Noël Vaes

Java Trainer & Consultant

Java Advanced 1

PXL-versie
Academiejaar 2019-2020

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Vrijwel alle namen van software- en hardwareproducten die in deze cursus worden genoemd, zijn tegelijkertijd ook handelsmerken en dienen dienovereenkomstig te worden behandeld.

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand of openbaar worden gemaakt in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnamen of op enige andere manier, zonder voorafgaande schriftelijke toestemming van de auteur. De enige uitzondering die hierop bestaat, is dat eventuele programma's en door de gebruiker te typen voorbeelden mogen worden ingevoerd, opgeslagen en uitgevoerd op een computersysteem, zolang deze voor privé-doeleinden worden gebruikt, en niet bestemd zijn voor reproductie of publicatie.

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

Ondanks alle aan de samenstelling van deze tekst bestede zorg, kan de auteur geen aansprakelijkheid aanvaarden voor eventuele schade die zou kunnen voortvloeien uit enige fout, die in deze uitgave zou kunnen voorkomen.

30/11/2019

Copyright© 2019 Noël Vaes

Inhoudsopgave

Hoofdstuk 1: Exception handling **6**
1.1 Inleiding 6
1.2 *Exceptions* afhandelen 6
1.2.1 Een *exception* veroorzaken 7
1.2.2 Een *exception* opvangen 8
1.2.3 Meerdere *exceptions* opvangen 10
1.2.4 Gemeenschappelijke *exception handlers* 11
1.2.5 Het finally blok 13
1.3 *Exceptions* genereren 15
1.3.1 Het throw-*statement* 15
1.3.2 *Exceptions* bij vervangen methoden 16
1.4 Soorten *exceptions* 16
1.4.1 *Exceptions* versus *errors* 16
1.4.2 *Checked exceptions* versus *runtime exceptions* 17
1.5 Zelf een *exception*-klasse maken 18
1.6 *Exceptions* opvangen, inpakken en verder gooien 19
1.7 Samenvatting 20
Hoofdstuk 2: Generieken **22**
2.1 Inleiding 22
2.2 Generieke klassen 22
2.2.1 Generieken definiëren 23
2.2.2 Het gebruikte type inperken 28
2.2.3 Onbepaald type 30
2.2.4 Subklassen van generieke klassen 31
2.3 Generieke interfaces 31
2.4 Generieke methoden 35
2.4.1 Formele generieke parameters 35
2.4.2 Formele generieke parameters met *wildcards* 35
2.4.3 Formele generieke parameters met *bounded wildcards* 36
2.4.4 Type-parameters 38
2.5 Achter de schermen van de generieken 39
2.6 Arrays en generieken 40
2.7 Samenwerking tussen oude en nieuwe code 41
2.8 Samenvatting 42
Hoofdstuk 3: Geneste en anonieme klassen **43**
3.1 Inleiding 43
3.2 Gewone geneste klassen (inner classes) 43
3.3 Lokale geneste klassen (local inner classes) 45
3.4 Anonieme geneste klassen (anonymous inner classes) 46
3.5 Static geneste klassen (static nested classes) 47
3.6 Samenvatting 49
Hoofdstuk 4: Lambda Expressions **50**
4.1 Inleiding 50
4.2 Functionele interfaces 52
4.3 Definitie van lambda expressions 52
4.4 Methodereferenties 55
4.4.1 Statische methoden van een klasse of interface 55
4.4.2 Methoden van een gebonden object 57
4.4.3 Methoden van een ongebonden object 58

Copyright© 2019 Noël Vaes

4.4.4 Constructorreferenties....58
4.5 Standaard functionele interfaces....61
4.5.1 Predicate<T>....61
4.5.2 Function<T,R>....62
4.5.3 Consumer<T>....63
4.6 Inleiding: interne versus externe iteraties....64
4.7 Bron van *streams*....66
4.8 Bewerkingen....68
4.8.1 Eindbewerkingen....68
4.8.2 Tussenliggende bewerkingen....72
4.9 Samenvatting....76
Hoofdstuk 5: Streaming API....77
5.1 Inleiding: interne versus externe iteraties....77
5.2 Bron van *streams*....78
5.3 Bewerkingen....80
5.3.1 Eindbewerkingen....80
5.3.2 Tussenliggende bewerkingen....84
5.4 Samenvatting....88
Hoofdstuk 6: Collections....89
6.1 Het *Collections Framework*....89
6.2 De interface Collection en implementaties....89
6.2.1 List....92
6.2.2 Set....98
6.2.3 SortedSet & NavigableSet....103
6.2.4 Queue....105
6.2.5 Deque....107
6.2.6 Vergelijking tussen de implementaties....108
6.2.7 Het sorteren van verzamelingen....109
6.2.8 Collections en streams....117
6.3 De interface Map en implementaties....118
6.3.1 Map....119
6.3.2 SortedMap & NavigableMap....122
6.3.3 Vergelijking tussen de implementaties....123
Hoofdstuk 7: Lezen en schrijven (I/O)....124
7.1 Inleiding....124
7.2 Mappen en bestanden....124
7.2.1 De interface Path....124
7.2.2 De klasse FileSystem....127
7.2.3 De klasse Files....127
7.2.4 De klasse File....130
7.3 IO-streams....130
7.3.1 Character streams....132
7.3.2 Byte streams....140
7.4 Object Serialization....145
7.4.1 Objecten serialiseren en deserialiseren....145
7.4.2 Klassen serialiseerbaar maken....146
7.4.3 Transiënte variabelen....147
7.4.4 Serialisatie en overerving....149
7.4.5 Versienummering....149
7.5 Programma-attributen....150
Hoofdstuk 8: Java via de commandolijn....154
8.1 Inleiding....154
8.2 Compileren....154
8.3 Modules maken....158

8.3.1 Inleiding....158
8.3.2 Een module definiëren....159
8.3.3 Pakketten exporteren....159
8.3.4 Afhankelijkheden van andere modules....160
8.3.5 Transitieve afhankelijkheden....163
8.4 JAR-bestanden maken....164
8.4.1 Basisprincipes van een JAR....164
8.4.2 Een JAR-bestand maken....165
8.4.3 Een JAR-bestand opnemen in het modulepad....167
8.4.4 Resources uit een JAR-bestand lezen....170
8.5 Programma's uitvoeren....171
8.6 Automatische modules....172
8.7 Linken....173
Hoofdstuk 9: Systeembronnen gebruiken....175
9.1 Inleiding....175
9.2 De *System*-klasse....175
9.2.1 Standaard-I/O streams....175
9.2.2 Systeemeigenschappen....180
9.2.3 Overige methoden....181
Hoofdstuk 10: Multithreading....183
10.1 Inleiding: multiprocessing en multithreading....183
10.2 Een nieuwe *thread* creëren....184
10.2.1 Subklasse van de klasse Thread....185
10.2.2 De interface Runnable....187
10.2.3 *Thread* met *lambda expression*....188
10.3 De levenscyclus van *threads*....189
10.4 De uitvoering van *threads* in de toestand RUNNABLE....190
10.4.1 De scheduler....190
10.4.2 Prioriteiten van *threads*....191
10.4.3 Preëmptieve multitasking....192
10.4.4 Coöperatieve multitasking....192
10.5 Daemon threads....193
10.6 De wachttoestand....194
10.6.1 De slaaptoestand....195
10.6.2 Wachten op de beëindiging van een andere *thread*....197
10.7 Synchronisatie van threads (monitoring)....198
10.7.1 Object locking....199
10.8 De Timer-klasse en de TimerTask-klasse....203
10.9 Concurrency framework....204
10.9.1 Concurrent collections....204
10.9.2 Atomaire objecten....206
10.9.3 Callable, ExecutorService and Future....208
10.10 Parallellisme met streams....210
Hoofdstuk 11: JUnit....212
11.1 Inleiding....212
11.2 Mijn eerste test....212
11.3 Integratie in de ontwikkelomgeving....213
11.4 De levenscyclus van een testklasse....214
11.5 Weergavenaam....216
11.6 Parameters....217
11.7 Testen uitschakelen....218
11.8 Assert-methoden....218
11.9 Grenzen testen....219
11.10 Exceptions testen....220

Copyright© 2019 Noël Vaes

Hoofdstuk 1: Exception handling

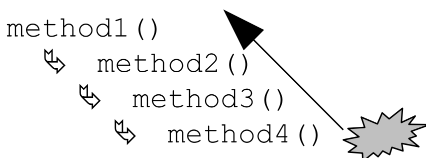
1.1 Inleiding

In ieder programma kunnen fouten optreden die de loop van het programma verstoren. Deze fouten kunnen zowel van de hardware als van de software komen. Zo'n fout noemt men ook wel een *exception* of *uitzondering*. Het is een uitzonderlijke gebeurtenis die op een bepaalde manier afgehandeld moet worden.

Definitie: Een *exception* is een gebeurtenis die optreedt tijdens de uitvoering van een programma en die de normale loop van het programma verstoort.

Wanneer er tijdens de uitvoering van het programma zo'n uitzondering optreedt, wordt er een *exception*-object gecreëerd. Dit *exception*-object bevat de omschrijving van de uitzonderingstoestand en de toestand van het programma op het moment dat de uitzondering optreedt. Dit object wordt aan het *runtime system* overhandigd. Dit noemt men *throwing an exception*. Het *exception*-object wordt als het ware naar het *runtime system* (JVM) gegooid, dat dan de nodige actie zal moeten ondernemen.

De JVM gaat vervolgens op zoek naar een methode die in staat is deze uitzonderingstoestand verder af te handelen. Zo'n methode noemt men een *exception handler*. In deze zoektocht worden de aanroepende methoden in omgekeerde volgorde doorlopen (*call stack*) totdat men een *exception handler* vindt voor dit specifieke uitzonderingsobject. Deze methode vangt de uitzondering op en onderneemt de nodige acties. Als er geen *exception handler* gevonden wordt, zal het programma abrupt eindigen.



Exception handling heeft een aantal voordelen ten opzichte van de traditionele methoden voor het opvangen van fouten:

1. De code voor het afhandelen van de fouten is **gegroepeerd** en **gescheiden** van de normale code van het programma. Dit maakt het programma veel overzichtelijker en beter onderhoudbaar.
2. Een uitzondering **borrelt** als het ware **omhoog** door de *call stack*[1]. Dit maakt het mogelijk dat ze wordt afgehandeld door de methode die uiteindelijk geïnteresseerd is in de uitzonderingstoestand. Tussenliggende methoden in de *call stack* hoeven zich niets van deze uitzonderingen aan te trekken. Het is dan ook niet nodig code te schrijven die de fout doorgeeft aan de aanroepende methode. Men dient enkel aan te geven dat bepaalde uitzonderingstoestanden kunnen optreden.
3. Een *exception object* is een object van een bepaalde klasse. Door het construeren van een **klassenhiërarchie** voor deze objecten is het mogelijk verschillende soorten uitzonderingen samen te brengen onder eenzelfde noemer. Een *exception handler* kan uitzonderingen van een bepaalde superklasse afhandelen.

1.2 Exceptions afhandelen

In deze paragraaf gaan we eerst een *exception* genereren om te zien hoe het systeem daarop reageert. Vervolgens zullen we deze *exception* zelf afhandelen in de code. Om

1 De *call stack* is de stapel van methoden die elkaar aangeroepen hebben vooraleer een bepaald stuk code uitgevoerd wordt.

Copyright© 2019 Noël Vaes

verschillende fouten af te handelen, kunnen we verschillende *exception handlers* maken ofwel een aantal fouten bundelen en ze laten afhandelen door één *exception handler*.

1.2.1 Een *exception* veroorzaken

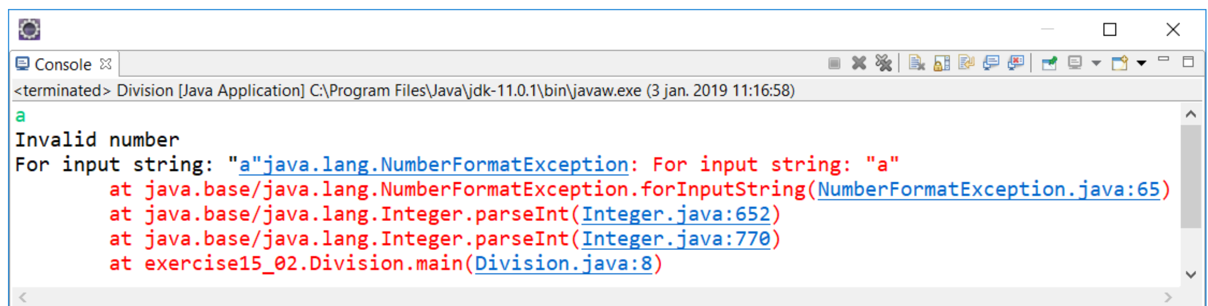
We vertrekken van het volgende voorbeeld dat twee getallen door elkaar deelt en het resultaat op het scherm toont.

```
public class Division {
   public static void main(String[] args) {
      Scanner keyboard = new Scanner(System.in);
      int num = Integer.parseInt(keyboard.next());
      int den = Integer.parseInt(keyboard.next());
      int div = num / den;
      System.out.format("%d/%d=%d", num,den,div);
      keyboard.close();
   }
}
```

Niets aan de hand op het eerste zicht. Stel dat we nu in plaats van een getal een letter invoeren.

Er wordt door de JVM een uitzondering gegenereerd en het programma wordt abrupt beëindigd. Er wordt een *exception* gegooid van de klasse

`java.lang.NumberFormatException.`



Afbeelding 1: Stacktrace bij een exception

Opdracht 1: Een exception genereren

- Compileer de code en voer ze uit. Test met gehele getallen, reële getallen en letters.

De methode `Integer.parseInt()` kan dus een bepaalde *exception* genereren. Dit vinden we tevens terug in de documentatie van deze methode onder de rubriek ***Throws***:

Copyright© 2019 Noël Vaes

Integer (Java SE 11 & JDK 11)

file:///C:/Java/jdk-11.0.1_doc-all/d

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP **Java SE 11 & JDK 11**

ALL CLASSES SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

parseInt

```
public static int parseInt(String s) throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a `String` containing the `int` representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Afbeelding 2: API-documentatie van de methode parseInt()

1.2.2 Een *exception* opvangen

In plaats van het programma abrupt te laten eindigen, kunnen we deze *exception* opvangen en de nodige stappen ondernemen.

```
method1()
  ↳ method2()
      ↳ method3()
          ↳ method4()
```

Uit de documentatie van de methode `Integer.parseInt()` weten we dat bij het aanroepen van deze methode een *exception* kan optreden. Het *exception*-object is van de klasse `NumberFormatException`.

De beschrijving van deze klasse vinden we ook in de documentatie:

NumberFormatException (Java SE 1

file:///C:/Java/jdk-11.0.1_doc-all

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP   Java SE 11 & JDK 11

ALL CLASSES   SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.lang

Class NumberFormatException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException

All Implemented Interfaces:

Serializable

```
public class NumberFormatException
extends IllegalArgumentException
```

Afbeelding 3: API-documentatie van de klasse NumberFormatException

De algemene syntax voor het opvangen van *exceptions* ziet er als volgt uit:

```
try {
    // Code die mogelijk een exception genereert
}
catch(Throwable exceptionObject){
    // Code die wordt uitgevoerd om de exception af te handelen
}
```

Vooreerst is er het `try`-codeblok. In dit codeblok worden de Java-*statements* geplaatst die eventueel een *exception* kunnen genereren.
Deze *exception* wordt vervolgens opgevangen in het `catch`-codeblok. Dit ziet er ongeveer uit als een methode met één parameter. Door middel van deze parameter wordt het *exception*-object doorgegeven aan het `catch`-blok. Het *exception*-object moet afgeleid zijn van de klasse `Throwable` of van een subklasse van deze klasse.

Toegepast op ons voorbeeld:

```
public class Division {
    public static void main(String[] args) {
        try {
            Scanner keyboard = new Scanner(System.in);
            int num = Integer.parseInt(keyboard.next());
            int den = Integer.parseInt(keyboard.next());
            int div = num / den;
            System.out.format("%d/%d=%d", num, den, div);
            keyboard.close();
        } catch (NumberFormatException nfe) {
```

```
            System.out.println("Invalid number");
        }
    }
}
```

De klasse `NumberFormatException` is afgeleid van de superklasse `Throwable` die deel uitmaakt van het pakket `java.lang`. De referentie naar het *exception*-object krijgt de naam `nfe`. Het is gebruikelijk deze naam samen te stellen uit de beginletters van elke woord van de *exception*-klasse: `NumberFormatException -> nfe`.

In het `catch`-blok hebben we de mogelijkheid om bijvoorbeeld een boodschap op het scherm te brengen. We kunnen hierbij ook gebruikmaken van de methode `getMessage()`. Dit is een methode van de klasse `Throwable`. Tevens kunnen we de *call stack* afdrukken met de methode `printStackTrace()` van de klasse `Throwable`.

```
      catch(NumberFormatException nfe) {
         System.out.println("Invalid number!");
         System.out.println(nfe.getMessage());
         nfe.printStackTrace();
      }
```

Opdracht 2: Een exception opvangen

- Zoek in de documentatie de beschrijving van de klasse `Throwable`.
- Voeg een *exception handler* toe in het programma "Division". Druk de foutboodschap en de *stack trace* op het scherm af.
- Compileer en voer de code uit.

1.2.3 Meerdere *exceptions* opvangen

Er kunnen meerdere soorten uitzonderingen optreden in een codeblok. In dit geval kan men voor ieder soort uitzondering een afzonderlijk `catch`-blok maken dat die specifieke uitzondering opvangt. De algemene syntax ziet er dan als volgt uit.

```
try{
   // Code die mogelijk een exception genereert
}
catch(ThrowableClass1 tc1){
   // Afhandeling van exceptions van klasse ThrowableClass1
}
catch(ThrowableClass2 tc2){
   // Afhandeling van exceptions van klasse ThrowableClass2
}
catch(ThrowableClass3 tc3){
   // Afhandeling van exceptions van klasse ThrowableClass3
}
catch(ThrowableClass4 tc4){
   // Afhandeling van exceptions van klasse ThrowableClass4
}
```

Stel dat we in ons voorbeeld proberen te delen door 0. We krijgen dan het volgende resultaat: