonth() default "first";
    boolean repeatable() default true;
    int hour() default 12;
}
```

Afbeelding 3.1.3 - Schedule annotatie

Belangrijk bij dit codevoorbeeld is de `@Repeatable(Schedules.class)`. Deze annotatie geeft aan dat Schedule een repeatable annotatie is met als argument de klasse verwijzing van de wrapper klasse (Zie afbeelding 3.1.4) (Annotation type repeatable, 2014)

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Schedules
{
    Schedule[] value();
}
```

Afbeelding 3.1.4 - Container annotatie

Deze nieuwe functionaliteit komt gepaard met nieuwe reflection functionaliteiten om deze repeatable annotations makkelijk uit te kunnen lezen. (Repeating annotations, 2014)

1. `AnnotatedElement.getAnnotations(Class<T>)`
2. `getDeclaredAnnotations()`

Wat zijn type annotations?

Type annotations zijn net als repeatable annotations een toevoeging van Java 8. Het biedt software ontwikkelaars de mogelijkheid aan om annotaties te declareren voor praktisch elk onderdeel van de Java syntax. Bekijk afbeelding 3.1.5 voor een aantal voorbeelden van annotations.

Annotation Example	Meaning
<code>@NonNull List<String></code>	A non-null list of Strings.
<code>List<@NonNull String></code>	A list of non-null Strings.
<code>@Regex String validation = "(Java JDK)[7,8]"</code>	Check at compile time that this String is a valid regular expression.
<pre>private String getInput(String parameterName){ final String retval = @Tainted request.getParameter(parameterName); return retval; }</pre>	The object assigned to <code>retval</code> is tainted and not for use in sensitive operations.
<pre>private void runCommand(@Untainted String... commands){ ProcessBuilder processBuilder = new ProcessBuilder(command); Process process = processBuilder.start(); }</pre>	Each command must be untainted. For example, the previously tainted String must be validated before being passed in here.

Afbeelding 3.1.5 - Voorbeelden type annotaties (Java 8 new type annotations, 2014)

De bovenstaande annotations zijn niet native, Java 8 heeft geen implementaties van type annotations meegeleverd. Hiervoor wordt het zelf implementeren van deze type annotaties of het gebruik van een framework aangeraden. Oracle zelf vermeld het checkers framework. Dit framework is te uitgebreid om behandeld te worden binnen dit onderzoek. Maar wellicht interessant voor een vervolgonderzoek. (Type annotations?, 2014)

Om vast te leggen op welke elementen annotations gebruikt kunnen worden, wordt er gebruik gemaakt van een de `@Target` annotatie. Bij gebruik van deze annotatie wordt als parameter een waarde van een enumerator genaamd `ElementType` meegegeven. Deze enumerator bevat de volgende waarden: (Target annotation, 2014)

1. `ANNOTATION_TYPE`
2. `CONSTRUCTOR`
3. `FIELD`
4. `LOCAL_VARIABLE`
5. `METHOD`
6. `PACKAGE`
7. `PARAMETER`
8. `TYPE`
9. `TYPE_PARAMETER(1.8)`
10. `TYPE_USE(1.8)`

Deze namen kunnen in eerste instantie syntactisch incorrect overkomen. Veel waardes van deze enumerator gebruiken het woord TYPE en niet altijd in een consistente volgorde. ANNOTATION_TYPE heeft bijvoorbeeld het woord TYPE aan het einde van de naam staan tegenover TYPE_PARAMETER. (Element type, 2014)

Types zijn in deze context Java types (klassen, native types, enumerators, interfaces en parameters.

De waarde TYPE maakt het mogelijk om annotaties te declareren voor alle types exclusief annotaties. Waarbij ANNOTATION_TYPE het mogelijk maakt voor gebruik met andere annotaties.

De waarde TYPE_PARAMETER maakt het mogelijk om annotaties te declareren voor type parameters (beter bekend als generic type parameters). TYPE_USE staat de ontwikkelaar toe om annotaties toe te passen op alle syntax die gebruik maakt van types. Denk hierbij aan keywords zoals new, throws en implements.

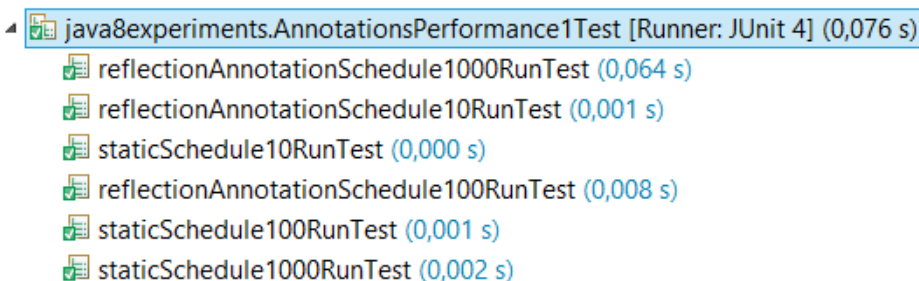
Voorheen werd gedrag zoals nullability in Javadoc verwerkt. Met type annotations is het mogelijk om dit gedrag in de bytecode te beschrijven. Deze kunnen vervolgens gebruikt worden voor compile time validatie, door zowel de Java compiler zelf of een externe tool (zoals sonar). Dit is geen alternatief voor runtime validatie, maar bedoelt om sneller fouten in de code te vinden die alleen tijdens run time plaatsvinden.

3.2 Performance

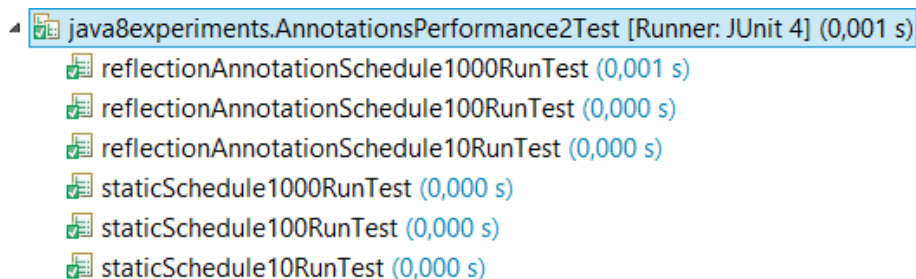
Wat voor een verschil in performance is er tussen het gebruik van repeatable annotations in combinatie met reflection tegenover het gebruik van een array van objecten om data uit het geheugen te lezen?

Het beantwoorden van deze vraag is interessant voor het gebruik van annotaties in combinatie van reflection. Bijvoorbeeld voor alle Schedule annotaties boven de createSchedule functie de createSchedule functie uitvoeren aan het begin van de applicatie. De Schedule annotatie is in deze context een eigen implementatie op basis van een voorbeeld.

Aangezien reflection functionaliteit erom bekend staat een negatieve invloed op performance te hebben, is er besloten om te kijken hoeveel performanceverlies er is bij het uitlezen van repeatable annotations. De resultaten worden aangetoond met twee experimenten. Deze experimenten worden nader beschreven en aangeleverd in bijlagen E en F.



Afbeelding 3.2.1 - Resultaten experiment één



Afbeelding 3.2.2 - Resultaten experiment twee.

Aan de resultaten van experiment E (zie afbeelding 3.2.1) is te zien dat het uitlezen van repeatable annotations rond de 100 runs al een factor van 6 aan executietijd heeft. Rond de 1000 runs zit het uitlezen van annotaties op 30 milliseconden, waarbij het uitlezen van objecten nog steeds op één milliseconde zit. Het 1000 maal uitvoeren van de functie om objecten uit te lezen (staticSchedule1000RunTest) doet er minder lang over dan het 100 keer uitvoeren van de functie die (repeatable)annotaties uitleest. (reflectionAnnotationSchedule100RunTest).

Aan de resultaten van experiment F (zie afbeelding 3.2.2) is te zien dat het uitlezen van meer repeatable annotations tegenover minder functieaanroepen veel performance efficiënter is. Hieruit kunnen wij de conclusie trekken dat de kosten in performance voornamelijk liggen in het opzoeken van de methode waar de annotaties vanaf gelezen moeten worden, tegenover het daadwerkelijk uitlezen van annotaties zelf.

Reflection is duur, waardoor het uitlezen van (repeatable) annotaties tijdens runtime een erg dure operatie is in tegenoverstelling tot het uitlezen van objecten. Echter kost het uitlezen van 10 repeatable annotaties tegenover 3 nauwelijks meer performance. Dus deze techniek is op basis van performance prima toe te passen op taken die eenmalig tot zeer weinig zoals uitgevoerd worden. Denk hierbij bijvoorbeeld aan initialisatie.

Kunnen repeatable annotations toegepast worden om unit tests meerdere keren uit te voeren?

Het beantwoorden van deze vraag heeft betrekking op innovatie. De bedoeling is om te achterhalen of in plaats van @Test annotaties om unit tests aan te duiden (binnen in het Junit framework), repeatable annotaties gebruikt kunnen worden. Hiermee zou de gebruiker een unit test meerdere keren kunnen uitvoeren op basis van hoe vaak een annotation gedeclareerd is.

Om deze vraag te beantwoorden wordt er gebruik gemaakt van een experiment (Bijlage G). Dit experiment is een mogelijke implementatie om het doel van deze vraag te bereiken. Meer informatie over de waardes en uitvoering van dit experiment zijn te vinden in de bijlage.

Dit experiment heeft als doel om unit tests te laten draaien door middel van repeatable @test annotaties. Het resultaat stelt vast dat dit mogelijk is, maar het resultaat zelf is niet de optimale uitwerking (op basis van persoonlijke mening).

Conclusie

Wat is de meerwaarde van repeatable annotations?

De meerwaarde van repeatable annotations bestaat uit een kleine verbetering in leesbaarheid en gebruiksgemak. Een gebruiker hoeft niet zoals bij elke toepassing van repeatable annotations in Java 7 opnieuw de wrapper ervoor te declareren. Ook kunnen deze met één functieaanroep uitgelezen worden. Tegenover het uitlezen van de gedeclareerde container en hier de repeatable annotations van uit te lezen. Type annotations kunnen gebruikt worden om compile time validatie toe te voegen met behulp van een framework en/of een eigen implementatie.

Bij het onderdeel performance is vastgelegd dat het uitlezen van repeatable annotations d.m.v reflection nog steeds een erg dure operatie is, Maar dat komt meer vanwege het verplichte gebruik van reflection. Repeatable annotations of Annotations in het algemeen, zijn alleen uit te lezen met reflection wanneer ze een @Retention(RUNTIME) boven hun declaratie hebben. (Java reflection annotations, 2015)

4 Lambda & Method References

4.1 Algemeen

Wat is lambda?

Lambda, dat is afgeleid uit Lambda Calculus (What is a lambda (function)?, 2014) (What is a lambda expression., 2014), maakt het mogelijk om anonieme functies te schrijven die je kunt gebruiken voor het maken van bijvoorbeeld 'delegate' types. Met het gebruik van lambda expressies, kun je hiermee lokale functies declareren die kunnen worden meegegeven als argument of die kunnen worden geretourneerd als uitkomst van een functie (Microsoft, 2014)

Hiermee stelt lambda je bijvoorbeeld in staat om een grotere functionaliteit in aanzienlijk minder regels code te schrijven.

Om bijvoorbeeld een lijst van alle personen van het vrouwelijke geslacht met een leeftijd tussen de 18 en 25 te printen, zou de volgende functie kunnen worden geschreven:

```
public Main() {
    List<Person> youngWomen = this.people.stream().filter( (p) -> p.getSex() == Person.FEMALE
        && p.getAge() >= 18
        && p.getAge() <= 25).collect(Collectors.toList());
    printList(youngWomen);
}

private void printList(List<Person> people) {
    people.forEach((p) -> System.out.println("Name: " + p.getName() + ", Age: " + p.getAge()));
}
```

Afbeelding 4.1.1 - Lambda expressie voor het filteren van een lijst op geslacht en leeftijd

Hoe is een lambda opgebouwd?

Veel talen zoals C++, C#, Scala, Clojure en nu Java 8, hebben anonieme functies op een eigen manier geïmplementeerd (Anonymous function, 2014). Door deze afwisselende implementatie bestaat er dan ook een verschil tussen de syntax benodigd om een anonieme functie te schrijven.

Bij Java 8 is de lambda op de volgende manier opgebouwd:

LambdaParameters '->' LambdaBody

Afhankelijk van het gebruik kun je de syntax op een logische of beknopte manier schrijven.

Indien je lambda expressie slechts een enkele parameter en functionaliteit heeft, zou je het op de volgende manieren kunnen schrijven (Bijlage H - Lambda Expressies):

person -> person.getName().toString()

Zou je echter een lambda expressie willen schrijven die meerdere regels beslaat, dan zal er gebruik gemaakt moeten worden van brackets. Bij het gebruik van meerdere parameters is het gebruik van ronde haken overigens aangeraden.

```
(person1, person2) -> {
    if(person1.getName().equals(person2.getName())) {
        person1.hasNameBuddy(true);
    }
}
```

Afbeelding 4.1.2 - Lambda met meerdere regels

Wat zijn method references?

Een Method Reference houdt in dat wanneer er een functie wordt verwacht, zoals een anonieme functie of een functional interface (Zie hoofdstuk functional interfaces), er een methode kan worden meegegeven.

Daarnaast zorgt Method Reference ervoor dat de nadelen van lambda, ongedaan worden. Echter kun je jezelf de vraag stellen of je wel Method Reference wil toepassen. Zoals beschreven (Java Lambdas Pragmatic Functional Programming, 2014), is het toepassen van method reference geen oplossing. De code die je in deze functie zou schrijven zou je namelijk ook in een normale functie kunnen schrijven. Dit zou de Lambda functionaliteit overbodig maken.

Zie onderstaande afbeelding voor een voorbeeld van method reference.

```
public static Integer addOneToValue(Integer number) {
    return number / 0;
}

public void methodLiftingExample() {
    final List<Integer> numbersPlusOne =
        numbers.stream().map(Main::addOneToValue).collect(Collectors.toList());

    print(numbers);
    print(numbersPlusOne);
}
```

Afbeelding 4.1.3 - Method Lifting voorbeeld

De bovenstaande code zal echter een `DivideByZeroException` gooien, wat gerelateerd is aan een probleem met de stacktrace, dit zal verder worden behandeld bij de vraag "Wat zijn de voor- en/of nadelen van het onderhoud van lambda?".

Er zijn vier soorten method references.

Binnen de definitie van method references (in Java8) bevinden zich vier varianten (Method references, 2014). Dit zijn:

Variant	Voorbeeld
Referentie naar een statische methode	<code>ContainingClass::staticMethodName</code>
Referentie naar een instantie methode van een specifiek object.	<code>containingObject::instanceMethodName</code>
Reference to an instance method of an arbitrary object of a particular type	<code>ContainingType::methodName</code>
Reference to a constructor	<code>ClassName::new</code>

Wat is Method Lifting?

Uit gesprek met Urs Peters tijdens het J-Fall 2014 evenement, bleek dat het concept van Method Lifting enorm breed is. Zo is Method Reference één van de kenmerken dat Method Lifting hanteert. Tevens is Method Lifting niet behandeld tijdens het evenement, en is enkel Method Reference behandeld als oplossing om Lambda goed te kunnen testen. Om deze reden zal method lifting dan ook verder buiten beschouwing worden gelaten.

4.2 Performance

Om performance te kunnen testen, is het gemakkelijk om hiervoor een klein programma te schrijven (zie bijlage A). Tijdens de uitvoer kwam het onverwachte resultaat dat lambda twee tot zes keer zo inefficiënt was betreft het filteren van een korte lijst bestaande uit tien elementen, en lichtelijk efficiënter bij grotere lijsten met ongeveer 200 elementen. Op basis van deze onverwachte uitkomst is vervolgens de community van Stackoverflow ingeschakeld (Java8 Lambda performance vs public functions., 2014), wat is gedaan om deze bevinding te delen en/of eventueel bevestiging en uitleg te krijgen. Hierbij heeft de community een kleine wijziging aangeraden aan de code zodat deze lijsten toenemend in grootte worden getest en in verkleinende stappen.

Na het toepassen van deze verbetering (zie Bijlage B), kan er aan de hand van de uitvoer (zie Bijlage C) worden geconcludeerd dat lambda niet efficiënt werkt tegenover publieke functies wanneer je een kleinere lijst of dezelfde lijst vaker filteren wilt. Een mogelijke verklaring zou kunnen zijn dat `Collectors.toList()` een zware operatie is. Uit onderzoek is echter gebleken dat de `toList()` functie zelf een $O(n)$ notatie heeft, wat geen zware operatie betreft (Is there a performance impact when calling `ToList()`?, 2014). Het is dan ook niet duidelijk wat deze inefficiëntie veroorzaakt.

Wanneer het een grote lijst en geen of weinig rotaties betreft komt lambda echter het meest efficiënt uit de test met een ruime voorsprong.

Hieruit kan worden geconcludeerd dat de ouderwetse manier om een publieke functie aan te roepen, efficiënter is wanneer het zal worden gebruikt om een kortere lijst te filteren. De efficiëntie van lambda is daarnaast vele malen groter wanneer het een grote lijst betreft.

4.3 Onderhoud & testbaarheid

Wat zijn de voor- en/of nadelen van het onderhoud van lambda?

Het gebruik van lambda brengt niet alleen voordelen met zich mee, maar ook nadelen. Zo zijn er een aantal aspecten dat het gebruik van lambda binnen Java 8 lastiger kan maken (Lambdas have come to Java!, 2014). De meeste van deze problemen zijn echter goed op te lossen met behulp van method references.

Niet herbruikbaar

Zoals beschreven in het hoofdstuk “Wat is lambda eigenlijk?” is lambda een anonieme functie die gebruik maakt van een functional interface dat een enkele functie bevat. Omdat lambda een anonieme, interne functie is, is deze functie niet aan te roepen. Dit zorgt ervoor dat de lambda functie niet herbruikbaar is en gelimiteerd tot een enkele scope.

Niet testbaar in isolatie

Omdat je een lambda functie niet kunt aanroepen, is het niet mogelijk om de functionaliteit van de lambda apart te gaan testen. Hiervoor zouden extra tests moeten worden geschreven voor de bovenliggende functie om de functionaliteit van de lambda goed te testen.

Geen code beschrijving

Lambda heeft geen enkele vorm van codebeschrijvingen. Hierbij kun je denken aan een functienaam die aangeeft wat de functionaliteit is van dat stuk code. Door de afwezigheid hiervan moeten ontwikkelaars de gehele lambda code goed doorlezen om te proberen te begrijpen wat de lambda als functionaliteit heeft. Naarmate de lambda in complexiteit toeneemt, is dit onbegonnen werk en zal het steeds meer ontwikkeltijd in beslag nemen om de functionaliteit naar boven te halen.

Beroerde stack trace

Wanneer er een Exception in de lambda expressie voorkomt, is de resulterende stack trace van enorm slechte kwaliteit. Zie onderstaande afbeelding voor een voorbeeld van een stack trace in het geval van een DivideByZeroException.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.lambda$1(Main.java:17)
    at Main$$Lambda$2/1159190947.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(Unknown Source)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Unknown Source)
    at java.util.stream.AbstractPipeline.copyInto(Unknown Source)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(Unknown Source)
    at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(Unknown Source)
    at java.util.stream.AbstractPipeline.evaluate(Unknown Source)
    at java.util.stream.ReferencePipeline.collect(Unknown Source)
    at Main.<init>(Main.java:17)
    at Main.main(Main.java:10)
```

Afbeelding 4.3.1 - Stack trace wanneer een exception zich voordoet in een lambda expressie

Dit wordt vooral veroorzaakt doordat je niet de gehele informatie van de stack krijgt maar slechts een fractie ervan, wat tevens enorm onduidelijk is. Zo krijg je bijvoorbeeld te zien welke Exception er gebeurt, maar niet in welke functie deze zich bevindt (zie regel 2) (The Dark Side Of Lambda Expressions in Java 8., 2014). Afhankelijk van de compiler kan dit echter worden verbeterd door *Named Functions*, *Method Reference*: “Wat zijn Method References?” of *Method Lifting*: “Wat is method lifting?” toe te passen.

Wat zijn Named Functions?

Named functies zijn in principe gewoon functie declaraties die een inputtype en een returntype bevatten met daarbij de handelingen die het zou moeten verrichten. Wanneer je een dergelijke named function vervolgens in een lambda implementeert, krijg je het volgende:

```

private List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

public void functionExample() {
    final Function<Integer, Integer> addOneToValue = number -> number + 1;
    final List<Integer> numbersPlusOne =
        numbers.stream().map(addOneToValue).collect(Collectors.toList());

    print(numbers);
    print(numbersPlusOne);
}

```

Afbeelding 4.3.2 - Named functions in combinatie met lambda

Stacktrace resultaat

In vergelijking met de eerder getoonde stack trace, kun je met implementatie van Method Reference zien dat de onderstaande stack trace nu meer duidelijkheid schept.

```

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.addOneToValue(Main.java:28)
    at Main$$Lambda$7/821270929.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(Unknown Source)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Unknown Source)
    at java.util.stream.AbstractPipeline.copyInto(Unknown Source)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(Unknown Source)
    at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(Unknown Source)
    at java.util.stream.AbstractPipeline.evaluate(Unknown Source)
    at java.util.stream.ReferencePipeline.collect(Unknown Source)
    at Main.methodLiftingExample(Main.java:32)
    at Main.<init>(Main.java:17)
    at Main.main(Main.java:10)

```

Afbeelding 4.3.3 - Verbeterde stack trace met behulp van method lifting

In welke situaties komt lambda wel goed van pas?

Veel van de nadelen zijn op te lossen door het toepassen van method references, zie “*Wat zijn Method References?*” in paragraaf 5.1. Echter zijn niet alle nadelen altijd even goed op te lossen. Om de behandelde redenen is het dan ook niet handig om lambda toe te passen wanneer het te complex begint te worden.

Een voorbeeld van lambda toegepast op code van te hoge complexiteit, gemaakt in de taal Scala (Lambdas have come to Java!, 2014):

```

val next = x.map {
  case Success(k) => {
    deriveValueAsynchronously(worker(initValue))(pec).map {
      case None => {
        val remainingWork = k(Input.EOF)
        success(remainingWork)
        None
      }
      case Some(read) => {
        val nextWork = k(Input.El(read))
        Some(nextWork)
      }
    }(dec)
  }
  case _ => { success(it); Future.successful(None) }
}(dec)

```

Afbeelding 4.3.4 - Complex lambda voorbeeld in Scala

Lambda is goed toe te passen op momenten dat je bijvoorbeeld met delegates, listeners, callback handlers (Yet Another Lambda Tutorial., 2014) of diverse kleinere taken gaat werken. Onder kleinere taken kan bijvoorbeeld het filteren van een lijst worden verstaan. In de afbeeldingen 4.3.5 en 4.3.6 worden voorbeelden getoond waarin lambdas onder andere goed van pas komen.

Het eerste voorbeeld betreft het implementeren van een listener:

```

// Without Lambda
JButton btnNewButton = new JButton("New button");
btnNewButton.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent arg0) {
        btnNewButton.setText("[Default]You Clicked Me!");
    }

});
contentPane.add(btnNewButton, BorderLayout.WEST);

// With Lambda
JButton btnNewButton_1 = new JButton("New button");
btnNewButton_1.addActionListener(event -> btnNewButton_1.setText("[Lambda]You Clicked Me!"));
contentPane.add(btnNewButton_1, BorderLayout.EAST);

```

Afbeelding 4.3.5 - Handige toepassingen van lambda op listeners

Het tweede voorbeeld betreft het implementeren en starten van een thread die een loop uitvoert.

```

public Main() {
    // Threads without Lambda
    new Thread(new Runnable() {

        @Override
        public void run() {
            forLoop("Without Lambda");
        }

    }).run();

    // Threads with Lambda
    new Thread( () -> forLoop("With Lambda")).run();
}

public void forLoop(String text) {
    System.out.println(text);
    for(int i = 0; i < 10; i++)
        System.out.println(i);
    System.out.println();
}

```

Afbeelding 4.3.6 - Handige toepassingen van lambda op threads

Unit testen met lambda

Zoals eerder genoemd, is lambda niet te testen in een aparte scope omdat lambda een anonieme functie is. Hierdoor zul je dus je tests moeten uitbreiden door de bovenliggende methode te testen met een specifieke test voor de functionaliteit van lambda. Dit kan opgelost worden door gebruik te maken van method reference.

Uiteraard kun je wel tests uitvoeren met behulp van Lambda (JUnit: Testing Exception With Java 8 And Lambda Expressions., 2014). Met behulp van lambda kun je hierbij gemakkelijk een methode aanroepen waarbij je diverse 'assertions' of vergelijkingen kunt uitvoeren om een waarde te testen.

Hieronder een voorbeeld van een unit test die gebruik maakt van lambda:

```

@Test
public void exercise_1_personWithCollectorToString() {
    Person p1 = new Person("Anna", 32, false);
    Person p2 = new Person("Peter", 21, true);
    List<Person> persons = Arrays.asList(p1, p2);
    assertThat(persons.stream().collect(personToString()), equalTo("(Anna -> V),(Peter -> M)"));
}

```

Afbeelding 4.3.7 - Unit testen met behulp van lambda

Variabelen in lambda

Bij het veranderen van lambda scope variabelen is er een kans dat de compiler een foutmelding geeft op final en effectively final. Voor lambda is het vereist dat een variabele een final of effectively final is (Local Classes, 2014). Dit komt door het feit dat lambda geen normaal gedeclareerde variabele kan aanroepen, wat wordt veroorzaakt door de lambda scope. Als een variabele namelijk niet final is, houdt het in dat de waarde zou kunnen veranderen terwijl deze

op dat moment in gebruik kan zijn voor berekeningen. Dit zal lijden tot onverwachte mutaties in de berekening.

Het verschil tussen final en effectively final, ligt zowel in de declaratie en initialisatie van een variabele, en hoe deze wordt gebruikt in de code (Difference between final and effectively final, 2014).

Een variabele wordt bijvoorbeeld gedeclareerd als een final wanneer de variabele op een later moment in de code niet meer gewijzigd mag worden.

Een effectively final variabele wordt daarentegen niet als final gedeclareerd. Maar wordt als elke andere variabele gedeclareerd, bijvoorbeeld als een private int. Het 'effectively final' concept ontstaat wanneer de variabele nergens in de code wordt aangepast, en dus dezelfde waarde behoudt als dat deze had tijdens de initialisatie.

4.4 Syntax & Leesbaarheid

Op welke manieren kan de duidelijkheid van een functie bewaard worden, ondanks het onbreken van de functienaam?

Lambda is een snelle manier om een stuk code uit te voeren, maar mist wel een functienaam. Hierdoor is niet binnen één oogopslag duidelijk wat een functie precies doet en dit kan de leesbaarheid benadelen.

Dit kan natuurlijk vooral opgelost worden door het gebruik van duidelijk commentaar. Normaal gesproken wil je voorkomen dat er commentaar in je code staat, omdat het inhoudt dat je geen method transparency hanteert (Transparency (behaviour), 2014). Method transparency houdt in dat je aan de naam van de methode kunt zien welke functionaliteit hij op zich zal nemen.

Wanneer er besloten is om alsnog commentaar aan een functie toe te voegen, bestaat deze vaak uit meerdere regels. Omdat Lambda een interne functie is, zal er in dat geval dus bovenaan de lambda functie in één regel commentaar kunnen worden geplaatst om de functie toe te lichten.

Naast commentaar zal er echter wel gebruik kunnen worden gemaakt van Method Reference om het gedeeltelijk duidelijk te maken.

Wat is het effect van method references op de leesbaarheid?

In tegenstelling tot anonieme lambdas kunnen method references ook meegegeven worden als parameter. Dit heeft voor de leesbaarheid als voordeel dat logica een naam gegeven kan worden. Als een lambda bijzonder uitgebreid is, dan kan de leesbaarheid verbeterd worden door hier een method reference van te maken. Dit omdat een benaming van logica in het algemeen korter(en leesbaarder is) is dan de uitwerking. Deze uitspraak is gebaseerd op de persoonlijke mening van het onderzoeksteam verantwoordelijk voor dit document.

4.5 Overig

Welke aanpassingen zijn er nodig geweest voor de ondersteuning van lambda's?

Om lambda's te implementeren zijn er diverse wijzigingen doorgevoerd aan zowel de taal als de compiler en Java Virtual Machine. Zo zullen er bijvoorbeeld wijzigingen nodig zijn in de reflective API's waaronder `java.lang.reflect` en `javax.lang.model` (JSR 335: Lambda Expressions for the Java™ Programming Language, 2014).

Lambda is vooral ontworpen om het 'vertical problem' op te lossen. Het 'vertical problem' houdt in dat er meerdere regels code geschreven moeten worden voor een simpele statement. Zo hebben interfaces en abstracte klassen zoals `Runnable`, `Callable`, `EventHandler` of een `Comparator` een enkele abstracte methode. Om deze interfaces en- of abstracte klassen te gebruiken, zou er in Java 7 en oudere versies rond de vijf regels code moeten worden geschreven voor een enkele statement (State of Lambda, 2014).

Om lambda vervolgens te compilen naar bytecode, is Oracle tot de oplossing gekomen om met `invokeDynamic` te gaan werken. Zoals vermeld in de presentatie van Brian Goetz (2011), "We can use `invokeDynamic` to embed a recipe for constructing a lambda at the capture site" (Goetz, 2014).

Zodra er in de compiler een lambda wordt gedetecteerd, zal de compiler `invokeDynamic` aanroepen om een SAM (Single Abstract Method) (Introduction to Functional Interfaces - A concept recreated in Java 8, 2014) te genereren, ook wel gerefereerd naar als de Lambda Factory. Met behulp van deze generatie wordt de opgevangen lambda expressie vertaald naar een static methode. Gegeven de volgende lambda:

```
s -> s.length() == 10
```

Zal worden vertaald naar (Goetz, 2014):

```
static boolean lambda$1(String s) {
    return s.length() == 10;
}
```

De JVM (interpreter) zal echter geen lambda instructie ontvangen, maar in plaats daarvan zal hij het volgende lezen:

```
invokedynamic #0:apply:()Ljava/util/function/Function;
```

Dit komt door de manier waarop `invokeDynamic` is ontworpen. De echte lambda instructie zal namelijk worden opgeslagen in een 'entry' in een aparte tabel waar de lambda instructie de #0 parameter toegewezen heeft gekregen (Weiss, 2014)

Hoe is lambda te vergelijken met andere programmeertalen?

Het overgrote deel van de programmeertalen ondersteunt de zogenaamde 'anonymous functions'. Elke taal heeft hier overigens wel een andere syntax voor. De lambda expressies in Java 8 hebben de meeste overeenkomsten met de lambda expressies in C++. Een lambda expressie in C++ ziet er als volgt uit:

```
[capture] (parameters) -> return_type { function_body }
```

De lambda expressies uit Java 8 zijn eigenlijk een versimpelde variant hierop, namelijk:

```
(parameters) -> { function_body }
```

In veel van de gevallen is het, zoals de naam ‘anonymous function’ doet verwachten, simpelweg een functie zonder naam. In JavaScript is dit bijvoorbeeld het geval;

```
function (parameters) { function_body }
```

Het grootste verschil is dan ook dat het daadwerkelijke woord ‘function’ erin voorkomt. Wat de leesbaarheid betreft is dat natuurlijk een groot voordeel. In de lambda expressies van Java 8 is dit woord weggelaten, wat ervoor zorgt dat de functies lastiger te onderscheiden zijn.

Hieruit kunnen we concluderen dat lambda expressies in Java 8 redelijk wat overeenkomsten hebben met andere talen, wat maakt dat ze gemakkelijk te begrijpen zijn. Wat de leesbaarheid betreft is het echter minder duidelijk leesbaar dan bijvoorbeeld JavaScript of PHP.

4.6 Conclusie

Lambda neemt behoorlijk veel voor- en nadelen met zich mee. Zo is het dankzij de introductie van lambda expressies niet langer nodig om uitgebreide code te schrijven om uiteindelijk een enkele abstracte methode te implementeren. Dit kan nu worden opgelost met een enkele regel code.

Daarnaast heeft lambda een aantal grote nadelen, waaronder bijvoorbeeld de beperking van testbaarheid of het kunnen hergebruiken van een lambda. Oracle heeft hier goed over nagedacht en hebben zij ervoor gezorgd dat deze nadelen grotendeels weggenomen kunnen worden door gebruik te maken van Method References.

5 Streams

5.1 Algemeen

Java heeft met versie 8 een nieuwe API geïntroduceerd om streams mee aan te maken en operaties over uit te voeren. In de documentatie van Oracle (2014) wordt de streams API als volgt beschreven: “Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections”. Hierbij wordt het volgende voorbeeld gegeven:

```
int sum = widgets.stream()  
    .filter(b -> b.getColor() == RED)  
    .mapToInt(b -> b.getWeight())  
    .sum();
```

Afbeelding 5.1.1 - Voorbeeld van een Stream

In dit voorbeeld is te zien hoe van de collectie widgets een stream wordt gemaakt met behulp van de functie stream(), vervolgens wordt de stream gefilterd en wordt het gewicht van de gefilterde elementen opgevraagd met behulp van lambda expressies. Tot slot wordt de som van alle resultaten berekend.

Streams zijn te onderscheiden in twee verschillende soorten: serieel en parallel. Seriële streams voeren, zoals de naam al suggereert, hun operaties achtereenvolgens uit. Ze werken een operatie van de pipeline uit voor alle elementen om vervolgens naar de volgende operatie te gaan. Parallele streams voeren de gehele pipeline in één keer uit voor losse elementen.

Waarin onderscheiden streams zich van collecties?

Streams lijken in eerste instantie veel weg te hebben van collecties, maar ze verschillen zeer op een aantal punten (Oracle, 2014). Ten eerste worden elementen in een stream verwerkt in een aaneensluiting van berekeningen en hebben deze geen opslag nodig. Ook passen streams hun bron niet aan, maar worden er na elke uitgevoerde functie nieuwe streams aangemaakt in plaats van bijvoorbeeld het verwijderen van elementen uit de bron van de collectie. Een andere eigenschap van streams is dat het de weg van de minste weerstand neemt. Als een bepaalde functie het toestaat zal de simpelste oplossing worden gekozen om een operatie te voltooien, zo kan bijvoorbeeld bij een worden vroegtijdig worden beëindigd zodra de voorwaarden van het filter zijn behaald. Alle operaties, behalve de terminale (laatste operatie), hanteren deze manier van werken. Streams kunnen in theorie ook oneindig zijn. Door het toepassen van bepaalde functies over streams (bijvoorbeeld limit(n), findFirst()) kunnen oneindige streams worden verwerkt in een eindige tijd. Tenslotte zijn streams consumeerbaar. Een element in een stream kan slechts een maal bezocht worden in de tijd dat de stream bestaat, mocht een element nogmaals geëvalueerd moeten worden, zal er een nieuwe stream van de bron moeten worden gemaakt.

5.2 Performance

Welke sorteeralgoritmes gebruiken streams?

Doordat Java 8 nog relatief nieuw is, is de hoeveelheid informatie over wat diepgaandere vragen nog zeer beperkt. Ook in het geval van deze vraag was er niet een snel en simpel antwoord te vinden, dus is er een klein experiment opgezet om te onderzoeken waar de sorteeralgoritmes zich bevinden en wanneer welke wordt toegepast. Simpelweg een stream aanmaken en de sorteerfunctie er over uit te voeren leidde niet direct tot de locatie van de implementatie van de algoritmes. Om dit toch op te lossen is een simpele oplossing bedacht: expres een fout in de code invoegen zodat er een exceptie in het sorteren optreedt die een stacktrace naar de locatie van het algoritme terug geeft zoals te zien in onderstaande afbeelding.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
    at java.lang.String.compareTo(Unknown Source)
    at java.util.Comparators$NaturalOrderComparator.compare(Unknown Source)
    at java.util.Comparators$NaturalOrderComparator.compare(Unknown Source)
    at java.util.TimSort.binarySort(Unknown Source)
    at java.util.TimSort.sort(Unknown Source)
    at java.util.Arrays.sort(Unknown Source)
    at java.util.stream.SortedOps$SizedRefSortingSink.end(Unknown Source)
    at java.util.stream.AbstractPipeline.copyInto(Unknown Source)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(Unknown Source)
    at java.util.stream.ForEachOps$ForEachOp.evaluateSequential(Unknown Source)
    at java.util.stream.ForEachOps$ForEachOp$OfRef.evaluateSequential(Unknown Source)
    at java.util.stream.AbstractPipeline.evaluate(Unknown Source)
    at java.util.stream.ReferencePipeline.forEach(Unknown Source)
    at test.main(test.java:13)
```

Afbeelding 5.2.1 - Stacktrace na fout in sorteren

Uit de stacktrace blijkt dat het sorteeralgoritme dat wordt gebruikt in streams de standaard sorteerfunctie is van de Arrays klasse. Voor de sequentiële streams geldt in het geval van primitieven dat gebruik wordt gemaakt van een Dual-Pivot Quicksort en bij het sorteren van een lijst van objecten wordt gebruik gemaakt van Timsort. Alhoewel Timsort gemiddeld een slechtere performance heeft dan Dual-Pivot Quicksort is het gebruik ervan essentieel omdat bij het sorteren van objecten stabiliteit moet worden gegarandeerd (Wikipedia, 2014). In beide gevallen is de keuze voor het sorteer algoritme bepaald doordat deze in gemiddelde gevallen de beste resultaten leveren.

Hoe stabiel zijn streams?

Om betrouwbare resultaten te behalen moeten streams een bepaalde vorm van stabiliteit hebben. Om dit te kunnen garanderen moet storing in de bron van de data voorkomen worden. In de meeste gevallen komt dit erop neer dat de bron van een stream in zijn geheel niet wordt aangepast tijdens het uitvoeren van de stream pipeline. Uitzonderingen hierop zijn bronnen van streams met een spliterator met de CONCURRENT eigenschap (Oracle, 2014). Deze spliterators zijn zodanig ingesteld dat deze om kunnen gaan met veranderingen in de bron van een stream.

Een andere wijze van handelen die is toegepast om streams stabiliteit te geven is ervoor te zorgen dat streams pas geconsumeerd worden op het moment dat de terminale operatie wordt uitgevoerd.

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));
Stream<String> sl = l.stream();
l.add("three");
String s = sl.collect(joining(" "));
```

Afbeelding 5.2.2 - Aanpassing stream na aanmaken

In de bovenstaande afbeelding is te zien hoe een arraylist wordt geïnitieerd met de twee strings “one” en “two”. Vervolgens wordt een stream aangemaakt van de arraylist daarna wordt er nog een derde waarde, “three”, aan de bron toegevoegd. Als deze code wordt uitgevoerd, wordt de stream geconsumeerd op het moment dat de terminale operatie collect wordt uitgevoerd. Dit heeft als gevolg dat de string s uit het voorbeeld zal bestaan uit alle 3 de elementen die in de arraylist zitten. Alle collections en de meeste classes in de JDK werken op deze manier.

Een probleem wat komt kijken bij uitvoering van stream pipelines is dat de resultaten kunnen verschillen elke keer dat deze wordt uitgevoerd of dat de resultaten niet kloppen als er een state afhankelijke lambda wordt gebruikt. Neem het volgende voorbeeld:

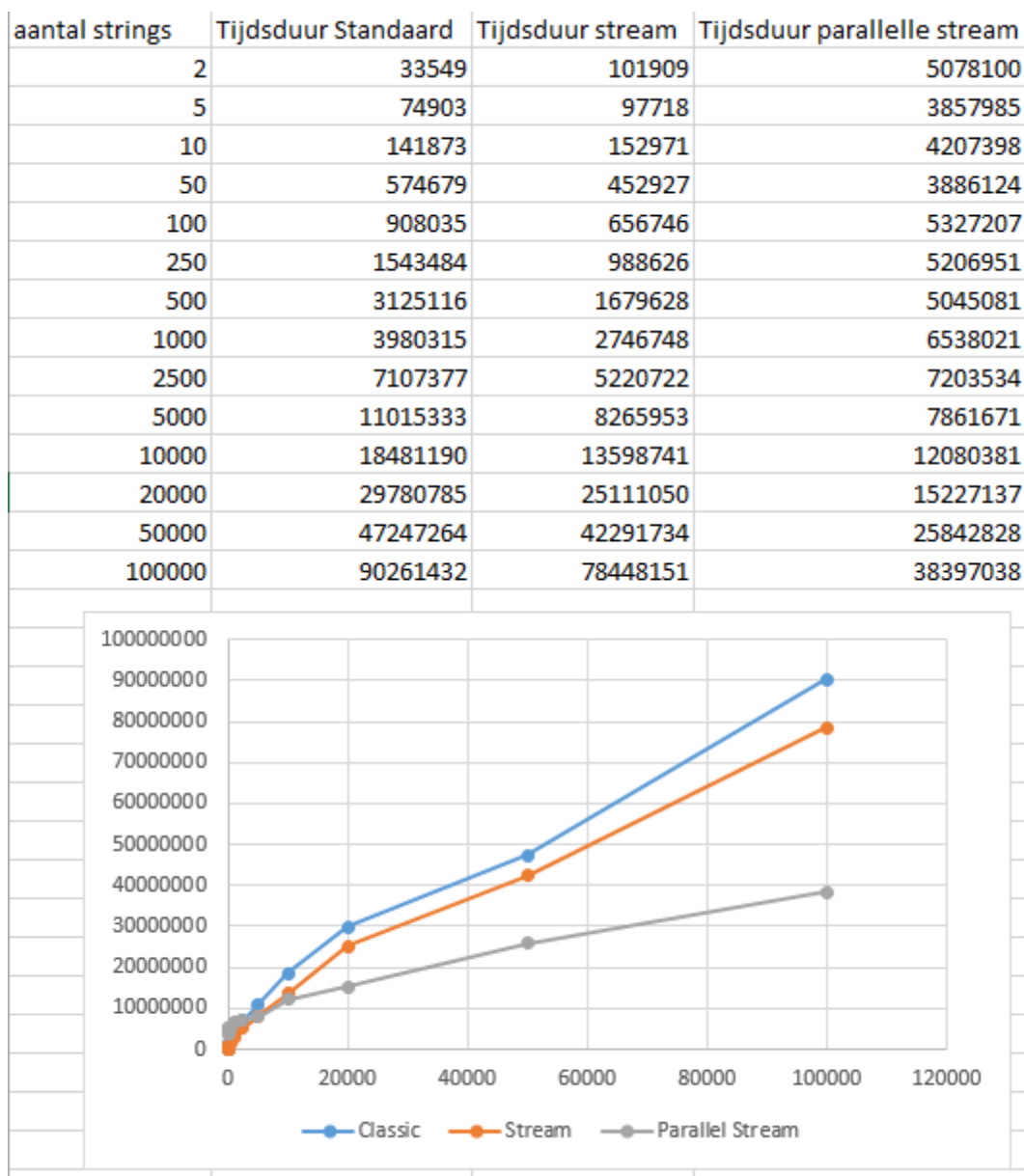
```
Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());
stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })...
```

Afbeelding 5.2.3 - Stream met ongedefinieerd gedrag

Aangezien de map operatie parallel wordt uitgevoerd kunnen de resultaten met elke uitvoering anders zijn doordat threads anders ingepland kunnen worden. Als er in het bovengenoemde voorbeeld gebruik gemaakt zou zijn van een lambda die niet van de state afhankelijk is, zouden de resultaten altijd hetzelfde zijn.

Wat is het verschil in performance tussen parallel streams en serial streams?

Om te onderzoeken in hoeverre parallel streams sneller zijn dan de standaard streams is er een microbenchmark opgezet. Hierbij wordt een simpele bewerking op drie verschillende manieren (d.m.v. for-each loop, stream en parallel stream) uitgevoerd over een variabel aantal strings in een array. De resultaten van de benchmark staan in afbeelding 5.3.3



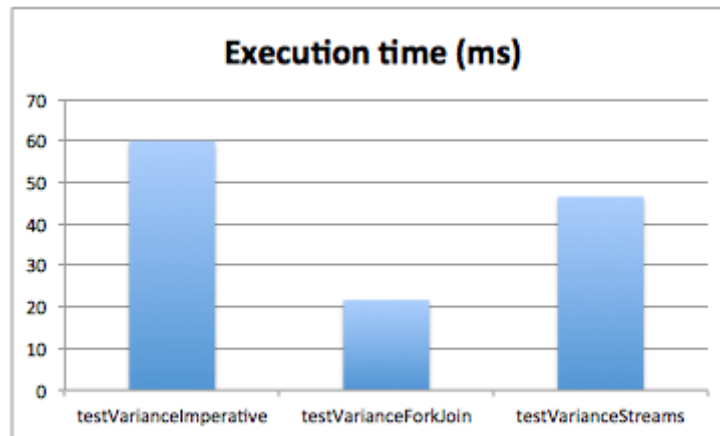
Afbeelding 5.2.4 - Resultaten benchmark streams

Uit de benchmark blijkt dat bij zeer kleine collecties zowel stream als parallel stream minder efficiënt zijn dan de traditionele for-each loop. Bij collecties met een grootte van ongeveer 10 tot ongeveer 5000 entries blijkt de normale stream het beste te presteren. Pas na een collectie met een grootte van meer dan 5000 begint de parallelle stream het beste te presteren.

Verder suggereren de resultaten dat de tijdsduur min of meer lineair loopt wat inhoudt dat het gebruik van parallelle streams, naar mate het aantal entries in de collectie groeit, steeds efficiënter wordt ten opzichte van de twee geteste alternatieven.

Wat is het verschil in performance tussen de Java 7 gebruikelijke join forks en/of threads tegenover parallel streams?

Er is door anderen een onderzoek gedaan naar de performance-verschillen tussen join forks en parallel streams (Comparison fork/join to streams, 2014). Hierbij is hetzelfde probleem op imperatieve wijze opgelost, zowel als met join forks en parallel streams. Wat hierbij opviel was dat de join forks veel meer code nodig had om hetzelfde te bereiken, maar dat deze gemiddeld ongeveer tweemaal zo snel als parallel streams waren.



Afbeelding 5.2.5 - Verschillen in uitvoeringstijd streams en fork/join

Om te verifiëren of deze testresultaten consistent waren, is deze ook lokaal uitgevoerd. Op een computer met een i7 quad-core zijn vergelijkbare resultaten weergegeven. Echter, op een oudere 2GHz dual-core laptop komt parallel streams ongeveer 10x zo langzaam als ForkJoin's (en ongeveer 5x zo langzaam als imperative). Zo is te zien dat parallel streaming in sommige condities enorm traag kan zijn. Dit heeft te maken met een combinatie van de complexiteit van het probleem en de hoeveelheid data. Overigens is parallellisme duidelijk effectiever met meerdere cores.

In conclusie, gebruik parallel-streaming niet blind. Kijk eerst of het een effectieve oplossing is in de context van het probleem, voor deze uit te voeren.

5.3 Onderhoud & Testbaarheid

De testbaarheid van streams is gerelateerd tot de testbaarheid en onderhoudbaarheid van lambda's. Dit is behandeld in het hoofdstuk betreft lambda's. De streams zelf zijn met unittests af te vangen.

5.4 Syntax & Leesbaarheid

De streaming functionaliteit vervangt of biedt een alternatief tot de traditionele fork/joins. Er is veel minder code nodig voor een implementatie met streams, waardoor de resulterende code ook beter te volgen is. Voor gebruikers die bekend zijn met lambda's zal het overzichtelijk zijn. De stream volgt een logische volgorde: de stream wordt aangemaakt, vervolgens wordt gedefinieerd wat de stream allemaal moet uitvoeren, tot slot wordt de stream uitgevoerd. Deze vaste volgorde wordt gehandhaafd, wat de algemene leesbaarheid bevordert.

Toch zal de programmeur bij het (leren) gebruiken van streams fouten maken. De volgorde die gehandhaafd wordt heeft als nadeel dat, als deze niet correct toegepast wordt, de gebruiker per ongelijk een oneindige stream kan maken, of errors zal krijgen. Deze leercurve is echter niet groot en zou geen probleem moeten vormen. Het biedt nog steeds een beter leesbaar resultaat dan fork/joins.

5.5 Conclusie

Streams kunnen handig zijn, maar hebben een paar regels die gehanteerd moeten worden. De code is korter ten opzichte van voorgaande alternatieven, wat de leesbaarheid enorm bevordert, maar de performance is niet altijd beter. Bij het gebruik van streams is het dus belangrijk om te kijken of de performance acceptabel is voor de machine waar deze op wordt toegepast.

6 Functional Interfaces

6.1 Algemeen

Interfaces

Java interfaces (Oracle, 2014) zijn een verzameling van methoden die kunnen worden geïmplementeerd door objecten. Hierbij worden de objecten gedwongen om de functionaliteit van de interfaces te overriden. Verschillende objecten bevatten dezelfde methoden en kunnen op dezelfde manier aan geroepen worden.

6.2 Functional interfaces.

Een functional interface (Oracle, sd) (Introduction to Functional Interfaces - A concept recreated in Java 8, 2014) is een interface die één abstracte methode bevat die uitgevoerd kan worden. Deze eigenschap maakt de functional interface een SAM (**S**ingle **A**bstract **M**ethod) interface. Voorbeelden van SAM interfaces in de Java 7 API zijn: Runnable, Callable, Comparator en ActionListener.

Het doel van een functional interface is om een methode aan te maken die telkens andere functionaliteit kan bevatten en toch kan worden uitgevoerd. De interfaces kunnen dienen als anonieme functies, oftewel functies die uitgevoerd worden maar niet gedeclareerd zijn zoals de normale functies in classes.

Een functional interface wordt gedeclareerd als een normale interface met een optionele “@FunctionalInterface” annotatie (Oracle, sd). Alhoewel deze annotatie optioneel is, wordt het als een good practice gezien deze toch toe te voegen.

```
@FunctionalInterface
public interface IExampleInterface {
    public void execute();
}
```

Afbeelding 6.2.1 - Voorbeeld van een functional interface.

De functional interface kan worden geïnstantieerd met een lambda (What's new in Java 8 lambda's, 2014) expressie, dit is alleen mogelijk vanwege het feit dat de functional interface maar één uitvoerbare methode bevat.

```
IExampleInterface doOneThing = () -> System.out.println("Hello");
IExampleInterface doMoreThings = () ->
{
    System.out.println("Hello");
    System.out.println("World");
};
doOneThing.execute(); // Hello
doMoreThings.execute(); // Hello World
```

Afbeelding 6.2.2 - Voorbeeld van het aanmaken van een instantie van een functional interface.

```

public void executeInterface(IExampleInterface e) {
    e.execute();
}
public void doExample() {
    executeInterface(() -> System.out.println("Hello"));
    executeInterface(() -> {
        System.out.println("Hello");
        System.out.println("World");
    });
}

```

Afbeelding 6.2.3 - Voorbeeld van een functional interface gebruikt als parameter (Introduction to Functional Interfaces - A concept recreated in Java 8, 2014).

Wat is het verschil ten opzichte van het instantiëren van een standaard Java interface?

Het verschil zit in de syntax, Java 8 heeft minder code nodig om een interface te declareren dankzij lambda expressies. De functional interface hoeft niet geïnstantieerd te worden door middel van een “new” cast. De functie die de functionaliteit gaat bevatten hoeft niet gedeclareerd en overriden te worden.

```

IExampleInterface java7 = new IExampleInterface() {
    @Override
    public void execute() {
        System.out.println("Hello World");
    }
};
IExampleInterface java8 = () -> System.out.println("Hello World");

```

Afbeelding 6.2.4 - Voorbeeld van een Java 7 en Java 8 instantie.

Wat is het verschil ten opzichte van de standaard Java interface functionaliteit?

Een normale interface kan meerdere methoden bevatten, functional interfaces zijn gelimiteerd tot één abstracte methode. Dit komt omdat de lambda expressie de abstracte functie declareert, dus is het onmogelijk om er meerdere hebben.

```

public interface INormalInterface {
    public void execute();
    public void work();
}

@FunctionalInterface
public interface IFunctionalInterface {
    public void execute();
    public void work();
}

```

Afbeelding 6.2.5 - Voorbeeld van een compiler error vanwege de 2de methode in de functional interface.

Ondanks dat de functional interface maar één abstracte methode mag bevatten, is het wel mogelijk om static of default methoden te declareren.

```
@FunctionalInterface
public interface IFunctionalInterface {
    public void execute();
    default String getName() {
        return "Functional Interface";
    }
    static void print() {
        System.out.println("Functional Interface Example");
    }
}
```

Afbeelding 6.2.6 - Static en default methoden met functional interface.

Hoe kan men parameters meegeven aan een functional interface?

In veel gevallen wil men parameters (Java 8 Functional Interfaces, 2014) meegeven aan een anonieme functie. Dit kan gerealiseerd worden door een type mee te geven aan de functional interface.

```
@FunctionalInterface
public interface IParameterInterface<T> {
    public void execute(T t);
}
```

Afbeelding 6.2.7 - Functional interface met type.

```
IParameterInterface<String> iString = (String s) -> System.out.println(s);
IParameterInterface<Integer> iInteger = (i) -> System.out.println(i);
iString.execute("Hello world"); //Hello world
iInteger.execute(10);           //10
```

Afbeelding 6.2.8 - Uitvoering Functional interface met type.

In het bovenstaande voorbeeld wordt er een keer met en een keer zonder typedeclaratie een lambda uitgevoerd. Dit is optioneel als het type wordt aangegeven bij het instantiëren. Als er geen type wordt aangegeven bij het instantiëren of bij de lambda parameter, dan zal de compiler een warning geven.

Hoe worden waardes buiten de functional interface gebruikt in een functional interface?

Vaak moeten waardes binnen de applicatie veranderd worden binnen een anonieme functie. Voor de primitieve en simpele data types geldt dat deze als final of effectively final gedeclareerd moeten zijn. Dit betekent dat de waardes van buiten de scope van de functional interface niet aanpasbaar zijn binnen de scope van de interface.

```
int i = 0;
IExampleInterface iExample= () -> {
    i = 5;
};
```

Afbeelding 6.2.8 - Aanpassing van een simpel data type binnen de scope.

Het is echter wel mogelijk om ingewikkelde data types aan te passen mits deze niet opnieuw worden gedeclareerd. De inhoud van een object of array kan worden aangepast omdat ze niet lokaal worden aangemaakt.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
String[] letters = new String[5];
IExampleInterface iExample = () -> {
    numbers.add(1);
    letters[0] = "a";
};
numbers.size(); //0
System.out.println(letters[0]); //null
iExample.execute();
numbers.size(); //1
System.out.println(letters[0]); //a
```

Afbeelding 6.2.9 - Aanpassing van arrays binnen de scope.

Hoe werken de functional interfaces die zijn toegevoegd aan de Java 8 API?

De nieuwe API van Java 8 bevat een aantal nieuwe functional interfaces. Twee van de nieuwe functies zijn predicate en consumer.

Predicate

De predicate interface (How to use predicate in Java 8, 2014) (Interface Predicate, 2014) bevat een conditie die men kan testen of als parameter aan een filter kan meegeven. Het is mogelijk door middel van chaining om verschillende condities tegelijk te testen.

```
Predicate<Integer> higherThanNine = (i) -> i > 9;
higherThanNine.test(5); //false
higherThanNine.test(21); //true
```

Afbeelding 6.2.10 - Predicate voorbeeld.

De filter functie van streams kan een predicate ontvangen. Zo kun je een predicate conditie declareren en meegeven aan een filter van een stream. Op deze manier wordt een loop in combinatie met een if-statement vervangen door de onderstaande syntax.

```
int[] numbers = { 1, 9, 12, 21, 40 };
IntPredicate higherThanTen = (i) -> i > 10;
IntPredicate lowerThanTwenty = (i) -> i < 20;
Arrays.stream(numbers).filter(higherThanTen.and(lowerThanTwenty)); //12
IntPredicate lowerThanEight = (i) -> i < 8;
Arrays.stream(numbers).filter(higherThanTen.or(lowerThanEight)); //1, 12, 21, 40
```

Afbeelding 6.2.11 - Predicate voorbeeld met chaining.

De code om bovenstaande functionaliteit te realiseren is met de nieuwe Java 8 een stuk minder geworden. De leesbaarheid van de code is duidelijk aangezien je in een regel kunt lezen wat er precies uitgevoerd wordt. In het volgende voorbeeld wordt de bovenstaande filtering gerealiseerd in Java7 met de dezelfde leesbaarheid.

```

private void filterNumbers() {
    int[] numbers = { 1, 9, 12, 21, 40 };
    for(int i = 0; i < numbers.length; i++) {
        if(higherThanTen(numbers[i]) && lowerThanTwenty(numbers[i])) {
            System.out.println(numbers[i]);
        }
    }
}
private boolean higherThanTen(int number) {
    return number > 10;
}
private boolean lowerThanTwenty(int number) {
    return number < 20;
}

```

Afbeelding 6.2.12 - Voorbeeld Java 7 leesbaarheid.

```

int[] numbers = { 1, 9, 12, 21, 40 };
IntPredicate higherThanTen = (i) -> i > 10;
IntPredicate lowerThanTwenty = (i) -> i < 20;
Arrays.stream(numbers).filter(higherThanTen.and(lowerThanTwenty)); //12

```

Afbeelding 6.2.13 - Voorbeeld Java 8 leesbaarheid.

Consumer

Een consumer (Java 8 consumer and supplier, 2014) (Interface Consumer, 2014) bevat functionaliteit die uitgevoerd wordt als de accept methode wordt aangeroepen. Verschillende consumers kunnen na elkaar uitgevoerd worden door middel van chaining.

```

Consumer<Integer> print = (i) -> System.out.println(i);
print.accept(5); //5
Consumer<Integer> printAgain = (i) -> System.out.println("Again " + i);
print.andThen(printAgain).accept(5); //5 Again 5

```

Afbeelding 6.2.14 -

Het is mogelijk om deze interfaces te combineren als vervanging van if statements. De predicate bepaalt op welke data de functionaliteit wordt uitgevoerd en de consumer behandelt de uitvoering.

```

private void apply( int number,
                   Predicate<Integer> predicate,
                   Consumer<Integer> consumer) {
    if (predicate.test(number)){
        consumer.accept(number);
    }
}
private void doExample() {
    IntStream numbers = Arrays.stream(new int[]{ 1, 9, 12, 21, 40 });
    Predicate<Integer> higherThanNine = (i) -> i > 9;
    Consumer<Integer> print = (i) -> System.out.println(i);
    numbers.forEach((i) -> apply(i, higherThanNine, print)); //12 21 40
}

```

Afbeelding 6.2.15 - Voorbeeld van de combinatie Predicate en Consumer

```

IntPredicate higherThanNine = (i) -> i > 9;
IntConsumer print = (i) -> System.out.println(i);
IntStream numbers = Arrays.stream(new int[]{ 1, 9, 12, 21, 40 });
numbers.filter(higherThanNine).forEach(print); //12 21 40
/*
numbers.filter((i) -> i > 9).forEach((i) -> System.out.println(i));
*/

```

Afbeelding 6.2.16 - Voorbeeld van een alternatieve kortere manier.

Hoe is de werking van de functional interfaces vergelijkbaar met functionele programmeertalen?

In JavaScript is het heel gewoon om anonieme functies te declareren. Deze worden vaak gebruikt bij asynchrone processen in de vorm van een callback. De nieuwe functional interfaces lijken op deze callbacks. Door het aanmaken van een functional interface, kan functionaliteit doorgegeven worden als parameter en later asynchroon worden uitgevoerd.

```

function asynchCall (callback) {
    setTimeout(function () {
        callback();
    }, 1000);
}
asynchCall(function () {
    alert("Hello world!");
});

```

Afbeelding 6.2.17 - Voorbeeld van een asynchrone aanroep van een anonieme functie in JavaScript.

```

private void asynchCall(IExampleInterface iExample) {
    new Thread(() -> {
        try {
            Thread.sleep(1000);
            iExample.execute();
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }).start();
}
private void doExample() {
    asynchCall(() -> System.out.println("Hello world!"));
}

```

Afbeelding 6.2.18 - Voorbeeld van een functional interface die asynchroon wordt aangeroepen.

De Runnable interface van Java is sinds Java 8 ook een functional interface geworden. Daarom kan bovenstaande voorbeeld ook verkort worden door de lambda direct aan de thread mee te geven.

```

private void doExample() {
    new Thread(() -> {
        try {
            Thread.sleep(1000);
            System.out.println("Hello world!");
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }).start();
}

```

Afbeelding 6.2.19 - Voorbeeld van een functional interface die asynchroon wordt aangeroepen.

Conclusie functional interfaces.

De nieuwe functionaliteit van Java 8 wat betreft functional interfaces zorgt aanzienlijk voor minder benodigde code. De SAM interfaces uit de voorgaande versies zijn compatible gemaakt met de nieuwe lambda notatie en zijn ook verkort wat betreft de syntax. De functionaliteit van de functional interfaces heeft de taal meer in de richting van functional programming gestuurd.

7 Optionals

7.1 Algemeen

Wat is een Optional?

Met de release van Java 8 is er een nieuwe klasse bijgekomen, de Optional. De Optional is samengevat “een wrapper” die een referentie naar een ander object bevat” (Johnson, 2014). Deze klasse is niet alleen bedoeld om NullPointerExceptions te voorkomen, maar ook om mensen te helpen om de code sneller te begrijpen. Zo zou je sneller kunnen zien of je een Optional kan verwachten, en dus rekening moet houden met de mogelijkheid geen valide waarde terug te krijgen. Hierdoor kun je je code beschermen tegen onverwachte NullPointerExceptions (Tired of Null Pointer Exceptions? Try using Java SE 8's Optional!, 2014).

Wanneer wordt een Optional gebruikt?

Optionals worden voornamelijk gebruikt om nullpointers weg te werken, echter wanneer is het wel of niet handig om een Optional te gebruiken? Het rare aan een Optional is dat het nullpointers maar gedeeltelijk wegwerkt, aangezien het nog steeds een nullpointer terug kan geven. Dit komt doordat de optional zelf null kan zijn. Dit betekent dat het juist moet worden geïmplementeerd, wil het daadwerkelijk nullpointer exceptions tegen kunnen gaan.

In principe kunnen Optionals worden gebruikt voor variabelen die leeg (null) kunnen zijn. Op die manier zou een nullpointer tegen gegaan kunnen worden. De implementatie van een Optional zou er als volgt uit kunnen zien (Johnson, 2014):

```
optionalString = Optional.ofNullable(optionalTest.getNullString());  
optionalString.ifPresent( s -> { System.out.println(s.toString()); } );
```

Het is natuurlijk de vraag of dit van toegevoegde waarde is. Hetzelfde kan namelijk gedaan worden met de volgende code:

```
String s = optionalTest.getNullString();  
if ( s != null ) { System.out.println(s.toString()); }
```

In principe is het gebruiken van een Optional als return-waarde van een functie de enige nuttige implementatie van Optionals (Jhades, 2014). Op deze manier wordt het aantal NullPointerExceptions in een applicatie daadwerkelijk verlaagd. Het voordeel hiervan is ook dat er niet stil gestaan hoeft te worden bij een eventuele null waarde als resultaat van een functie.

7.3 Leesbaarheid & Syntax

Hoe wordt een Optional toegepast?

Optional vereist uiteraard andere syntax dan de controle op een eventuele null waarde. Om een goed voorbeeld te geven van de benodigdheden zal er een code demonstratie gegeven worden.

Voor het zoeken naar een specifieke fruitnaam in een Array met meerdere fruitnamen zou de functie er als volgt uit kunnen zien (Java 8 Optional Objects, 2014):

```
// Conventional way
public Fruit findFruitConventional (String name, List<Fruit> fruits) {
    for(Fruit fruit : fruits) {
        if (fruit.fruitName.equals(name)) {
            return fruit;
        }
    }
    return null;
}

// Java 8 using Optional
public Optional<Fruit> findFruitOptional (String name, List<Fruit> fruits) {
    for(Fruit fruit : fruits) {
        if (fruit.fruitName.equals(name)) {
            return Optional.of(fruit);
        }
    }
    return Optional.empty();
}
```

Afbeelding 7.3.1 - Optional tegenover traditionele handelswijze

Deze code zou kunnen worden aangeroepen om een specifieke fruitnaam te vinden. Een dergelijke aanroep zou er dan als volgt uit kunnen zien:

```
// Conventional way
Fruit conFruit = fruit.findFruitConventional("banana", fruits);
if (conFruit != null) { System.out.println(conFruit.toString()); }
else { System.out.println("Unable to find this fruit"); }

// Java 8 using Optional
Optional<Fruit> optFruit = fruit.findFruitOptional("banana", fruits);
if (optFruit.isPresent()) { System.out.println(optFruit.get().toString()); }
else { System.out.println("Unable to find this fruit"); }
```

Afbeelding 7.3.2 - Wijze van aanroepen optional tegenover traditionele handelswijze

Aan de hand van bovenstaand voorbeeld zou je kunnen concluderen om niet voor Optional te kiezen in het geval van een enkele if-statement. Zul je echter een nested-if structure moeten gaan toepassen, dan zou je weer bij een compleet andere structuur uit kunnen komen om de Optional toe te passen (Tired of Null Pointer Exceptions? Try using Java SE 8's Optional!, 2014).

Wanneer je meerdere values moet opvragen die mogelijk null of Optional kunnen zijn, zul je nested if statements of mapping moeten toepassen. Zie onderstaande code voor een voorbeeld van mapping met het gebruik van een Optional (Optional in Java 8 Cheat Sheet, 2014).

```
return person.flatMap(Person::getAddress)
               .flatMap(Address::getValidFrom)
               .filter(x -> x.before(now()))
               .isPresent();
```

Afbeelding 7.3.3. - Mapping met optional

Gegeven de bovenstaande voorbeelden, kan er geconcludeerd worden dat het niet per direct leesbaarder wordt, dat de hoeveelheid code in principe gelijk blijft en dat het vooral van de situatie afhankelijk is of je Optional zou willen toepassen. Daarnaast heeft iedereen zijn of haar eigen smaak en gebruiken qua syntax en kan de mening hierover verschillen.

7.2 Performance

Wat is het performanceverschil tussen Optionals en nested if-statements?

Zoals vermeld, is een Optional een Object dat naar een ander Object verwijst. Doordat elk object zijn eigen geheugenadres heeft, is het gebruik van Optionals dus minder efficiënt wat betreft het gebruik van memory.

Een leeg object neemt bijvoorbeeld 8 bytes in het geheugen in beslag, een object met een enkele boolean neemt daarentegen al snel 16 bytes in beslag (Memory usage of Java objects: general guide, 2014). De Optional klasse neemt dus zelf al een grote hoeveelheid bytes in beslag in het geheugen. Daar komt het benodigde geheugen van je eigen klasse nog bovenop.

Ondanks dat een nested-if statement er niet al te charmant eruit ziet, heeft deze geen last van het toewijzen van extra geheugen zoals dat bij Optional het geval is.

Het verschil in de benodigde hoeveelheid processorkracht met de huidige technologie is echter zo miniem, dat het niet uit maakt met welke methode je zal gaan werken. Er zal dan ook niet dieper worden in gegaan op het verschil in processorkracht voor beide methoden. In het verschil tussen de twee methoden ligt de nadruk dan ook vooral op het geheugengebruik.

7.4 Overige

Waarom is het gebruiken van Optionals beter dan het controleren op null waarden?

Het gebruik van Optionals is niet per direct duidelijker dan het simpelweg afhandelen van nullpointers met behulp van nested-if statements. Het feit is dat het onmogelijk is om een NullPointerException te produceren met een Optional, wanneer deze juist wordt toegepast (Optional in Java 8 Cheat Sheet, 2014).

Een Optional zal namelijk altijd minimaal gevuld moeten worden met Optional.empty() in plaats van null. Een Optional is een soort 'wrapper' voor een object. Dat object kan null zijn, en kan dus een NullPointerException veroorzaken. Door altijd Optional.empty() te gebruiken, en

Optionals dus juist toe te passen, kan het vermijden van een `NullPointerException` gegarandeerd worden.

7.5 Conclusie

Optionals zijn een waardevolle toevoeging aan Java, deze mening kan echter verschillen per persoon. Wat leesbaarheid betreft is het bijvoorbeeld niet direct duidelijker of minder duidelijk, maar meer een persoonlijke voorkeur. Zowel het gebruik van Optionals en nested if-statements heeft namelijk zijn voor- en nadelen.

De voordelen van het daadwerkelijk toepassen van een Optional worden pas duidelijk wanneer deze gebruikt wordt als returnwaarde van een functie. Op die manier hoeft er namelijk geen rekening gehouden te worden met het terugkrijgen van een null-waarde en kan het aantal `NullPointerExceptions` daadwerkelijk verminderd worden. Helaas is juiste implementatie van Optionals wel wat aan de lastige kant, aangezien deze in eerste instantie redelijk lastig te begrijpen zijn en als omslachtig aangezien kunnen worden.

8 Conclusie

Algemeen

Java 8 heeft veel nieuwe functionaliteit gekregen die, mits goed toegepast, de leesbaarheid en productiesnelheid bevorderen. Elke functionaliteit blijkt echter ook zijn nadelen te hebben. Lambda expressies zijn bijvoorbeeld moeilijk te testen en streams winnen niet altijd performance. Het is van belang om deze functionaliteit niet blind toe te passen zonder bekend te zijn met de nadelen die ze mee kunnen brengen, anders zal de winst waarschijnlijk verloren gaan.

Annotations

De meerwaarde van repeatable annotations bestaat uit een kleine verbetering in leesbaarheid en gebruiksgemak. Een gebruiker hoeft niet zoals bij elke toepassing van repeatable annotations in Java 7 opnieuw de wrapper ervoor te declareren. Ook kunnen deze met één functieaanroep uitgelezen worden. Tegenover het uitlezen van de gedeclareerde container en hier de repeatable annotations van uit te lezen. Type annotations kunnen gebruikt worden om compile time validatie toe te voegen met behulp van een framework en/of een eigen implementatie.

Er is vastgelegd dat het uitlezen van repeatable annotations d.m.v reflection nog steeds een erg dure operatie is. Maar dat komt meer vanwege het verplichte gebruik van reflection.

Lambdas

Lambda brengt behoorlijk veel voor- en nadelen met zich mee. Zo is het dankzij de introductie van lambda expressies niet langer nodig om uitgebreide code te schrijven om uiteindelijk een enkele abstracte methode te implementeren. Dit kan nu worden opgelost met een enkele regel code.

Daarnaast heeft lambda een aantal grote nadelen, waaronder bijvoorbeeld de beperking van testbaarheid of het kunnen hergebruiken van een lambda. Gelukkig heeft Oracle hier goed over nagedacht en hebben zij ervoor gezorgd dat deze nadelen grotendeels weggenomen kunnen worden door gebruik te maken van Method References.

Streams

Streams zijn een handig maar hardnekkig hulpmiddel. Het kan je leven makkelijker maken, maar heeft een paar regels waar aan gehouden moet worden. De code is korter ten opzichte van voorgaande alternatieven, wat de leesbaarheid enorm bevordert, maar de performance is niet altijd beter. Bij het gebruik van streams is het dus belangrijk om te kijken of de performance acceptabel is voor de machine waar deze op wordt toegepast.

Functional Interfaces

De nieuwe functionaliteit van Java 8 wat betreft functional interfaces zorgt aanzienlijk voor minder benodigde code. De SAM interfaces uit de voorgaande versies zijn compatible gemaakt met de nieuwe lambda notatie en zijn ook verkort wat betreft de syntax. De functionaliteit van de functional interfaces heeft de taal meer in de richting van functional programming gestuurd.

Optionals

Optionals zijn een waardevolle toevoeging aan Java, deze mening kan echter verschillen per persoon. Wat leesbaarheid betreft is het bijvoorbeeld niet direct duidelijker of minder duidelijk, maar meer een persoonlijke voorkeur. Zowel het gebruik van Optionals en nested if-statements heeft namelijk zijn voor- en nadelen.

De voordelen van het daadwerkelijk toepassen van een Optional worden pas duidelijk wanneer deze gebruikt wordt als returnwaarde van een functie. Op die manier hoeft er namelijk geen rekening gehouden te worden met het terugkrijgen van een null-waarde en kan het aantal NullPointerExceptions daadwerkelijk verminderd worden. Helaas is juiste implementatie van Optionals wel wat aan de lastige kant, aangezien deze in eerste instantie redelijk lastig te begrijpen zijn.

Hoofdvraag

Betreft de vraag of deze functionaliteiten een goede toevoeging geven en of deze wel thuishoren in Java 8, kan er worden gezegd dat deze wel degelijk goed bevonden worden. Vrijwel alle huidige talen ondersteunen over het algemeen één of meer van de nieuwe features in Java 8 waardoor Java in eerste instantie een achterstand had op de andere talen. Met de introductie van Java 8 en zijn features wordt de taal in positieve zin uitgebreid. Zo hebben veel mensen gewacht op de lambda introductie in Java en is dat met deze release goed meegenomen.

De implementatie lijkt echter af toe aan de wensen over te laten. Zo heerst er binnen de Java community nog wel twijfels op de manier waarop bijvoorbeeld lambda is geïmplementeerd, maar werkt het op de manier waarop het zou moeten.

Literatuurlijst

- (2014). In R. Warburton, *Java Lambdas Pragmatic Functional Programming* (pp. 103 - 104). *Annotation type repeatable*. (2014, 10 12). Opgehaald van Stackoverflow. :
<http://stackoverflow.com/a/15517544/2076351>
- Anonymous function*. (2014, 11 6). Opgehaald van
http://en.wikipedia.org/wiki/Anonymous_function
- Comparison fork/join to streams*. (2014, 11 8). Opgehaald van Infoq:
<http://www.infoq.com/articles/forkjoin-to-parallel-streams>
- Compatibility Guide for JDK 8*. (2014, 12 02). Opgehaald van Oracle:
<http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html>
- Difference between final and effectively final*. (2014, 11 06). Opgehaald van Stackoverflow:
<http://stackoverflow.com/questions/20938095/difference-between-final-and-effectively-final>
- Element type*. (2014, 12 3). Opgehaald van
<https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/ElementType.html>
- Goetz. (2014, 11 26). *From Lambdas to Bytecode*. Opgehaald van
<http://medianetwork.oracle.com/video/player/1113272510001>
- How to use predicate in Java 8*. (2014, 11 18). Opgehaald van
<http://howtodoinjava.com/2014/04/04/how-to-use-predicate-in-java-8/>
- Interface Consumer*. (2014, 11 20). Opgehaald van
<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>
- Interface Predicate*. (2014, 11 18). Opgehaald van Oracle:
<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>
- Introduction to Functional Interfaces - A concept recreated in Java 8*. (2014, 11 10). Opgehaald van <http://java.dzone.com/articles/introduction-functional-1>
- Is there a performance impact when calling ToList()?* (2014, 11 10). Opgehaald van Stackoverflow.: <http://stackoverflow.com/a/15517544/2076351>
- Java 8 consumer and supplier*. (2014, 11 20). Opgehaald van
<http://www.byteslounge.com/tutorials/java-8-consumer-and-supplier>
- Java 8 Friday: 10 Subtle Mistakes When Using the Streams API*. (2014, 12 7). Opgehaald van
<http://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/>
- Oracle.
<http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>
- Java 8 Functional Interfaces*. (2014, 11 10). Opgehaald van
<http://radar.oreilly.com/2014/08/java-8-functional-interfaces.htm>
- Java 8 new type annotations*. (2014, 12 3). Opgehaald van Oracle:
https://blogs.oracle.com/java-platform-group/entry/java_8_s_new_type
- Java 8 Optional Objects*. (2014, 11 19). Opgehaald van InformaTech:
<http://blog.informatech.cr/2013/04/10/java-optional-objects/>
- Java Tester - What Version of Java Are You Using?* (2014, 12 1). Opgehaald van JavaTester:
<http://javatester.org/version.html>

- Java8 Lambda performance vs public functions.* (2014, 10 8). Opgehaald van Stackoverflow. : <http://stackoverflow.com/questions/26252672/java8-lambda-performance-vs-public-functions>
- Jhades. (2014, 11 18). *Java 8 Optional: How to Use it.* Opgehaald van <http://blog.jhades.org/java-8-how-to-use-optional/>
- Johnson, H. (2014, 11 18). *Java 8 Optional.* Opgehaald van <http://huguesjohnson.com/programming/java/java8optional.html>
- JSR 335: Lambda Expressions for the Java™ Programming Language.* (2014, 11 26). Opgehaald van JSR: Java Specification Request: <https://jcp.org/en/jsr/detail?id=335>
- JUnit: Testing Exception With Java 8 And Lambda Expressions.* (2014, 11 04). Opgehaald van CodeLeak. : <http://blog.codeleak.pl/2014/07/junit-testing-exception-with-java-8-and-lambda->
- JUnit-team. (2014, 11 26). *Test-runners.* Opgehaald van <https://github.com/junit-team/junit/wiki/Test-runners>
- Lambdas have come to Java!* (2014, 10 28). Opgehaald van Youtube: <http://youtu.be/ZP35pviNsU0?t=19m21s>
- Lambdas have come to Java!* (2014, 10 14). Opgehaald van Youtube: <http://youtu.be/ZP35pviNsU0?t=16m47s>
- Local Classes.* (2014, 12 9). Opgehaald van <http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class>
- Love and hate for Java 8.* (2014, 12 1). Opgehaald van Infoworld.: <http://www.infoworld.com/article/2611558/application-development/love-and-hate-for-java-8.html>
- Memory usage of Java objects: general guide.* (2014, 11 18). Opgehaald van JavaMex: http://www.javamex.com/tutorials/memory/object_memory_usage.shtml
- Method references.* (2014, 10 18). Opgehaald van Oracle: <https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>
- Microsoft. (2014, 10 7). *Lambda Expressions (C# Programming Guide).* Opgehaald van <http://msdn.microsoft.com/en-us/library/bb397687.aspx>
- Once Upon an Oak.* (2014, 11 25). Opgehaald van Artima: <http://www.artima.com/weblogs/viewpost.jsp?thread=7555>
- Optional in Java 8 Cheat Sheet.* (2014, 11 18). Opgehaald van Dzone: <http://java.dzone.com/articles/optional-java-8-cheat-sheet>
- Oracle. (2014, 11 5). *What is an interface?* Opgehaald van <https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>
- Oracle. (2014, 12 2). *Windows XP and Java.* Opgehaald van <https://www.java.com/en/download/faq/winxp.xml>
- Oracle. (sd). *Annotation Type FunctionalInterface.* Opgehaald van <https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>
- Package java.util.stream.* (2014, 10 28). Opgehaald van <http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>
- Repeating annotations.* (2014, 10 12). Opgehaald van Java tutorials: <http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html>

- Repeating annotations in java 8.* (2014, 11 10). Opgehaald van Softwarecave:
<http://stackoverflow.com/a/15517544/2076351>
- Sorting Algorithm.* (2014, 28 10). Opgehaald van Wikipedia:
http://en.wikipedia.org/wiki/Sorting_algorithm#Stability
- State of Lambda.* (2014, 11 26). Opgehaald van
<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-3.html>
- Target annotation.* (2014, 12 3). Opgehaald van
<https://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Target.html>
- Teruggaan naar Java 7 na installatie van Java 8.* (2014, 12 11). Opgehaald van
https://java.com/nl/download/help/java_revert.xml
- The Dark Side Of Lambda Expressions in Java 8.* (2014, 10 14). Opgehaald van Takipiblog:
<http://www.takipiblog.com/the-dark-side-of-lambda-expressions-in-java-8/>
- The Java Community Process(SM) Program.* (2014, 11 25). Opgehaald van JCP:
<https://jcp.org/en/home/index>
- Tired of Null Pointer Exceptions? Try using Java SE 8's Optional!* (2014, 11 19). Opgehaald van Oracle:
<http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html>
- Trail: the reflection API.* (2014, 12 9). Opgehaald van Oracle:
<http://docs.oracle.com/javase/tutorial/reflect/index.html>
- Transparency (behaviour).* (2014, 11 14). Opgehaald van
http://en.wikipedia.org/wiki/Transparency_%28behavior%29
- Type annotations?* (2014, 12 2). Opgehaald van Oracle:
https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html
- Wat zijn de systeemvereisten voor Java?* (2014, 12 2). Opgehaald van
<http://java.com/nl/download/help/sysreq.xml>
- Webinar: Lambdas Have Come To Java 8.* . (2014, 10 7). Opgehaald van Typesafe:
<http://typesafe.com/blog/webinar-lambdas-have-come-to-java-8>
- Weiss, T. (2014, 11 26). *Compiling Lambda expressions: Scala vs. Java 8.* Opgehaald van
<http://blog.takipi.com/compiling-lambda-expressions-scala-vs-java-8/>
- What is a lambda (function)?* (2014, 10 7). Opgehaald van Stackoverflow.:
<http://stackoverflow.com/questions/16501/what-is-a-lambda-functio>
- What is a lambda expression.* (2014, 10 7). Opgehaald van Dice:
<http://news.dice.com/2013/08/13/what-is-a-lambda-expression/>
- What's new in Java 8 lambda's.* (2014, 11 10). Opgehaald van
<http://radar.oreilly.com/2014/04/whats-new-in-java-8-lambdas.html>
- Yet Another Lambda Tutorial.* (2014, 10 28). Opgehaald van pythonconquerstheuniverse:
http://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/

Bijlagen

Bijlage A - Lambda performance test eerste versie

Zie bijlage op CD.

Locatie: Snippets > Lambda > Lambda_2_PerformanceTest > src

Bestand: MainOld.java

Bijlage B - Lambda performance test aangepaste versie

Zie bijlage op CD.

Locatie: Snippets > Lambda > Lambda_2_PerformanceTest > src

Bestand: Main.java

Bijlage C - Lambda performance test resultaat

Performance met List Size: 10

Loops (Runs)	Totale Grote	Methode	Duur (ms)
2.500.000	2.500.000	lambda foreach	513
2.500.000	2.500.000	lambda collect	522
2.500.000	2.500.000	default loop	327
5.000.000	5.000.000	lambda foreach	1030
5.000.000	5.000.000	lambda collect	927
5.000.000	5.000.000	default loop	604
7.500.000	7.500.000	lambda foreach	1243
7.500.000	7.500.000	lambda collect	1463
7.500.000	7.500.000	default loop	877
10.000.000	10.000.000	lambda foreach	1613
10.000.000	10.000.000	lambda collect	1948
10.000.000	10.000.000	default loop	1259

Performance met List Size: 100

Loops (Runs)	Totale Grote	Methode	Duur (ms)
250.000	750.000	lambda foreach	111
250.000	750.000	lambda collect	129
250.000	750.000	default loop	132
500.000	1.500.000	lambda foreach	227
500.000	1.500.000	lambda collect	247
500.000	1.500.000	default loop	266
750.000	2.250.000	lambda foreach	364
750.000	2.250.000	lambda collect	400
750.000	2.250.000	default loop	456
1.000.000	3.000.000	lambda foreach	449
1.000.000	3.000.000	lambda collect	515
1.000.000	3.000.000	default loop	584

Performance met List Size: 1000

Loops (Runs)	Totale Grote	Methode	Duur (ms)
25.000	575.000	lambda foreach	199
25.000	575.000	lambda collect	105
25.000	575.000	default loop	166
50.000	1.150.000	lambda foreach	331
50.000	1.150.000	lambda collect	200
50.000	1.150.000	default loop	253
75.000	1.725.000	lambda foreach	502
75.000	1.725.000	lambda collect	331
75.000	1.725.000	default loop	477
100.000	2.300.000	lambda foreach	664
100.000	2.300.000	lambda collect	397
100.000	2.300.000	default loop	532

Performance met List Size: 10000

Loops (Runs)	Totale Grote	Methode	Duur (ms)
2.500	612.500	lambda foreach	239
2.500	612.500	lambda collect	233
2.500	612.500	default loop	247
5000	1.225.000	lambda foreach	415
5000	1.225.000	lambda collect	440
5000	1.225.000	default loop	539
7.500	1.837.500	lambda foreach	709
7.500	1.837.500	lambda collect	638
7.500	1.837.500	default loop	736
10.000	2.450.000	lambda foreach	851
10.000	2.450.000	lambda collect	849
10.000	2.450.000	default loop	1014

Performance met List Size: 100000


Loops (Runs)	Totale Grote	Methode	Duur (ms)
250	634.000	lambda foreach	391
250	634.000	lambda collect	421
250	634.000	default loop	476
500	1.268.000	lambda foreach	590
500	1.268.000	lambda collect	616
500	1.268.000	default loop	758
750	1.902.000	lambda foreach	847
750	1.902.000	lambda collect	931
750	1.902.000	default loop	1178
1.000	2.536.000	lambda foreach	1140
1.000	2.536.000	lambda collect	1286
1.000	2.536.000	default loop	1552

Performance met List Size: 1000000

Loops (Runs)	Totale Grote	Methode	Duur (ms)
25	622.150	lambda foreach	273
25	622.150	lambda collect	280
25	622.150	default loop	328
50	1.244.300	lambda foreach	562
50	1.244.300	lambda collect	602
50	1.244.300	default loop	676
75	1.866.450	lambda foreach	852
75	1.866.450	lambda collect	895
75	1.866.450	default loop	976
100	2.488.600	lambda foreach	1283
100	2.488.600	lambda collect	1287
100	2.488.600	default loop	1479

Bijlage D – Specificaties testsysteem M

HWiNFO64 - System Summary



CPU

Intel Core i5-3550

Stepping: E1

Codename: Ivy Bridge-DT

SSPEC: SR0P0

Platform: Socket H2 (LGA1155)

Cache: 4 x (32 + 32 + 256) + 6M

TDP: 77 W

Cores: 4

Logical: 4

μCU: 12

Prod. Unit:

CPU #0

Features:

MMX	3DNow!	3DNow!-2	SSE	SSE-2	SSE-3	SSSE-3
SSE4A	SSE4.1	SSE4.2	AVX	AVX2	AVX-512	
BMI	ABM	TBM	FMA	ADX	XOP	
DEP	VMX	SMX	SMEP	SMAP	TSX	MPX
EM64T	EIST	TM1	TM2	HTT	Turbo	
AES-NI	RDRAND	RDSEED	SHA			

Operating Point	Clock	Ratio	Bus	VID
CPU LFM (Min)	1600.0 MHz	x16	100.0 MHz	-
CPU HFM (Max)	3300.0 MHz	x33	100.0 MHz	-
CPU Turbo	3700.0 MHz	x37	100.0 MHz	-
CPU Status	-	-	99.8 MHz	0.8456 V

GPU

Gainward GeForce GTX 770

NVIDIA GeForce GTX 770

GK104-425

PCIe v3.0 x16 (8.0 Gb/s) @ x16 (2.5 Gb/s)

GPU #0

2 GB

GDDR5 SDRAM

256-bit

ROPs: 32

Shaders: Unified: 1536

Current Clocks (MHz)

GPU: 135.0

Memory: 162.0

Shader: -

Memory Modules

[#0] G Skill F3-10666CL9-4GBNT

Size: 4 GB

Clock: 667 MHz

ECC: N

Type: DDR3-1333 / PC3-10600 DDR3 SDRAM UDIMM

Freq	CL	RCD	RP	RAS	RC	Ext.	V
666.7	9	9	9	24	33	-	1.50
600.0	8	8	8	22	30	-	1.50
533.3	7	7	7	20	27	-	1.50
466.7	7	7	7	17	23	-	1.50
400.0	6	6	6	15	20	-	1.50

Memory

Size: 8 GB

Type: DDR3 SDRAM

Clock: 665.2 MHz = 6.67 x 99.8 MHz

Mode: Dual-Channel

CR: 1T

Timing: 9 - 9 - 9 - 24 tRC tRFC 107

Operating System: UEFI Boot

Microsoft Windows 7 Ultimate (x64) Build 7601

Motherboard: ASRock B75M R2.0

Chipset: Intel B75 (Panther Point-M Enhanced)

BIOS Date: 08/22/2012

BIOS Version: P1.00

UEFI

Drives:

- ✓ SATA 6 Gb/s: Samsung SSD 840 EVO 250GB [250 GB]
- ✓ SATA 6 Gb/s: TOSHIBA DT01ACA100 [1000 GB, 23652KB]
- ATAPI: ATAPI iHAS124 D [DVD+R DL]

Bijlage E - Annotations performance experiment 1

Toelichting

Dit experiment wordt opgebouwd in Java met behulp van het JUnit framework voor het vastleggen van executie tijden.

Dit experiment wordt uitgevoerd op testsysteem M. De specificaties voor dit systeem zijn de vinden in bijlage D.

Om het verschil in performance te achterhalen is er besloten om een klasse te maken met twee functies. Waarvan één drie (repeatable)annotaties uitleest d.m.v reflection. En één functie een array van drie objecten. Zowel de objecten en (repeatable)annotaties bevatten dezelfde data. Namelijk dayOfMonth, hour en repeatable.

Om een duidelijk beeld te krijgen van de verschillen in executie tijden, wordt elke functie ieder door drie unit tests uitgevoerd. Deze tests verschillen in één factor: De hoeveelheid aan uitvoeringen. Dit heeft zes unit tests als resultaat.

Source code

Zie bijlage op CD.

Locatie: Snippets > Annotations

Bijlage F – Annotations performance experiment 2

Toelichting

Dit experiment wordt opgebouwd in Java met behulp van het JUnit framework voor het vastleggen van executie tijden.

Dit experiment wordt uitgevoerd op testsysteem M. De specificaties voor dit systeem zijn de vinden in bijlage D.

Source code

Zie bijlage op CD.

Locatie: Snippets > Annotations

Bijlage G – Opzet toepassen repeatable annotations voor unit testing

Toelichting

Dit experiment wordt opgebouwd in Java met behulp van het JUnit framework voor unit testing. Het doel van het experiment is Repeatable annotations toepassen in unit testing. Hiervoor wordt de BlockJUnit4ClassRunner class overgeërft en opnieuw geïmplementeerd om de @RepeatableTest annotations te herkennen.

[//verwijzing naar node](#)

Bijlage H – Lambda expressies

```
( ) -> { }                // No parameters; result is void

( ) -> 42                  // No parameters, expression body
( ) -> null                // No parameters, expression body
( ) -> { return 42; }      // No parameters, block body with return
( ) -> { System.gc(); }    // No parameters, void block body

( ) -> {
    if (true) return 12;
    else {
        int result = 15;
        for (int i = 1; i < 10; i++)
            result *= i;
        return result;
    }
}                          // Complex block body with returns

(int x) -> x+1              // Single declared-type parameter
(int x) -> { return x+1; }  // Single declared-type parameter
(x) -> x+1                  // Single inferred-type parameter
x -> x+1                    // Parens optional for single inferred-type case

(String s) -> s.length()   // Single declared-type parameter
(Thread t) -> { t.start(); } // Single declared-type parameter
s -> s.length()             // Single inferred-type parameter
t -> { t.start(); }        // Single inferred-type parameter

(int x, int y) -> x+y       // Multiple declared-type parameters
(x,y) -> x+y                // Multiple inferred-type parameters
(final int x) -> x+1        // Modified declared-type parameter
(x, final y) -> x+y         // Illegal: can't modify inferred-type parameters
(x, int y) -> x+y           // Illegal: can't mix inferred and declared types
```