Java gevorderd

βeta VZW

(versie 2019.01-beta)

Inhoud

Hoofdstuk 1. Inleiding

1. Nieuw sinds Java 2 (1.4)

De basiscursus Java handelt grotendeels over Java 2 versie 1.4. In deze cursustekst bekijken we wat er nieuw is in latere versies van Java. Hierbij moeten we een onderscheid maken tussen nieuwigheden in de taal en nieuwigheden in de libraries of bibliotheken.

Nieuwigheden in de bibliotheken zijn dikwijls minder belangrijk omdat ze ook deel kunnen uitmaken van externe libraries die niet behoren tot een standaard Java installatie. Een bekend voorbeeld hiervan is de JodaTime library. De ondersteuning om met datums te werken was tot en met Java 7 ondermaats. De Date- en Calendar-klassen waren erg omslachtig om te gebruiken. Wanneer men in de praktijk met datums moest werken, gebruikte men meestal een extra bibliotheek: de JodaTime library. Aangezien deze bibliotheek geen speciale taalconstructies nodig had, kon ze met elke Java versie gebruikt worden. Vanaf Java 8 behoort de *java.time API* tot een standaard installatie van Java. Deze nieuwe bibliotheek vervangt grotendeels de JodaTime library.

Nieuwigheden in de taal zijn veel belangrijker. Zolang Java geen annotations ondersteunde was het onmogelijk om annotations te gebruiken, zelfs niet met externe libraries. Sommige van die constructies konden echter wel ondersteund worden met work-arounds zoals de XDoclet preprocessor die met een soort van annotations kon werken.

In dit overzicht bekijken we kort wat er veranderd is in de verschillende javaversies. We vermelden de wijzigingen in de taal en de belangrijkste klassen die zijn toegevoegd.

1. A. Nieuw in Java 5 (2004)

Java en .NET hebben altijd haasje-over gespeeld qua mogelijkheden. Toen Microsoft in .NET 1.1 de mogelijkheid aanbood om meta-informatie te bewaren in de broncode, kon Sun natuurlijk niet achterblijven. Eén van de nieuwigheden die het meeste invloed heeft gehad op de verdere ontwikkeling van de taal, waren de annotations. Ze zorgden er mee voor dat het Java EE platform een tweede adem kreeg. Door niet alleen Javacode te kunnen schrijven, maar bij die code ook informatie over die code te kunnen meegeven, heeft dat het ontwikkelen van applicaties in Java sterk vergemakkelijkt.

We kunnen hier een onderscheid maken tussen source annotations en runtime annotations. De @Override annotation is een source annotation. Ze zorgt ervoor dat de compiler bij het compileren controleert of de methode een methode in de basisklasse overschrijft. Wanneer dat niet het geval is, wordt er een compileerfout gegenereerd. De annotation wordt echter niet overgenomen in het .class bestand. Tijdens het uitvoeren kan ze dus niet meer worden opgevraagd. Een runtime annotation is toegankelijk in de Java Virtual Machine en kan

gevolgen hebben voor het uitvoeren van het programma. Vanaf Java 8 is er nog een uitbreiding gekomen bij de mogelijkheden van annotations. Ze kunnen niet alleen meer gebruikt worden bij declaraties, maar ook bij het gebruik van types.

Generics waren een tweede belangrijke verbetering in Java 5. Ze werden vooral gebruikt in de Collection classes (List, Set, Map, …). In plaats van een ArrayList te definiëren die toeliet dat eender welk object werd toegevoegd, werd bij de declaratie van het list-object meegegeven wat men kon bewaren in de lijst:

```
List<String> stringlijst = new ArrayList<String>();
```

Het object stringlijst kan alleen Strings bevatten. Het generics concept is echter nooit tot in de definitie van de bytecode geraakt. Het bleef een taalconstructie. De compiler moest ervoor zorgen dat er extra bytecode werd gegenereerd om te controleren of er alleen strings werden toegevoegd aan de lijst.

Andere aanpassingen waren minder revolutionair: het ondersteunen van enumerations, variabel aantal argumenten in functies en een foreach-loop constructie.

1. B. Java 6(2006)

Java 6 was minder revolutionair dan Java 5. Het was de eerste versie die niet meer ondersteund werd op Windows 9x (95/98). Op het gebied van de taal zelf veranderde er eigenlijk niets. Wat de ingebouwde libraries betrof werd JDBC 4.0 ondersteund. Met JDBC 4.0 kan men gebruik maken van autoloading van de driver.

1. C. Java 7(2011)

Java 7 zorgde voor een aantal aanpassingen in de taal waardoor sommige zaken wat gemakkelijker werden. De diamond operator zorgde ervoor dat men het type bij de declaratie van een generics object niet moest herhalen:

```
List<String> stringlijst = new ArrayList<>();
```

In plaats van

```
List<String> stringlijst = new ArrayList<String>();
```

Het try-with-resources statement maakte het schrijven van een try… finally gemakkelijker:

```
try(Connection c = new Connection(…)){
…
}
```

In plaats van

```
Connection conn = null;
```

```
try{
   ….
}finally{
  conn.close();
}
```

Verder introduceerde Java 7 ook het gebruik van Strings in switch-statements. Het komt er dus in feite op neer dat men niets nieuws kon doen in Java. Bestaande constructies werden alleen wat eenvoudiger gemaakt.

1. D. Java 8(2014)

Java 8 introduceerde weer enkele revolutionaire aanpassingen aan de taal. De belangrijkste hiervan is waarschijnlijk de lambda expression. Lambda expressions zorgen ervoor dat functies als argument of returnwaarde gebruikt kunnen worden in een andere functie. Ze vormen eigenlijk een uitbreiding op anonymous inner classes. Stel dat we een lijst met namen hebben die we omgekeerd willen sorteren. We kunnen de namen in een StringBuilder zetten. Die heeft een reverse()-methode. De strings die daar uitkomen, kunnen we vergelijken met compareTo:

```
Arrays.sort(namen, new Comparator<String>(){
  @Override
  public int compare(String a, String b) {
    return new StringBuilder(a).reverse().toString().compareTo(new
StringBuilder(b).reverse().toString());
  }
});
```

De interface Comparator is een zogenaamde functional interface. Er is maar één abstracte methode (compare). In plaats van een object te maken van de klasse Comparator en vervolgens de compare()-methode te implementeren, kunnen we dit omzetten naar een lambda expression. Een lambda expression bestaat uit drie delen: de argumenten, een pijl (->) en de function body. De code wordt nu:

```
Arrays.sort(namen, (String a, String b) -> new
StringBuilder(a).reverse().toString().compareTo(new
StringBuilder(b).reverse().toString()));
```

Daarmee samenhangend ondersteunt Java 8 ook default methodes in een interface. Een lambda expression kan alleen gebruikt worden wanneer de interface maar één methode bevat. De naam van de methode wordt immers niet vermeld in de expression. In de Comparator-interface zijn er echter nog meer methodes voorzien. Er is bijvoorbeeld ook nog een reversed()-methode. Dit is echter geen abstracte methode. Er is default-code voorzien en ze is als volgt gedeclareerd:

```
default Comparator<T>   reversed() {…}
```

Tussen de accolades staat de code die moet worden uitgevoerd. Door het keyword default toe te voegen, mogen we in een interface code schrijven.

Default methodes zijn ook een manier om interfaces uit te breiden terwijl ze nog steeds backward compatible zijn. Het is tevens een manier om een soort van multiple inheritance toe te laten in Java(zonder state).

Op het niveau van libraries introduceert Java 8 ook een langverwachte verbetering. Datums worden nu beschreven via de nieuwe Java DateTIme API (java.time package). Die API is afgeleid van het JodaTime package en vergemakkelijkt het werken met datums aanzienlijk.

1. E. Java 9(2017)

Java 9 voegde geen nieuwe taalconstructies toe. Maar er werd wel een belangrijk concept gelanceerd: modules. Modules brengen het concept van *dependencies* binnen in Java. Via build tools zoals Maven en Gradle konden we al lang afhankelijkheden definiëren tussen bibliotheken. Vanaf Java 9 zijn modules een onderdeel van Java. Een module is een soort van superklasse. We moeten bepalen welke packages van de module beschikbaar zijn voor anderen modules (*public*) en we kunnen ook bepalen welke andere modules deze module nodig heeft.

Een module bevat een module-info.java bestand dat er als volgt zou kunnen uitzien:

```
module org.betavzw.java9.modules.auto {
    requires org.betavzw.java9.modules.engines;
    exports org.betavzw.java9.modules.car.handling;
}
```

Deze module heeft de module org.betavzw.java9.modules.engines nodig en maakt het package org.betavzw.java9.modules.car.handling beschikbaar voor andere modules.

Door Java modulair te maken, wordt het gemakkelijker om Java te installeren op kleinere apparaten.

Verder is er nog een uitbreiding op het try-with-resources statement van Java 7. Vanaf Java 9 is het mogelijk om een variabele te gebruiken die gedeclareerd is buiten het try statement:

```
Connection dbCon = DriverManager.getConnection("url", "user", "password");
try(dbCon){
…
}
```

Tenslotte is er ook nog de JShell waarmee we Java kunnen gebruiken in een Read-Eval-Print-Loop (REPL). Het komt erop neer dat we Java statements interactief kunnen gebruiken.

1. F. Java 10(2018)

Java 10 introduceert het *var* keyword voor *local variable type inference.* Tijdens het compileren wordt er bepaald wat het type van een variabele is, zonder dat we dat extra moeten meegeven:

```
HashMap<String,String> lijst = new HashMap<String,String>();
```

Kan nu het volgende worden:

```
var lijst = new HashMap<String,String>();
```

2. Nieuw in de Java Virtual Machine

2. A. Ondersteuning voor andere talen dan Java

Vanaf Java 7 ondersteunt de Java Virtual Machine ook andere talen dan Java. Met de *invokedynamic* instructie worden ook *dynamically typed* talen ondersteund. Java is een *statically typed language*. Dat wil zeggen dat we bij het schrijven van het programma voor elke variabele moeten declareren wat het type is. De *type checking* gebeurt tijdens het compileren. *Type checking* houdt in dat er gecontroleerd wordt of een variabele wordt ingevuld met het juiste type. Het statement

```
String naam = 42;
```

zal niet aanvaard worden tijdens het compileren. Het is dus onmogelijk om een class bestand te generen dat deze code bevat.

Een *dynamically typed language* doet de type checking tijdens het uitvoeren van het programma. We geven geen type mee in de programma code. In een taal zoals *Ruby* kunnen we het volgende schrijven (zonder *naam* eerst gedeclareerd te hebben):

```
naam = "Joske"
```

Aangezien de compiler niet meer altijd kan weten welk type een bepaalde variabele is, worden bepaalde Java constructies plots een stuk ingewikkelder voor de compiler:

```
… = a + b;
```

Voor een *statically typed language* zoals Java is het duidelijk wat de plus operator moet doen. Wanneer a en b integers zijn, moet de integer-optelling gebruikt worden. Wanneer het strings zijn, moet de concatenatie gebruikt worden. In een *dynamically typed language* zal de virtual machine pas tijdens het uitvoeren weten wat er juist bewaard wordt in de variabelen a en b. Pas dan kan er beslist worden of er een optelling of een concatenatie moet gebeuren.

De belangrijkste talen die gebruikt kunnen worden in de Java Virtual Machine naast Java zijn:

- Scala: een functionele programmeertaal die ook Java bibliotheken kan gebruiken
- Clojure: een dialect van LISP (functionele programmeertaal)
- Groovy: een scripting taal

- JRuby en Jython: implementaties van Ruby en Python die kunnen worden uitgevoerd in een Java Virtual Machine.

2. B. Veranderingen in Garbage collection

Garbage collection is het proces waarbij geheugen dat niet meer gebruikt wordt, vrijgegeven wordt. We moeten hierbij in eerste instantie een onderscheid maken tussen *stack*- en *heapgeheugen*. Wanneer we de volgende functie hebben:

```
public void printInformation() {
  String naam="Joske Vermeulen";
  ArrayList<String> lijst = new ArrayList<>();
  int aantal = 5;
  for(int i= 0; i < aantal; i++) {
    lijst.add(naam);
  }
}
```

dan kunnen we twee soorten variabelen onderscheiden:

1. de primitieve variabelen *aantal* en *i.*;
2. de objecten *naam* en *lijst*.

Primitieve variabelen in een functie worden bewaard op de stack. Op het einde van een functie wordt de stack leeggemaakt. Dat wil zeggen dat de geheugenplaats die door primitieve variabelen wordt ingenomen, automatisch wordt leeggemaakt. Voor objecten wordt de nodige plaats voorzien op de *heap*. Die geheugenplaatsen worden niet automatisch vrijgegeven. In "oudere" programmeertalen zoals C was de programmeur zelf verantwoordelijk voor het vrijgeven van die geheugenplaatsen voordat de functie verlaten werd. In "managed" programmeertalen zoals Java en .NET wordt die taak uitgevoerd door het *garbage collection* proces.

Een garbage collector werkt in twee fazen: de *mark* faze en de *sweep* faze. Tijdens de *mark* faze overloopt het proces de heap en zoekt naar geheugenplaatsen die niet meer toegankelijk zijn in het programma. In de functie *printInformation* zijn de geheugenplaatsen die ingenomen worden door de objecten *lijst* en *naam* niet meer toegankelijk na het uitvoeren van de functie. Tijdens de *sweep* faze worden de geheugenplaatsen die gemarkeerd werden in de *mark* faze vrijgemaakt.

Het overlopen van het geheugen op zoek naar geheugenplaatsen die niet meer toegankelijk zijn kan een langdurig proces zijn wanneer de applicatie veel geheugen gebruikt. Daarom delen garbage collectors het geheugen in generaties in. Voor Java zijn dat er drie: *young, tenured* en *permanent*. Vanaf Java 8 bestaat de *permanent generation* niet meer en is die vervangen door de *metaspace generation*.

De Java garbage collector verwijdert niet alleen het vrije geheugen. Er wordt ook gezorgd dat er geen gaten in het geheugen vallen door de geheugenplaatsen die niet verwijderd kunnen worden samen te zetten: *deletion with compacting*.

De *Serial Garbage collector* is de standaard garbage collector voor 32-bit machines in Java 5 en 6. Er wordt slechts 1 virtuele processor gebruikt voor de garbage collection. Gedurende het garbage collection proces kan de applicatie niet verder. Men noemt dat ook een “stop the world”-proces.

De *Parallel* of *Throughput Garbage Collector* maakt gebruik van meerdere threads voor de garbage collection van de young generation. De tenured en permanent generation worden standaard opgeruimd met een serial collector. Men kan een parallelle collector kiezen door bij het opstarten van de applicatie de optie -XX:+UseParallelGC te kiezen. Wanneer er echter maar één processor is, zal de Java Virtual Machine altijd de Serial Garbage Collector gebruiken. Wanneer we ook een parallel collector willen gebruiken voor de tenured en permanent generations, kunnen we dat via -XX:+UseParallelOldGC.

De *Concurrent Mark Sweep Collector* maakt voor de oude generatie gebruik van een garbage collection die tezamen met de applicatie loopt, vandaar de naam *concurrent*. De young generation wordt nog steeds afgehandeld met een parallel collection. Omdat de garbage collection tezamen met de applicatie loopt, worden objecten die nog in gebruik zijn niet verplaatst in het geheugen (*compacting*). De *Concurrent Mark Sweep Collector* wordt gebruikt met de schakeloptie -XX:+UseConcMarkSweepGC.

Vanaf Java 7 kunnen we ook gebruik maken van de G1 of *Garbage First Collector*. De bedoeling is dat de G1 collector op termijn de Concurrent Mark Sweep Collector zal vervangen. Het is ook een collector die parallel met de applicatie wordt uitgevoerd. Het grote verschil tussen de G1 collector en de vorige is dat bij de G1 collector het geheugen in blokken wordt verdeeld die even groot zijn. Sommige blokken zijn young, andere zijn tenured en nog andere (in Java 7) zijn permanent. Bij een garbage collection wordt elk blok behandeld door een eigen thread. Na de mark faze behandelt de garbage collection de blokken waar de meeste niet-meer-gebruikte geheugenplaatsen eerst. Om de G1 collector te gebruiken, is de schakeloptie -XX:+UseG1GC nodig.

Hoofdstuk 2. Nieuwe taalconstructies sinds Java 2

1. Enhanced for loop

In tegenstelling tot een aantal andere talen, kende Java voor Java 5 alleen maar de klassieke for-lus:

```
for (int i=0;i<lengte;i++){…}
```

Voor collections werd dikwijls de volgende constructie gebruikt:

```
for (Iterator<String> i = c.iterator(); i.hasNext();){
  String s = i.next();
  …
}
```

In Java 5 kan deze constructie vervangen worden door:

```
for (String s: c){
  …
}
```

De variabele s bevat in beide gevallen de huidige String. Voor deze aanpassing moest er alleen een wijziging gebeuren in de compiler. Achter de schermen wordt dit nog steeds vertaald naar een constructie met een Iterator. In de Java Virtual machine moet er niets veranderen.

De constructie kan gebruikt worden met arrays en met klassen die de interface java.lang.Iterable implementeren. In Java 5 is de interface java.lang.Collection gewijzigd zodat die ook de interface Iterable implementeert.

De enhanced for-loop heeft wel een belangrijke beperking ten opzichte van de Iterator. Aangezien een Iterator een remove()-methode heeft, kunnen we het huidige element verwijderen wanneer we een Iterator() gebruiken. Een enhanced for loop kan alleen gebruikt worden om alle elementen van een collectie te lezen, van voor naar achter.

2. Autoboxing

Boxing en unboxing zijn technieken om primitieve waarden om te zetten naar de overeenkomstige objecten en objecten om te zetten naar de primitieve waarde. Vroeger moest men de volgende code schrijven wanneer men een primitieve variabele wilde gebruiken op de plaats waar een object verwacht werd:

```
int getal = 42;
Integer integerGetal = new Integer(getal);
```

Tegenwoordig zorgt Java zelf voor die omzetting:

```
int getal = 42;
Integer integerGetal = getal;
```

Ook de omgekeerde code:

```
Integer integerGetal = new Integer(42);
int getal = integerGetal.intValue();
```

is niet meer nodig. We kunnen een Integer object direct toewijzen aan een primitieve int:

```
Integer integerGetal = new Integer(42);
int getal = integerGetal;
```

3. Typesafe Enums

Enums zijn een nieuw type in de Java-taal. Ze worden gebruikt wanneer een variabele maar een beperkt aantal waarden kan aannemen zoals bijvoorbeeld bij een toestand (*state*). De meeste eenvoudige declaratie van een enum-type ziet er als volgt uit:

```
public enum Seizoen{ZOMER, HERST, WINTER, LENTE}; //; niet noodzakelijk
```

In eerste instantie is dit te vergelijken met wat we vroeger moesten doen in Java wanneer we een beperkt aantal (integer) waarden nodig hadden:

```
public class Seizoen {
  private Seizoen(){}
  public static final Seizoen ZOMER  = new Seizoen();
  public static final Seizoen HERFST  = new Seizoen();
  public static final Seizoen WINTER  = new Seizoen();
  public static final Seizoen LENTE  = new Seizoen();
}
```

In beide gevallen kunnen we Seizoen gebruiken als returnwaarde of als argument voor een functie:

```
double geefGemiddeldeTemperatuur(Seizoen s){…}
```

We hebben echter nog meer mogelijkheden. We kunnen twee enums vergelijken:

```
if (s == Seizoen.ZOMER)(…}
```

En we kunnen ze ook overlopen met een enhanced for-loop en de namen afdrukken

```
for(Seizoen s: Seizoen.values()) {
    System.out.printf("%-8s: %d\n", s.toString(), s.ordinal());
}
```

De output van dit stukje code is:

```
ZOMER   : 0
HERFST  : 1
WINTER  : 2
LENTE   : 3
```

Hier zien we dat elke enum waarde automatisch ook een integerwaarde heeft gekregen.

Een enum kan ook gebruikt worden in een switch:

```
switch (s) {
    case WINTER:
        System.out.println("Koud en nat");
        break;
    case LENTE:
        System.out.println("Koud en nat");
        break;
    case ZOMER:
        System.out.println("Warm en nat");
        break;
    case HERFST:
        System.out.println("Warm en nat");
        break;
}
```

Tot nu toe hebben we in de voorbeelden alleen de “klassieke” mogelijkheden van het Enum type getoond. We kunnen nog een stapje verder gaan. Stel dat we een lijst met search engines hebben waaruit men kan kiezen. Aangezien de lijst beperkt is, zouden we een enum kunnen gebruiken. Bij elke searchengine willen we ook de URL bijhouden.

Dat kunnen we via enums als volgt doen:

```
enum SearchEngines{
    GOOGLE("http://www.google.be"),
    BING("http://www.bing.be"),
    ASK("http://nl.ask.com");
    private SearchEngines(String url){
        this._url = url;
    }
    private final String _url;
    public String getUrl(){
        return _url;
    }
}
```

We hebben aan de SearchEngines klasse een private constructor toegevoegd. We roepen die op door bij de waarden voor de enums de waarde voor de url mee te geven tussen ronde haakjes. Het is dus een constructor aanroep zonder *new*.

Vanaf nu kunnen we voor elk van de SearchEngine enums de url afdrukken:

```
    public static void main(String[] args) {
        for(SearchEngines se: SearchEngines.values()){
            System.out.printf("%s: %s\n", se.toString(), se.getUrl());
        }
    }
```

Het zou in principe zelfs mogelijk zijn dat we toelaten dat de url gewijzigd wordt (door final weg te laten en een setter te voorzien). Op dat moment gaan we voorbij aan de bedoeling van een enum. Enumwaarden zijn in principe constanten.

4. Varargs

Sommige methodes hebben een variabel aantal argumenten. Een voorbeeld hiervan is de printf()-methode van PrintStream. Voor Java 5 kon een methode alleen met een variabel aantal argumenten werken door een array door te geven.

De nieuwe varargs in Java 5 doen in feite hetzelfde, alleen heeft men hier speciale syntax voor voorzien. De printf() methode van PrintStream is gedeclareerd als

```
void printf(String format, Object… args);
```

De variabele args binnen de functie Is een array waarvan we het aantal ,elementen kunnen opvragen via de .length-property.

Op het eerste zicht zou dit problemen kunnen opleveren met method-overloading. Stel dat we de volgende twee functies hebben:

```
public void function1(String s1, String... s){
    System.out.println("function1 met varargs");
}
public void function1(String s1, String s2){
    System.out.println("function1 met 2 Strings");
}
```

Beide functies voldoen aan de functie-aanroep: function1(“a”, “b”). Maar Java lost dit op door de meest specifieke functie voorrang te geven. Aangezien er een functie is gedeclareerd met twee Strings zal die in dit geval gebruikt worden. De functie met varargs wordt gebruikt bij functie-aanroepen met 1, 3 of meer Strings.

5. Static import

Static import probeert een oplossing te zijn voor het “Constant interface antipattern”. Het idee achter het Contant interface antipattern is dat bepaalde constanten die we in verschillende klassen nodig hebben in een interface kunnen worden gezet. Als voorbeeld zouden we een aantal wiskundige constanten kunnen nemen:

```
interface MathConstants{
    final double PI = Math.PI;
    final double E = Math.E;
}
```

Wanneer we die constanten nodig hebben in een klasse, laten we die klasse die interface implementeren. De constanten PI en E zijn nu rechtstreeks aanspreekbaar in de klasse.

```
class Cirkel implements MathConstants{
    public double berekenOmtrek (){
        return 2 * PI * straal;
    }
}
```

Alhoewel dit patroon ervoor zorgt dat we de constante PI gemakkelijk kunnen gebruiken in de klasse Cirkel, misbruiken we hier het interface concept. Interfaces zijn types die een bepaald gedrag definiëren. De interface MathConstants is geen type en bepaalt geen gedrag.

Via static import kunnen we de constanten van een klasse importeren (net zoals we de types van een package kunnen importeren. Wanneer we de constante PI van Math zouden willen gebruiken zonder de namespace kunnen we het volgende import-statement gebruiken:

```
import static java.lang.Math.PI;
```

De klasse Cirkel moet de interface MathConstants niet meer implementeren. De constante PI kan zonder problemen gebruikt worden.

```
class Cirkel {
    public double berekenOmtrek (){
        return 2 * PI * straal;
    }
}
```

Anderzijds mag men het voordeel van namespaces niet uit het oog verliezen.

6. Generics

Generics zorgen ervoor dat de Java Collection-klassen werken zoals arrays. Wanneer we een array declareren, zal dat een array zijn die een bepaald type bevat: int[], String[], … Bij de collection klassen tot en met Java 1.4 waren dit altijd klassen die met Objecten werkten. En aangezien alles een Object is in Java, was het niet mogelijk om een ArrayList te gebruiken die alleen maar Strings kon bevatten.

Een mogelijke oplossing bestond erin dat men in een afgeleide klasse de add()-methodes overschreef zodat men bijvoorbeeld alleen maar Strings kon toevoegen aan de ArrayList:

```
class StringList extends ArrayList{
    @Override
    public boolean add(Object e) {
        if (!(e instanceof String )){
            throw new UnsupportedOperationException("only Strings");
        }
        return super.add(e);    }
```

```
}
```

Het probleem is dat dit runtime typechecking is. Indien mogelijk is compiletime typechecking te verkiezen.

Wanneer een klasse is gedefinieerd om met generics te werken zijn er één of meerdere placeholders voor de types voorzien. Bij de declaratie van een object van die klasse moeten we die placeholders vervangen door types, bijvoorbeeld:

```
ArrayList<String> stringlist = new ArrayList<String>();
```

In dit voorbeeld hebben we dus geen stringlist klasse gedefinieerd, maar een stringlist object. Het stringlist object is een ArrayList object dat alleen Strings kan bevatten. De klasse ArrayList is in Java als volgt gedefinieerd:

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable {
      Boolean add(E e) {…}
      …
}
```

De placeholder “E” wordt ingevuld bij de declaratie van het stringlist object als een String. Overal waar in de definitie van ArrayList een E staat zal die voor het stringlist object vervangen worden door “String”. Daarom werkt de Add()-methode alleen met Strings. In feite zorgen generics ervoor dat we een volledig nieuwe klasse hebben gedefinieerd.

We kunnen generics niet alleen gebruiken bij klassen, het lukt ook met methodes. In de Util klasse is de volgende methode gedefinieerd:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

De declaratie van de placeholders K en V gebeurt vlak voor het returntype van functie. We kunnen de methode als volgt gebruiken:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

Bij de declaratie van de generics klasse moet er gedefinieerd worden welke placeholders er zijn en waar ze in de klasse gebruikt worden. In het volgende voorbeeld wordt er een val gedefinieerd die objecten van een bepaalde soort kan “vangen”:

```
class Val<T> {
```

```
    private T gevangene ;
    public void vang(T gevangene) {
        this.gevangene = gevangene;
    }

    public T bevrijd() {
        T temp = gevangene;
        gevangene = null;
        return temp;
    }
}
```

We kunnen die val als volgt gebruiken(Muis is een klasse die al gedefinieerd is):

```
        Val<Muis> muizenval = new Val<Muis>();
        Muis mickey = new Muis();
        muizenval.vang(mickey);
        Muis mickey2 = muizenval.bevrijd();
```

De placeholder T wordt bij de declaratie van het *muizenval* object vervangen door *Muis*. De private member variabele zal van het type Muis zijn, de methode vang() werkt alleen met muizen en de methode bevrijd() geeft een muis terug.

Om het interessanter te maken, zouden we het voorbeeld kunnen uitbreiden. Stel dat de gevangene een geluid maakt bij de gevangenneming. Voor dat geluid passen we de klassen aan:

```
abstract class Dier{
    public abstract void maakGeluid();
}
class Muis extends Dier{

    @Override
    public void maakGeluid() {
        System.out.println("Piep");
    }

}
```

Wanneer we naar de definitie van de Val-klasse kijken, is er niets dat zegt dat T een dier is. De enige methodes die we op T kunnen gebruiken, zijn de methodes van de klasse Object. Om de methode maakGeluid() te kunnen gebruiken, moeten we ervoor zorgen dat de klasse Val alleen met objecten die afgeleid zijn van dier gebruikt kan worden. Daarvoor kunnen we *extends* gebruiken:

```
class Val<T extends Dier> {
    private T gevangene ;
    public void vang(T gevangene) {
        this.gevangene = gevangene;
        this.gevangene.maakGeluid();
    }

    public T bevrijd() {
```

```
        T temp = gevangene;
        gevangene = null;
        return temp;
    }
}
```

Bij de definitie van een Val-object zijn we nu verplicht om een afgeleide van de klasse Dier te gebruiken. Omdat we er nu zeker van zijn dat T altijd afgeleid zal zijn van Dier, mogen we de methode maakGeluid() gebruiken.

Generics en inheritance werken niet goed samen. De bovenstaande val kunnen we met eender welk Dier gebruiken. Maar is er een inheritance relatie tussen een val die zoogdieren vangt en een val die muizen vangt? We zullen dit eens nader bekijken in het geval van polymorfisme. Stel dat we de volgende functie zouden hebben:

```
public void printGetallen(ArrayList<Number> getallen){
   for(Number n: getallen){
      System.out.println(n);
   }
}
```

Number is de basisklasse voor alle getallen, dus ook voor Integers. We zouden kunnen denken dat we deze functie kunnen gebruiken met een list van Integers:

```
ArrayList<Integer> integerList = new ArrayList<Integer>();
printGetallen(integerList); //compileerfout
```

Dit geeft echter een compileerfout. Dat is misschien op het eerste zicht nogal raar want de volgende code geeft geen compileerfout:

```
public void printGetallen(Number[] getallen){
   for(Number n: getallen){
       System.out.println(n);
   }
}
Integer[] getallen = new Integer[10];
…
printGetallen(getallen);
```

Wanneer we dus hetzelfde doen met arrays (een array van Integers bewaren in een array van Numbers), protesteert de compiler niet. Dat heeft te maken met hoe de compiler omgaat met de type-informatie die we in de broncode meegeven. Voor arrays is die informatie over het type bekend in de bytecode. De Java Virtual Machine “weet” dat “getallen” een array van Integers is. Dat wil bijvoorbeeld zeggen dat de volgende code geen compileerfout zal geven, maar wel een runtime fout:

```
Integer[] getallen = new Integer[10];
Number[] numbers = getallen;
Numbers[0] = 3.14; //runtime fout
```

De Java Virtual Machine weet dat numbers eigenlijk een array van integers is. Wanneer we in het eerste element een Double proberen in te vullen (autoboxing), krijgen we een fout tijdens het uitvoeren. (ArrayStoreExeption)

Het verschil tussen Arrays en Generics is dat de compiler bij Generics de type informatie niet doorgeeft aan de bytecode. Een ArrayList<Number> wordt in bytecode een gewone ArrayList. (*type erasure*) waarbij er op sommige plaatsen een extra cast wordt toegevoegd. Maar de Java Virtual Machine zou bij de volgende "bytecode" geen runtime fout geven:

```
ArrayList getallen = new ArrayList();
ArrayList numbers = getallen;
numbers.add(3.14);
```

Wanneer deze code in de Java Virtual Machine zou kunnen worden uitgevoerd, verliezen we de typesafety. Vandaar dat de compiler het niet toelaat. En dit is een groot probleem wanneer we gebruik willen maken van polymorfisme.

De oplossing bestaat erin een onderscheid te maken tussen het opvragen van gegevens uit de ArrayList en het wegschrijven van gegevens in de ArrayList. We spreken hier ook over covariantie en contravariantie.

Covariantie wil zeggen dat we gegevens kunnen lezen van een supertype. Wanneer we de Number arraylist als volgt declareren:

```
ArrayList<? extends Number> numbers;
```

Dan kunnen we in de variabele een ArrayList van integers bewaren en gegevens opvragen:

```
ArrayList<Integer> getallen = new ArrayList<Integer>();
…
numbers = getallen;
Number n = numbers.get(0);
```

Aangezien in de ArrayList numbers alleen elementen mogen zitten die afgeleid zijn van Number (extends), mogen we het resultaat van de get()-functie bewaren in een Number.

Contravariantie wil zeggen dat we gegevens kunnen bewaren in een supertype. We declareren de ArrayList nu als volgt (super wil zeggen dat de klasse een parent moet zijn):

```
ArrayList<? super Integer> getallen;
```

In deze lijst kunnen we een Number arraylist bewaren:

```
ArrayList<Number> numbers = new ArrayList<Number>();
getallen = numbers;
getallen.add(3.14);
```

Aangezien 3.14 een Double, en dus ook een Number is, mogen we de waarde toevoegen aan de arraylist numbers.

7. Annotations

Annotations worden gebruikt om extra informatie toe te voegen aan Javacode. Tot en met Java 1.4 kon die extra informatie alleen maar worden toegevoegd via (XML) configuratiebestanden. In plaats van die XML-bestanden manueel in te vullen, werd er toen al gebruik gemaakt van preprocessors zoals XDoclet. Een voorbeeld van een XDoclet configuratie in een Java bestand zag er als volgt uit:

```
/**
 * This is the Account entity bean. It is an example of how to use the
 * EJBDoclet tags.
 *
 * @see Customer
 *
 * @ejb.bean
 *     name="bank/Account"
 *     type="CMP"
 *     jndi-name="ejb/bank/Account"
 *     local-jndi-name="ejb/bank/LocalAccount"
 *     primkey-field="id"
 *
 * @ejb.finder
 *     signature="java.util.Collection findAll()"
 *     unchecked="true"
 *
 * @ejb.transaction
 *     type="Required"
 *
 * @ejb.interface
 *     remote-class="test.interfaces.Account"
 *
 * @ejb.value-object
 *     match="*"
 *
 * @version 1.5
 */
```

De configuraties stonden allemaal in Java commentaar zodat de compiler er geen last van had. De XDoclet preprocessor las de commentaar van het Java bestand en zorgde ervoor dat op basis van de @xxx instellingen de juiste code voor de XML configuratiebestanden werd gegenereerd.

Vanaf Java 5 behoren annotations tot de Javataal en worden ze herkend door de compiler. Er worden drie soorten annotations onderscheiden die van elkaar verschillen op basis van de “retentionpolicy”. Dat wil zeggen dat ze op verschillende momenten beschikbaar zijn. Er zijn drie mogelijke retentionpolicies:

- Source: de annotation verdwijnt na het compileren en is niet beschikbaar in de class-file

- Class: de annotation wordt bewaard in het .class bestand maar is niet noodzakelijk beschikbaar in de Java Virtual Machine
- Runtime: de annotation is beschikbaar tijdens het uitvoeren van het programma.

Een voorbeeld van een Source annotation is @Override. We gebruiken deze annotation om aan te geven dat de methode een methode overschrijft in een parent klasse. Wanneer dat niet het geval is, zal de compiler een fout geven. Deze informatie is na compilatie niet meer nuttig vandaar dat ze niet wordt opgenomen in het .class bestand.

Runtime annotations worden vooral gebruikt door Java Frameworks. Java EE heeft configuratiebestanden bijna volledig vervangen door annotations zoals @Resource voor het injecteren van een DataSource of @WebServlet voor de definitie van een Servlet. De annotations worden tijdens het uitvoeren van het programma gelezen via reflection.

Annotations met een Class-retention policy worden minder gebruikt. Ze zijn bedoeld voor post-compile tools. Dat zijn tools die de bytecode verwerken na compilatie. Een voorbeeld hiervan zou een obfuscator tool kunnen zijn, die class-bestanden onleesbaar maakt voor disassemblers.

8. String in switch statements

Tot en met Java 6 kon men voor een switch-statement alleen char, byte, short, int of enum (vanaf Java 5) gebruiken (+ de overeenkomstige classes zoals Character). Vanaf Java 7 kan men ook een String gebruiken in een switch statement:

```
function String getActie(String weekdag){
  switch(weekdag) {
    "maandag":
    "dinsdag":
    "woensdag":
    "donderdag":
    "vrijdag":
      return "Aan het werk";
      break;
    "zaterdag":
    "zondag":
      return "Rust maar uit";
      break;
    default:
      throw new IllegalArgumentException("weekdag");
  }
}
```

Tijdens het uitvoeren wordt de hashcode van de string gebruikt voor de verschillende cases.

9. Resource protection block

Een resource protection block of *try-with-resources* is een alternatief voor een try..finally statement. Wanneer een klasse de interface java.lang.AutoCloseable implementeert (of java.lang.Closeable die afgeleid is van AutoCloseable), kan een try…finally statement vervangen worden door een *try-with-resources* statement:

```
try(Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten"){
…
}catch (SQLException ex) {
  Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
```

We kunnen het opvangen van de SQLException ook weglaten:

```
try(Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten"){
…
}
```

maar dan zullen we wel een *throws SQLException* moeten toevoegen aan de functie waarin de connectie wordt geopend.

Try-with-resources zal ervoor zorgen dat de connectie wordt gesloten bij het verlaten van het try-block. Dit gebeurt altijd, los van het feit of er nu een exception optreedt of niet, net zoals bij try…finally. In tegenstelling tot bij try…finally, zijn we bij try-with-resources beperkt in het soort objecten dat we kunnen gebruiken. Een object moet AutoCloseable of Closeable implementeren.

Wanneer we try…finally gebruiken, moeten we er rekening mee houden dat de close()-methode ook een SQLException kan opleveren. Die exception wordt meestal opgevangen via een extra try…catch in het finally-block:

```
try{
  …
}finally{
  try{
     conn.close();
  catch(SQLException ex){
     //do nothing
  }
}
```

Aangezien er geen goed scenario is voor foutafhandeling wanneer het sluiten van de connectie niet lukt, wordt hier dikwijls (hoogst uitzonderlijk) een leeg catch block gebruikt.

We vangen de exception van de close()-methode op omdat die de exception in het try-blok kan verbergen. Wanneer er een exception zou optreden zowel in het try-blok als in het finally-blok, zullen we alleen de exception van de close()-methode in het finally-blok zien.

In het try-with-resources voorbeeld kunnen er ook twee exceptions optreden, namelijk in het try-blok en bij het sluiten van de connectie. In dit geval zal de exception van het sluiten van de connectie verborgen worden. Aangezien de exception van het sluiten van de connectie minder belangrijk is, is dit veel logischer. We kunnen de exception van het sluiten van de connectie opvragen via de getSuppressed methode van Throwable.

10. Multiple exception handling

Soms willen we meerdere exceptions op een gelijkaardige manier afhandelen. Stel dat we een stuk code hebben dat zowel een IOException als een SQLException kan opleveren. We willen beide exceptions op dezelfde manier afhandelen, namelijk door ze te loggen en te rethrowen. Vóór Java 7 moesten we twee catch blokken schrijven met identieke code:

```
try{
…
}catch (IOException ex) {
     logger.log(ex);
     throw ex;
}catch (SQLException ex) {
     logger.log(ex);
     throw ex;
}
```

Vanaf Java 7 kunnen we beide Exceptions tegelijkertijd vermelden in het catch-block:

```
try{
…
}catch (IOException | SQLException ex) {
     logger.log(ex);
     throw ex;
}
```

We maken hier gebruik van een soort bitwise or om de Exceptions te scheiden van elkaar.

11. Diamond syntax

Een kleine maar handige verbetering bij het werken met generics in Java 7 is de diamond syntax. Wanneer we een List van Strings zouden declareren en initialiseren, moesten we dat vóór Java 7 als volgt doen:

```
List<String> lijst = new ArrayList<String>();
```

De generic parameter *String* moest natuurlijk hetzelfde zijn voor de interface List en de klasse ArrayList. Vanaf Java 7 kunnen we de generic parameter weglaten aan de rechterkant van het statement:

```
List<String> lijst = new ArrayList<>();
```

Java weet immers wat het type moet zijn.

12. Default methodes in interface

Interfaces kunnen geen code bevatten, want dan moeten het abstracte klassen zijn. Vanaf Java 8 kunnen we echter wel een default implementatie schrijven voor een functie. Neem bijvoorbeeld de interface Naam:

```
interface Naam{
    String getVoornaam();
```

```
    String getAchternaam();
    default String getNaam(){
        return getVoornaam() + " " + getAchternaam();
    }
}
```

Een naam is een combinatie van een voornaam en een achternaam. Met het *default* keyword kunnen we deze implementatie standaard voorzien.

Aangezien we meerdere interfaces kunnen implementeren en die interfaces (default) methodes kunnen hebben, zouden we dit kunnen zien als de introductie van *multiple inheritance* in Java. Maar aangezien de default methodes niet met *state* kunnen werken (er is geen state in een interface) is het toch niet helemaal hetzelfde.

Wanneer twee interfaces die allebei dezelfde default methodenaam gebruiken geïmplementeerd worden, zal de klasse de methode zelf moeten invullen. Er is geen mechanisme voorzien waardoor Java weet welke methode voorrang krijgt.

De voornaamste reden voor de introductie van default methodes in interfaces is de functional interface in lambda functies.

13. Lambda expressies

Lambda expressies zijn een feature waar al lang achter gevraagd wordt in Java. Ze introduceren de mogelijkheden van *functional programming* in Java. Dat wil zeggen dat ze onder meer toelaten dat functies gebruikt worden als argumenten van andere functies en dat we dus een bijkomend type hebben in de taal: naast klassen, interfaces en enums hebben we nu ook functies.

In tegenstelling tot bij .NET is men in Java altijd terughoudend geweest wanneer het ging over het introduceren van nieuwe mogelijkheden. Eén van de belangrijkste voorbeelden hiervan zijn generics. Alhoewel ze een nieuwe taalconstructie vormen, introduceerden ze niets nieuw op het niveau van de JVM.

Met lambda functies heeft men ook geen nieuwe constructie willen opzetten om functies als types te definiëren. In .NET heeft men dit bijvoorbeeld wel gedaan door de *delegates* te introduceren. In Java zal men interfaces gebruiken als functie types, meer bepaald de *functional interfaces.*

Een functional interface is een interface die maar één methode bevat, zoals bijvoorbeeld:

```
@FunctionalInterface
interface ControleerPersoon{
    boolean test(Persoon p);
}
```

De annotation @FunctionalInterface controleert of de interface maar één (abstracte) methode bevat.

We zouden die interface kunnen gebruiken bij de definitie van een methode die personen afdrukt, maar alleen de personen die voldoen aan de test mogen worden getoond:

```
static void printPersoon(List<Persoon> lijst, ControleerPersoon controle){
  for(Persoon p: lijst){
    if (controle.test(p)){
      System.out.println(p.getNaam());
    }
  }
}
```

Om deze methode aan te roepen, moeten we niet eerst een klasse definiëren die ControleerPersoon implementeert, vervolgens een object creëren en dat meegeven bij de aanroep. We kunnen in Java al lang hiervoor een anonieme klasse gebruiken:

```
printPersoon(lijst, new ControleerPersoon() {
  @Override
  public boolean test(Persoon p) {
    return p.getLeeftijd() >= 10;
  }
});
```

Dit is een constructie die we kunnen veranderen in een lambda expressie:

```
printPersoon(lijst, (p) -> p.getLeeftijd() >= 10);
```

Er is niets veranderd aan de definitie van printPersoon(). Die methode verwacht nog steeds een interface. Maar bij het aanroepen van de methode definiëren we als tweede argument een expressie die bestaat uit:

1. Eén of meerdere argumenten: (p).
2. Een pijl: ->
3. Een expression of een statement. In het voorbeeld gebruiken we een expressie. De waarde van de expressie wordt teruggegeven via een return. We hadden ook één of meerdere statements kunnen gebruiken. Dan hadden we ze tussen accolades moeten zetten.

Hetzelfde voorbeeld met meerdere statements:

```
printPersoon(lijst, (p) -> {int l = p.getLeeftijd(); return l >= 10;});
```

De code rechts van de pijl wordt dus uitgevoerd als body van de test()-methode. Omdat er maar één methode gedefinieerd is in de interface, “weet” Java welke methode er moet gebruikt worden.

Wanneer de ControleerPersoon interface meerdere methodes zou declareren, kunnen we geen lambda expressie meer gebruiken, tenzij het default methodes zijn.

14. Streams

Om te beginnen moet er een mogelijk misverstand uit de weg geruimd worden. *Streams* hebben niets te maken met I/O streams die we gebruiken om bestanden te lezen en schrijven of met *network sockets* te werken. Wanneer we de *stream API* gebruiken, zijn we meestal met collections bezig en gaat het eigenlijk over *functional programming*. Streams en Lambda expressions horen dus eigenlijk samen. We zullen beginnen met een voorbeeld:

```
List<String> lijst = Arrays.asList("Jan", "Piet", "Joris", "Corneel");
lijst.stream().filter(s -> s.startsWith("J"))
              .map(s -> s.toLowerCase())
              .sorted(Collections.reverseOrder())
              .forEach(System.out::println);
```

Het resultaat van dit stuk code is

```
joris
jan
```

De namen die beginnen met een 'J' worden omgezet naar kleine letters en omgekeerd gesorteerd afgedrukt. Dit is iets wat voor de komst van de streams API natuurlijk ook al mogelijk was. Maar de kracht van streams zit in het feit dat we relatief eenvoudige en gemakkelijk leesbare code kunnen schrijven om deze taken uit te voeren.

Om een collection te kunnen gebruiken met de Stream API moeten we eerst een omzetting uitvoeren naar een stream (via .stream()). De methodes die we vervolgens uitvoeren, worden onderverdeeld in *intermediate* en *terminal* operaties. Een *intermediate* operatie geeft een stream terug, zodat we meerdere *intermediate* operaties achter elkaar kunnen hangen (*method chaining*). De filter() functie is een voorbeeld van een *intermediate* operatie.

Een *terminal* operatie heeft als return type void. Zoals de naam al aangeeft is een terminal operatie de laatste operatie in de reeks. Sterker nog: een stream() keten *moet* eindigen met een terminal operatie, anders wordt ze niet uitgevoerd. In het volgende voorbeeld:

```
lijst.stream().filter( s -> {System.out.println(s);
                        return s.startsWith("C");
                  });
```

zetten we in de filter() functie een lambda expressie die bestaat uit twee statements. Het eerste statement drukt de string af, het tweede statement geeft de returnwaarde terug die verwacht wordt door de filter()-functie. De output van dit totale stream-statement is echter leeg. De filter()-functie wordt dus met andere woorden niet uitgevoerd. En de reden daarvoor is dat de stream-keten niet eindigt met een terminal operatie. Wanneer we een terminal opereatie toevoegen,

```
lijst.stream().filter( s -> {System.out.println(s);
                        return s.startsWith("C");
                  }).forEach(System.out::println);
```

krijgen we wel output:

```
Jan
Piet
Joris
Corneel
Corneel
```

Dat heeft te maken met het feit dat intermediate operaties *lazy* worden uitgevoerd. De *filter()* functie filtert niet echt, maar een creëert een stream die doorlopen zal worden wanneer de terminal operatie wordt uitgevoerd. De stream krijgt gegevens binnen en stuurt de gefilterde gegevens door naar de volgende stream in de reeks.

De meeste intermediate operaties zijn stateless. Een stateless intermediate operatie kan een element dat binnenkomt op de inputstream meteen doorsturen (of niet in het geval van een filter). Een stateful operatie zoals bijvoorbeeld sort, kan het resultaat pas doorsturen naar de outputstream wanneer alle elementen behandeld zijn.

We kunnen volgen in welke volgorde een *stream pipeline* doorlopen wordt, door de code licht aan te passen. We voegen ook een tweede filter toe om te kunnen laten zien wat er gebeurt na de sort()-methode:

```
lijst.stream().filter(s -> {
    System.out.printf("Filtering %s\n", s);
    return s.startsWith("J");
}).map(s -> {
    System.out.printf("Mapping %s\n", s);
    return s.toLowerCase();
}).sorted( (s1, s2) -> {
    System.out.printf("Sorteer %s en %s\n", s1, s2);
    return s1.compareTo(s2);
}).filter(s->{
    System.out.printf("Filtering2 %s\n", s);
    return true;
}).forEach(s -> System.out.printf("Foreach %s\n", s));
```

De output van deze code is:

```
Filtering Jan
Mapping Jan
Filtering Piet
Filtering Joris
Mapping Joris
Filtering Corneel
Sorteer joris en jan
Filtering2 jan
Foreach jan
Filtering2 joris
Foreach joris
```

Na de eerste filter (van Jan) wordt Jan verder in de pipeline gestuurd naar de mapping. Na de mapping stopt het doorsturen omdat de sort()-functie eerst alle elementen moet hebben

voordat er gesorteerd kan worden. Na de sortering worden de elementen echter weer één voor één doorgestuurd in de rest van de pipeline.

Bij het opstellen van een pipeline is de volgorde van de functies belangrijk. Doordat we de filter eerst hebben gezet, hadden mapping en sorting minder werk. Wanneer we eerst gesorteerd, dan gemapt en dan gefilterd hadden, zou het resultaat hetzelfde zijn, maar het zou meer tijd gekost hebben.

Omdat elementen één voor één worden doorgestuurd, is een element “weg” uit de pipeline wanneer het verwerkt is. We kunnen een stream dus niet hergebruiken. Wanneer we de volgende code proberen uit te voeren:

```
Stream<String> stroom = lijst.stream().filter(s -> s.startsWith("J"));
stroom.anyMatch(s -> s.endsWith("n"));
stroom.allMatch(s -> s.endsWith("n"));
```

krijgen we bij het uitvoeren een exception op het laatste statement:

```
Exception in thread "main" java.lang.IllegalStateException: stream has
already been operated upon or closed
```

Wanneer we in plaats van een stream een supplier van een stream gebruiken, kunnen we met de get() methode twee streams creëren die gebruikt kunnen worden door anyMatch en allMatch():

```
Supplier<Stream<String>> supplier = () -> lijst.stream().filter(s ->
s.startsWith("J"));
supplier.get().anyMatch(s -> s.endsWith("n"));
supplier.get().allMatch(s -> s.endsWith("n"));
```

Doordat voor sommige stream-operaties (zoals filter() of map()) de elementen afzonderlijk verwerkt worden en er geen beïnvloeding is tussen de elementen onderling, zouden de operaties ook parallel kunnen worden uitgevoerd. Dat kunnen we uitproberen door de parallel()-functie toe te voegen. In het volgende voorbeeld genereren we een IntStream van integers (1-45). Voor elk van die integers berekenen we het getal van Fibonacci en zetten we dat in een array:

```
IntStream getallen = IntStream.rangeClosed(1, 45);
long start = System.nanoTime();
int[] fibs = getallen.map(i -> fib(i)).toArray();
long einde = System.nanoTime();
System.out.printf("in serie: %d ms\n", (einde-start)/1000000);
getallen = IntStream.rangeClosed(1, 45);
start = System.nanoTime();
fibs = getallen.parallel().map(i -> fib(i)).toArray();
einde = System.nanoTime();
System.out.printf("in parallel: %d ms\n", (einde-start)/1000000);
```

Op een dual core systeem zou dit het volgende resultaat kunnen opleveren:

```
in serie: 8640 ms
in parallel: 7039 ms
```

Wanneer we code in parallel uitvoeren, zullen er meerdere threads gebruikt worden. Maar dat wil niet zeggen dat de code automatisch sneller zal zijn. In het bovenstaande voorbeeld, zal de seriële uitvoering sneller zijn wanneer we minder getallen berekenen. De generatie van de extra threads vraagt relatief veel tijd en zal pas een voordeel opleveren wanneer de uitvoering van één element in de pipeline relatief lang duurt.

15. Optional class

Stel dat we een klasse Persoon hebben. Een Persoon heeft een voornaam en een achternaam. De achternaam kan leeg zijn. We zullen dat als volgt implementeren:

```
class Persoon{
    private String voornaam;
    private String achternaam;

    public Persoon(String voornaam){
        this.voornaam = voornaam;
    }
    public Persoon(String voornaam, String achternaam) {
        this.voornaam = voornaam;
        this.achternaam = achternaam;
    }

//getters en setters

}
```

Stel dat we die klasse Persoon als volgt gebruiken:

```
Persoon p = new Persoon("Joske");
System.out.printf("%s %s\n", p.getVoornaam().toUpperCase(),
p.getAchternaam().toUpperCase());
```

Aangezien we geen achternaam mee hebben gegeven, zullen we een nullpointerexception krijgen. We hadden eerst moeten controleren of getAchternaam() een String teruggeeft. Er zijn twee problemen met deze code:

1. het vraagt een relatief uitgebreide constructie om te testen of een waarde null is;
2. de gebruiker van de code wist waarschijnlijk niet dat achternaam null kon zijn.

Bij deze twee problemen kan de Optional class helpen. We beginnen met achternaam als een Optional String te definiëren:

```
class Persoon{
    private String voornaam;
    private Optional<String> achternaam;

    public Persoon(String voornaam){
        this.voornaam = voornaam;
```

```
        this.achternaam = Optional.empty();
    }
   public Persoon(String voornaam, String achternaam) {
        this.voornaam = voornaam;
        this.achternaam = Optional.of(achternaam);
    }


    public Optional<String> getAchternaam() {
        return achternaam;
    }
    public void setAchternaam(String achternaam) {
        this.achternaam= Optional.of(achternaam);
    }
}
```

Door het feit dat getAchternaam() een Optional<String> teruggeeft, weet de gebruiker van de klasse dat de achternaam null kan zijn. Bijkomend kunnen we bij het gebruik van de achternaam "optional" de orElse(*default*) gebruiken om gemakkelijk een default waarde mee te geven:

```
System.out.printf("%s %s\n", p.getVoornaam().toUpperCase(),
p.getAchternaam().orElse("niemand").toUpperCase());
```

De beste oplossing zou natuurlijk zijn dat de Persoon klasse een default waarde zou voorzien voor de achternaam. Wanneer we klassen zo onafhankelijk mogelijk van elkaar willen maken, willen we niet dat een null pointer kan overspringen van de ene klasse naar de andere. Maar soms kunnen we niet anders. Wanneer een databank bijvoorbeeld null-waarden kan bevatten, zullen we die informatie in sommige gevallen toch moeten doorgeven aan de gebruiker.

Hoofdstuk 3. Werken met bestanden

1. Inleiding

Bestanden in Java zijn voorbeelden van I/O Streams. Een I/O stream mag niet verward worden met de nieuwe Streams in Java 8 voor functional style programming. I/O streams worden specifiek gebruikt voor I/O "apparaten" zoals bestanden, andere programma's, geheugen en netwerk sockets. Voor I/O streams maken we een onderscheid tussen InputStreams en OutputStreams. Van InputStreams kan alleen gelezen worden, naar OutputStreams kan alleen geschreven worden.

I/O streams steunen heel sterk op het Decorator *design pattern.* Ze kunnen bijvoorbeeld gecombineerd worden met een Buffer-klasse zodat het lezen of het schrijven kan gebufferd worden.

Naast de standaard I/O is er vanaf Java 1.4 ook de NIO (non-blocking I/O) die uitgebreid werd in Java 7 (NIO.2). In NIO is de buffer ingebouwd in klassen. Men zegt dat ook dat de standaard I/O met streams werkt en de NIO met buffers werkt. Verder is de standaard I/O een blocking mechanisme. Wanneer we lezen van een standaard I/O InputStream, moet het programma wachten tot de data ingelezen zijn. Bij het lezen van NIO buffer worden de gegevens gelezen die aanwezig zijn en vervolgens kan het programma (of de thread) verder gaan. Wanneer er nieuwe data voorhanden zijn, kunnen die worden ingelezen. Ondertussen kan het programma iets anders doen, bijvoorbeeld data van een andere stream lezen.

Omdat één thread op die manier tegelijkertijd verantwoordelijk kan zijn voor meerdere streams, beschikt NIO ook over selectors. Selectors zorgen ervoor dat de thread kan wisselen tussen verschillende input- of outputbuffers wanneer die klaar zijn om gebruikt te worden.

Aangezien de meeste streams gesloten moeten kunnen worden na gebruik, implementeren de meeste klassen Closeable of AutoCloseable. Het is misschien niet helemaal logisch, maar Closeable is een childinterface van AutoCloseable. Dat heeft voor een deel met historiek te maken. Closeable dateert van Java 5, terwijl AutoCloseable geïntroduceerd werd in Java 7 voor de ondersteuning van het try-with-resources statement. Elk Closeable object kan dus gebruikt worden met try-with-resources. Dat is natuurlijk handig voor de compatibiliteit. Maar de werking van Closeable was te strikt voor try-with-resources. De close()-methode die hoort bij Closeable is gedefinieerd as "throws IOException". Ze kan dus alleen gebruikt worden met I/O-klassen. Verder is het een zogenaamde idempotente operatie. Meerdere malen de methode aanroepen na elkaar heeft geen bijkomend effect. We kunnen dus bijvoorbeeld close() aanroepen op een object dat al gesloten is.

De close()-methode van Autoclose() is gedefinieerd als "throws Exception". Ze kan dus gebruikt worden met niet-I/O klassen. Verder hoeft ze niet idempotent te zijn (maar dit wordt wel aangeraden). Dat alles maakt dat try-with-resources minder strikte voorwaarden

oplegt aan de objecten dan de Closeable interface. En dus is het logisch dat Closeable een childinterface is van AutoCloseable.

2. Input- en OutputStreams

Zoals als de naam al impliceert worden outputstreams gebruikt om data te schrijven en inputstreams om data te lezen. De abstracte klasse java.io.OutputStream definieert de volgende methodes:

- close() (vanwege de Closeable interface): sluit de stream en geeft eventuele system resources (file descriptors, netwerk connecties, …) terug vrij.
- flush() (vanwege de Flushable interface): zorgt ervoor dat alle data die nog gebufferd is werkelijk wordt weggeschreven.
- write(byte[]), write(byte[], offset, length), write(int): schrijft gegevens naar de stream onder de vorm van bytes. Voor write(int) wordt er ook een byte weggeschreven, namelijk de 8 laagste bits. De reden waarom write(int) bestaat heeft waarschijnlijk te maken met het feit dat het moeilijker is om met losstaande bytes te werken in Java. Een getal is bijvoorbeeld altijd een int. De aanroep write(5) zou een fout geven wanneer write(byte) zou gedefinieerd zijn. Verder geeft de read() methode van InputStream een integer terug. Wanneer we die waarde zouden willen gebruiken in write(byte) zouden we een cast moeten toevoegen.

Alle methodes zijn gedeclareerd als throws IOException. De meest voorkomende reden van een exception zal zijn dat de stream al gesloten is (behalve in het geval van close()).

De meeste eenvoudige OutputStream in termen van implementatie is de ByteArrayOutputStream. Intern wordt er een byte[] bijgehouden. De grootte van de array wordt automatisch aangepast aan de inhoud van de ByteArray. In de Oracle Hotspot JVM start de array met lengte 32. Wanneer de array volledig gevuld is, wordt de grootte verdubbeld. De ByteArrayOutputStream kan dienst doen als buffer voor de andere outputstreams.

De java.io.InputStream definieert de volgende methodes:

- close() (vanwege de Closeable interface): sluit de stream en geeft eventuele System resources (file descriptors, netwerk connecties, …) terug vrij.
- long skip(int n): probeert n bytes over te slaan. De returnwaarde bevat het werkelijke aantal bytes dat overgeslagen is. Wanneer de stream geen seeking ondersteunt is dat een reden voor een IOException
- int available(): het aantal bytes dat gelezen (of overgeslagen) kan worden zonder dat het programma geblokkeerd wordt.
- mark(int readlimit) en reset(). De methode mark() markeert de huidige positie. De methode reset() keert terug naar de laatst gemarkeerde positie. De readlimit in mark bevat het aantal bytes dat gelezen mag worden zonder dat de markpositie ongeldig wordt. Het idee hierachter is dat na een reset de bytes vanaf de markpositie terug opnieuw gelezen moeten kunnen worden. De inputstream moet die dus bijhouden.

Wanneer we meer bytes dan readlimit inlezen tussen mark() en reset() wordt het niet meer gegarandeerd dat we via een reset() kunnen terugkeren.

- boolean markSupported(): ondersteunt de stream marks?
- abstract int read(), int read(byte[]), int read(byte[], offset, length). De twee laatste methodes bewaren de ingelezen gegevens in de byte array die we meegeven. De returnwaarde is het aantal ingelezen bytes. De bedoeling van de abstracte methode is dat er één byte wordt teruggegeven.

3. Tekstbestanden

De basisklasse om gegevens weg te schrijven naar een bestand is FileOutputStream.Het soort data dat wordt geschreven zijn bytes. In de praktijk zullen de data die we wegschreven echter geen bytes zjn, maar integers, floats, strings, objecten, … FileOutputStream is niet de meest efficiënte manier om gegevens weg te schrijven naar een bestand. Hiervoor kunnen we beter FileWriter gebruiken (om characters weg te schrijven) of PrintWriter (om strings weg te schrijven).

Aangezien FileOutputStream met files werkt, kunnen we eventueel de File Descriptor en het FileChannel opvragen. Die lowlevel structuren hebben we meestal niet zelf nodig.

Om tekstbestanden te schrijven zullen we meestal gebruik maken van een PrintWriter. Een PrintWriter is geen FileOutputStream, maar is afgeleid van Writer. Een Writer klasse schrijft karakters naar een outputstream.

Alles wat wordt weggeschreven via een PrintWriter wordt dus vertaald naar tekst. Wanneer we dus de integer 42 wegschrijven, worden de characters ‘4’ en ‘2’ geschreven. Wanneer we een object wegschrijven wordt achter de schermen String.ValueOf(object) uitgevoerd om het object eerst voor te stellen als een String.

Tekstbestanden worden gelezen via een BufferedReader die afgeleid is van de Reader klasse. Buiten het feit dat het lezen gebufferd wordt (via de buffer van de InputStream), is er een readLine() methode waarmee een volledige lijn tekst kan gelezen worden tot aan de End-of-Line indicator: \n, \r of \r\n.

De constructor van de BufferedReader gebruikt een FileReader die een child klasse is van een InputStreamReader. In het volgende voorbeeld wordt een tekstbestand geschreven en gelezen. Om ervoor te zorgen dat de bestanden correct worden gesloten na gebruik, worden twee try-with-resources statements gebruikt:

```
try (PrintWriter w = new PrintWriter("conscience.txt")) {
  w.print("In een dorp tusschen Hoogtraten en Calmpthout, in de antwerpsche
Kempen, ");
  w.println("woonde Peer Gansendock, de Baes uit de afspanning de St
Sebastiaen.");
}
try(BufferedReader r = new BufferedReader(new
FileReader("conscience.txt"))){
  String regel;
  while((regel = r.readLine()) != null){
```

```
    System.out.println(regel);
  }
}
```

We zien hier dat we voor het wegschrijven van de teksten naar de console gebruik maken van dezelfde println methode als bij het schrijven naar het bestand. Dat komt omdat System.out een PrintStream is, die net zoals PrintWriter afgeleid is van FilterOutputStream.

De kracht van I/O streams komt naar voor wanneer we gebruik willen maken van gecomprimeerde bestanden in het zipformaat. Een zipbestand is een verzameling van zipentries. Een zipentry object verwijst naar een plaats in het zipbestand waar een bestand begint. Wanneer we het vorige voorbeeld willen herhalen met zipbestanden, krijgen we de volgende code:

```
try (ZipOutputStream wzip = new ZipOutputStream(new
FileOutputStream("conscience.zip"))) {
  ZipEntry entry = new ZipEntry("conscience.txt");
  wzip.putNextEntry(entry);
  try (PrintWriter w = new PrintWriter(wzip)) {
    w.print("In een dorp tusschen Hoogtraten en Calmpthout, in de
antwerpsche Kempen, ");
    w.println("woonde Peer Gansendock, de Baes uit de afspanning de St
Sebastiaen.");
  }
}
try (ZipInputStream zis = new ZipInputStream(new
FileInputStream("conscience.zip"))) {
  zis.getNextEntry();
  try (BufferedReader r = new BufferedReader(new InputStreamReader(zis))) {
    String regel;
    while ((regel = r.readLine()) != null) {
      System.out.println(regel);
    }
  }
}
```

Bij het schrijven beginnen we met een nieuwe zipentry die we toevoegen aan het zipbestand. Vervolgens gebruiken we een PrintWriter als decorator om regels te kunnen wegschrijven naar het zip bestand.

Bij het lezen positioneren we ons op het gezipte tekstbestand. De constructor van BufferedReader verwacht een Reader en ZipInputStream is een stream. Vandaar dat we nog een extra klasse nodig hebben: InputStreamReader. Dat is een Reader die met een stream kan werken.

4. Binaire bestanden

Binaire bestanden zijn meestal opgebouwd volgens een recordstructuur. Om ze te kunnen lezen moeten we weten welke soort data op welke plaats in het bestand staat. Hiervoor maken we meestal gebruik van de DataInputStream en DataOutputStream bestanden. Dit zijn child klassen van de FilterInputStream en FilterOutputStream klassen. DataInputStream

en DataOutputStream kunnen primitieve types, arrays van bytes en (UTF) strings lezen en schrijven:

De klassen kunnen niet rechtstreeks met bestanden werken. In de constructor moeten we een Input/OutputStream meegeven. Om het lezen en schrijven efficiënt te laten verlopen, wordt meestal nog een buffered stream toegevoegd.

Bij het schrijven kunnen we gebruik maken van de specifieke writeXXX()-methodes voor de verschillende types. In dit voorbeeld willen we een integer en een String wegschrijven voor een aantal personen. Hiervoor hebben we verschillende mogelijkheden: writeChars(String), writeUTF(String) of voor elk karakter writeChar(char).

We zullen gebruik maken van writeChars(String) omdat we een vaste recordstructuur willen. writeUTF schrijft voor elke string de lengte weg. Deze methode is meer geschikt voor records van variabele lengte. De methode writeChar(char) is een meer primitieve versie van writeChars(String). Bij writeChar(char) moeten we zelf de string overlopen en één voor één de karakters wegschrijven. De methode writeChars(String) doet dat voor ons.

In de onderstaande code is elke record 104 bytes lang: 4 bytes voor de integer en 2x50 bytes voor de naam die maximaal 50 karakters lang mag zijn. Elke char is immers 2 bytes lang. Na het wegschrijven van de volledige naam, vullen we de overblijvende posities op met 0-waarden.

```
final int NAAM_MAX_LENGTH = 50;
Map<Integer, String> lijst = new HashMap<>();
lijst.put(1, "Karen");
lijst.put(2, "Kristel");
lijst.put(3, "Kathleen");
try (DataOutputStream out = new DataOutputStream(new
ufferedOutputStream(new FileOutputStream("k3.bin")))) {
   for (Map.Entry<Integer, String> entry : lijst.entrySet()) {
     int id = entry.getKey();
     String naam = entry.getValue();
     out.writeInt(id);
     out.writeChars(naam);
     for (int i = naam.length(); i < NAAM_MAX_LENGTH; i++) {
       out.writeChar(0);
     }
   }
}
```

Bij het lezen van een DataInputStream moeten we weten wanneer we aan het einde van het bestand zijn gekomen. Bij het lezen van een tekstbestand konden we zien dat we aan het einde waren gekomen doordat de readLine() functie <null> teruggaf. Bij een lezen van een DataInputStream wordt het einde van de stream gemeld via een EOFException. De exception opvangen is dus de standaard manier om te controleren wanneer we het einde van een bestand hebben bereikt.

```
try (DataInputStream in = new DataInputStream(new BufferedInputStream(new
FileInputStream("k3.bin")))) {
```

```
  int recordnr = 0;
  try {
    while (true) {
      int id = in.readInt();
      char[] charNaam = new char[NAAM_MAX_LENGTH];
      int charPositie = 0;
      char karakter = in.readChar();
      while (karakter != 0) {
        charNaam[charPositie] = karakter;
        charPositie++;
        karakter = in.readChar();
      }
      in.skipBytes((NAAM_MAX_LENGTH -charPositie - 1)*2);
      String naam = String.valueOf(charNaam, 0, charPositie);
      System.out.format("Record %d: %s (%d)\n", recordnr, naam, id);
      recordnr++;
    }
  } catch (EOFException ex) {
    System.out.println("Einde bestand");
  }
}
```

In dit codevoorbeeld kunnen we de integer inlezen via readInt. Om de String in te lezen, blijven we lezen tot we een 0-karakter tegenkomen. Om naar het begin van de volgende record te gaan, slaan we de overblijvende bytes over via skipBytes().

5. (Serializable) Objectbestanden

Binaire bestanden zijn interessant wanneer we gegevens met andere applicaties moeten uitwisselen. Dit komt minder en minder voor omdat de meeste bestandsuitwisseling tegenwoordig via XML of JSON gebeurt.

Maar wanneer we gegevens willen uitwisselen tussen twee Java applicaties, zullen dit meestal objecten zijn. Het is dus interessant wanneer we objecten rechtstreeks kunnen schrijven naar een bestand of lezen van een bestand. Dit is mogelijk wanneer de klassen waartoe de objecten behoren de Serializable interface implementeren. De Serializable interface is een *marker* interface. Dat wil zeggen dat er geen methodes zijn gedefinieerd.

In het volgende voorbeeld definiëren we een klasse Persoon die Serializable is:

```
class Persoon implements Serializable{
    private static final long serialVersionUID=1L;
    private int id;
    private String naam;
    public Persoon(int id, String naam) {
        this.id = id;
        this.naam = naam;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNaam() {
```

```
        return naam;
    }
    public void setNaam(String naam) {
        this.naam = naam;
    }
}
```

De private member variabele serialVersionUID is niet verplicht maar wordt sterk aangeraden. Het is een controlemechanisme bij het deserialiseren om te kunnen controleren of de juiste klasse aanwezig is op de machine waarop de deserialisatie wordt uitgevoerd.

We kunnen een lijst van personen op de volgende manier serialiseren:

```
List<Persoon> lijst = new ArrayList<>();
lijst.add(new Persoon(1, "Karen"));
lijst.add(new Persoon(2, "Kristel"));
lijst.add(new Persoon(3, "Kathleen"));
try(ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("lijst.obj"))){
  out.writeObject(lijst);
}
```

De writeObject()-methode van de ObjectOutputStream is dus voldoende om een object te serialiseren. Het opnieuw inlezen van de lijst is even eenvoudig:

```
List<Persoon> lijst2;
try(ObjectInputStream in = new ObjectInputStream(new
FileInputStream("lijst.obj"))){
  lijst2 = (List<Persoon>)in.readObject();
}
for(Persoon p: lijst2){
  System.out.printf("%d: %s\n", p.getId(), p.getNaam());
}
```

Bij het serialiseren wordt dus de volledige object-hiërarchie bewaard.

Een klasse die afgeleid wordt van een Serializable klasse is in principe ook Serializable. Het kan echter mislopen wanneer ze extra member variabelen declareert die niet Serializable zijn. Dat is namelijk de tweede voorwaarde (naast de Serializable interface implementeren): elk veld moet zelf Serializable zijn. Dat wil zeggen dat het ofwel een primitief type moet zijn, ofwel de Serializable interface moet implementeren.

Wanneer dat niet het geval is, is er dikwijls ook een mogelijkheid om zelf te bepalen hoe een veld serialiseerbaar is. We moeten ten eerste aanduiden dat het veld niet automatisch geserialiseerd moet worden door het *transient* te maken. Vervolgens voegen we twee extra private methoden toe readObject(ObjectInputStream) en writeObject(ObjectInputStream). Wanneer deze methodes bestaan zal de serialisatie engine ze gebruiken in het serialisatie proces. Stel dat we een klasse Adres hebben die niet serialiseerbaar is:

```
class Adres{
    private String straat;
    public String getStraat() {
        return straat;
    }
    public void setStraat(String straat) {
        this.straat = straat;
    }
}
```

Wanneer we nu een klasse PersoonAdres zouden afleiden van Persoon en hieraan een private member variabele *adres* toevoegen, is er een probleem. Omdat PersoonAdres afgeleid is van Persoon en dus Serializable implementeert maar een veld bevat dat niet serialiseerbaar is, zullen we een foutmelding krijgen tijdens het serialiseren: java.io.NotSerializableException: Adres.

Wanneer we het adres veld transient maken, krijgen we geen foutmelding meer. Maar de waarde van het adres veld zal niet geserialiseerd worden. In readObject en writeObject kunnen we definiëren hoe de serialisatie moet verlopen. In dit geval slaan we zelf de waarde van String “straat” op:

```
class PersoonAdres extends Persoon{
    private static final long serialVersionUID=2L;
    private transient Adres adres = new Adres();
    public PersoonAdres(int id, String naam, String straat) {
        super(id, naam);
        this.adres.setStraat(straat);
    }
    public void setStraat(String straat){
        adres.setStraat(straat);
    }
    public String getStraat(){
        return adres.getStraat();
    }
    private void writeObject(ObjectOutputStream out) throws IOException{
        out.defaultWriteObject();
        out.writeObject(adres.getStraat());
    }
    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException{
        in.defaultReadObject();
        adres.setStraat((String)in.readObject());
    }
}
```

De mehodes defaultWriteObject en defaultReadObject serialiseren het stuk dat wel serialiseerbaar is van PersoonAdres. De waarde van *straat* schrijven en lezen we zelf.

6. Nio.2

6. A. Path

Nio bestaat al sinds Java 1.4 maar vanaf Java 7 zijn er een aantal uitbreidingen bijgekomen. Die uitbreidingen vallen onder nio.2. Eén van die nieuwe nio.2 klassen is de Path-klasse. Via

deze klasse kunnen we een pad in het filestystem naar een bestand of een directory beschrijven.

Stel dat we een Path object willen krijgen dat verwijst naar de subdirectory src/niovoorbeelden. Met de Paths klasse kunnen we een Path object creëren dat als startpunt de huidige directory heeft:

```
Path p = Paths.get(“src”, “niovoorbeelden”);
```

In de get()-methode kunnen we de afzonderlijke onderdelen (directories) van het pad meegeven. Om een volledig absoluut path te verkrijgen, kunnen we de methode toAbsolutePath() gebruiken:

```
p = p.toAbsolutePath();
```

Wanneer we de parent directory van de huidige directory te pakken willen krijgen, kunnen we als volgt te werk gaan:

```
Path p = Paths.get(“..”);
p = p.toAbsolutePath();
p = p.normalize();
```

De functie normalize() vertaalt “..” naar *de hogergelegen directory*. Gecombineerd met het absolute path komen zo terecht in de parent directory van de huidige directory.

Belangrijk bij de functies toAbsolutePath() en normalize() is dat er niet naar het bestandssysteem wordt gekeken. Er wordt dus niet gecontroleerd of de directory of het bestand werkelijk bestaan. Om te laten controleren of het bestand werkelijk bestaat, kunnen we de functie toRealpath() gebruiken. Deze methode zet het pad om naar een absoluut pad, verwerkt “.” en “..” maar controleert ook of het bestand of de directory werkelijk bestaan. Wanneer dat niet het geval is, krijgen we een NoSuchFileException.

Vertrekkend vanaf een path kunnen we ook verder bouwen met de resolve() functie. Wanneer we vanuit het vorige path een nieuw path willen dat verwijst naar het bestand Main.java in die directory lukt dat op de volgende manier:

```
Path main =p.resolve(Paths.get("Main.java"));
```

Verder hebben we ook nog de mogelijkheid om het relatief path tussen twee paden te laten genereren:

```
Path relatief = main.relativize(p);
```

In dit voorbeeld zal dit “..” opleveren omdat het path “p” het parent path is van “main”.

6. B. Bestandsbeheer

Eénmaal dat we een Path hebben kunnen iets doen met dat bestand of die directory. We gebruiken daarvoor meestal de Files klasse. Hiermee kunnen we een bestand kopiëren:

```
Path main2 = p.resolve(Paths.get("Main.java.bak"));
Path newfile = Files.copy(main, main2);
```

We kunnen ook de inhoud opvragen van een directory via een DirectoryStream die we kunnen overlopen met een extended for:

```
try(DirectoryStream<Path> dirs = Files.newDirectoryStream(p)){
  for(Path entry:dirs){
    System.out.println(entry);
  }
}
```

Of een bestand verwijderen:

```
Files.delete(newfile);
```

We kunnen ook een stap verdergaan en bijvoorbeeld een lijst van bestanden opvragen die voldoen aan bepaalde filtervoorwaarden. Hierbij kunnen we gebruik maken van *globs* en het *visitor design pattern*.

Globs zijn bestandsnamen met *wildcards*. Heel wat mensen zijn bekend met '*' als wildcard in Windowssystemen. Zo duidt "*.txt" alle bestanden aan met als extensie "txt". De asterisk staat voor "hier mag eender wat staan, zelfs niets". De globs die gebruikt worden in Java zijn uitgebreider dan wat men kan gebruiken in Windows:

- * (asterisk): komt overeen met een willekeurig aantal (kan ook 0 zijn) willekeurige karakters. Opgelet: de grens van een directory (\ of /) wordt niet gezien als een willekeurig karakter. Men zegt ook wel eens dat '*' de grenzen van een directory niet overschrijdt.
- ** (twee asterisks): hetzelfde als *, maar nu worden de grenzen van een directory wel overschreden. Om het duidelijker te maken:
  *.txt vindt a.txt en b.txt, maar niet c:\users\joske\documenten\a.txt
  **.txt vindt c:\users\joske\documenten\a.txt wel
- ? (vraagteken): 1 willekeurig karakter
- {} (accolades): een mogelijkheid uit een reeks mogelijkheden. *.{docx, xlsx}: bestanden met de extensie .docx of .xslx
- [] (vierkante haakjes): een karakter uit een reeks: doc[xt], zowel 'docx' als 'doct' zijn juist. Het min-teken duidt een reeks aan: doc[0-9] is 'doc', gevolgd door een cijfer.

We kunnen een glob gebruiken in een PathMatcher. Die krijgen we via het filesystem:

```
PathMatcher matcher =
FileSystems.getDefault().getPathMatcher("glob:**java);
```

Een PathMatcher kunnen we gebruiken in een FileVisitor. Een FIleVisitor doorloopt recursief een bepaalde directorytree. Elke directory en elk bestand zal bezocht worden. Java heeft een speciale interface hiervoor: FIleVisitor. In deze interface zijn de volgende methodes gedefinieerd:

- preVisitDirectory(): uitgevoerd voordat een directory bezocht wordt. Hier kunnen we definiëren dat een directory niet bezocht moet worden.
- postVisitDirectory(): uitgevoerd nadat een directory bezocht wordt. Hier zouden we een actie kunnen uitvoeren om de directory te verwijderen (nadat we de bestanden erin verwijderd hebben)
- visitFile(): uitgevoerd wanneer een bestand bezocht wordt
- visitFileFailed(): uitgevoerd wanneer een bestand niet bezocht kan worden. Als argument krijgen we de IOException binnen die ervoor gezorgd heeft dat het bestand niet bezocht kon worden.

Elk van die methodes krijgt een verwijzing naar het bestand of de directory binnen en geeft een FileVisitResult terug:

- FileVisitResult.CONTINUE: ga verder
- FileVisitResult.SKIP_SIBLINGS: sla de items op hetzelfde niveau van deze file of deze directory over
- FileVisitResult.SKIP_SUBTREE: sla de children van deze directory over
- FileVisitResult.TERMINATE: stop

Om het gebruik hiervan te vergemakkelijken, is er ook een klasse gedefinieerd: SimpleFileVisitor. Die implementeert de interface FIleVisitor en geeft voor alle methodes CONTINUE terug. In visitFileFailed wordt de IOException gethrowd.

Een voorbeeld van een FileVisitor die op basis van een patroon een lijst van bestanden kan teruggeven, zou er als volgt kunnen uitzien:

```
class MyFileVisitor extends SimpleFileVisitor<Path> implements
Iterable<Path>{
    private final List<Path> paden = new ArrayList<>();
    PathMatcher matcher;

    public MyFileVisitor(String globPatroon) {
        matcher = FileSystems.getDefault().getPathMatcher("glob:" +
globPatroon);
    }
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
throws IOException {
        if (file !=null && matcher.matches(file)){
            paden.add(file);
        }
        return FileVisitResult.CONTINUE;
    }
    @Override
    public Iterator<Path> iterator() {
```

```
        return paden.iterator();
    }
}
```

We implementeren bijkomend de interface Iterable<Path> om ervoor te zorgen dat we de visitor achteraf in een extended for kunnen gebruiken. In visitFile controleren we of het huidige bestand voldoet aan het patroon. Wanneer dat het geval is, bewaren we het.

We kunnen de MyFileVisitor als volgt gebruiken om bijvoorbeeld alle .java bestanden op te vragen. De statische methode walkFileTree van Files gebruikt een root path en een visitor om de bestanden te overlopen.

```
Path srcroot = Paths.get("..", "..", "..").toAbsolutePath().normalize();
System.out.println(srcroot);
MyFileVisitor visitor = new MyFileVisitor("**src**java");
Files.walkFileTree(srcroot, visitor);
for(Path pad: visitor){
   System.out.println(pad);
}
```

Hoofdstuk 4. Werken met databanken

1. Inleiding

In dit hoofdstuk wordt behandeld hoe men via de Java Database Connectivity bibliotheek de gegevens aanspreekt in een relationele databank. Tegenwoordig wordt er dikwijls gebruik gemaakt van een Object-Relation Mapping framework (ORM) zoals Hibernate of EclipseLink of van bepaalde hulpklassen in het Spring framework. Hier bekijken we echter de basisfunctionaliteit van het werken met databanken, zonder extra hulpmiddelen buiten de mogelijkheden die standaard in Java 8 zitten.

De JDBC klassen worden gebruikt om:

1. een connectie te maken met een datasource;
2. opvragingen te doen en update statements uit te voeren;
3. de resultaten van een opvraging te verwerken.

JDBC bestaat uit vier onderdelen:

1. de DriverManager klasse om Java code te koppelen aan een JDBC driver. In een Java EE omgeving zal er meestal gebruik gemaakt worden van een DataSource object dat geregistreerd is via de Java Naming and Directory Interface;
2. de JDBC API met de klassen die nodig zijn om gegevens op te vragen en te wijzigen in een databank;
3. de JDBC testsuite om te testen of een JDBC driver geschikt is een programma
4. de JDBC-ODBC bridge die zorgt voor de connectie tussen JDBC en de meer algemene ODBC drivers.

Qua architectuur kan JDBC zowel in een *two-tier* als in een *three-tier* architectuur gebruikt worden. In een *two-tier* architectuur spreekt de client applicatie rechtstreeks met de databank. Die databank kan in het netwerk staan, maar ook op de client machine zelf. De JDBC driver moet dus aanwezig zijn op de client. In een *three-tier* architectuur communiceert de client met een *application server*. De *application server* maakt een connectie met de databank. Er moet dus geen JDBC code aanwezig zijn op de client. Alleen de application server moet een JDBC connectie kunnen maken. In een Java EE application server zal de databank geregistreerd zijn via de Java Naming and Directory Interface (JNDI).

Om een connectie te kunnen maken met een databank via JDBC moeten we een JDBC-driver hebben voor de databank. Er worden 4 soorten drivers onderscheiden:

1. Type1: JDBC-ODBC bridge driver. Kan gebruikt worden met elke databank waarvoor er een ODBC driver bestaat. JDBC maakt een connectie met de databank via de ODBC DSN. Dit is de minst performante driver. In de beginjaren van JDBC was dit de

belangrijkste manier om een connectie te maken met een databank omdat er nog maar weinig databanken waren die een JDBC driver hadden
2. Type2: JDBC-Native API. Is te vergelijken met de JDBC-ODBC bridge maar JDBC roept nu geen ODBC API aan, maar de specifieke databank API
3. Type3: JDBC-NET: Deze driver wordt gebruikt in een three tier architectuur waar de driver via netwerk sockets een connectie maakt met een type 1/2/4 JDBC driver op een application server
4. Type4: 100% pure Java: een JDBC driver die via netwerk sockets rechtstreeks een connectie maakt met de databank. Deze driver wordt voorzien door de databank fabrikant

JDBC 4 drivers hebben nog een aantal extra's ten opzichte van de oudere drivers:

- Autoloading van de driver klasse. Tot en met Java 5 moest men zelf zorgen dat de JDBC driver klasse geladen werd door Java via Class.forName("…"). Vanaf Java 6 kan er gebruik gemaakt worden van het Service Provider Mechanism van Java SE. In het jar-bestand van de driver moet er in de META-INF directory een bestand services/java.sql.Driver staan. In dit bestand wordt de naam van de driver vermeld, bijvoorbeeld:
  `org.apache.derby.jdbc.EmbeddedDriver`
- Connection Management: men kan vanaf nu ook een connectie maken met een Connection Factory. Naast gewone connecties kan men ook *pooled* connecties hebben. Het voordeel van *pooled* connecties is dat er een reeks open connection objecten voorzien worden. Wanneer een applicatie een connectie wil openen, wordt er een geopende connectie toegewezen aan de applicatie. Wanneer de applicatie de connectie "sluit", wordt ze niet echt gesloten, maar gaat ze terug naar de *pool*. Op die manier wordt er geen tijd verloren met het telkens openen en sluiten van de connectie.
- Ondersteuning voor RowID. In Oracle en DB2 kan de databank een unieke rowId genereren voor records die geen unieke id kolom hebben. De rowID kan gezien worden als het adres van de rij en is dus per definitie uniek.
- Uitbreiding van de SQLException klasse. Er worden twee nieuwe categorieën gedefinieerd voor subklassen van SQLException: SQLNonTransientExceptions die gebruikt worden wanneer het geen zin heeft het commando opnieuw uit te voeren zonder iets te veranderen en SQLTransientExceptions die aanduiden dat er tijdelijk iets misgelopen is, maar dat het commando opnieuw uitvoeren zou kunnen lukken.
- Chained exceptions voor SQLException gecombineerd met een extended for loop. Sommige databanken kunnen meer dan één fout terugsturen. De verschillende exceptions zullen in een exception chain geplaatst worden. Aangezien de SQLException klasse de Iterable interface implementeert, kunnen de verschillende excepties worden overlopen met een extended for loop.
- Ondersteuning voor National Character sets: de types NCHAR, NVARCHAR, NTEXT, NCLOB,… worden ondersteund
- De DataSet klasse. Wordt gebruikt om een databank zowel in connected mode als in disconnected mode aan te spreken. In connected mode werkt de DataSet klasse zoals een Resultset, in disconnected mode functioneert ze als een CachedRowSet.

Aangezien de klasse de List interface implementeert kunnen de rijen gemakkelijk overlopen worden.

2. Een connectie met een databank

Om een connectie te maken met een databank hebben we een driver nodig, een connectiestring die definieert hoe de connectie gemaakt moet worden en een Connection object. De driver moet aanwezig zijn in het ClassPath, maar wanneer het een JDBC 4 driver is, moeten we de driver niet meer zelf laden (via Class.forName). Java kan de driver namelijk zelf vinden.

Om de connectie op te vragen, maken we gebruik van de DriverManager klasse. De methode getConnection() geeft een geopende connectie terug. Om te weten welke driver er gebruikt moet worden en met welke databank er een connectie moet gemaakt worden, moeten we een connectiestring als argument meegeven aan de methode getConnection. Voorbeelden van connectiestrings zijn:

```
jdbc:mysql://localhost:3306/
jdbc:derby://localhost:1527/planten
```

Een connectiestring bestaat altijd uit *jdbc* gevolgd door de naam van de databank. Op basis van die naam zal de DriverManager op zoek gaan naar de juiste driver. Daarna volgt de URI van de databank. In dit geval staan de databanken op dezelfde machine als de applicatie (localhost). De poort hangt samen met het soort databank. Voor MySQL is dit meestal 3306. Normaal gesprokenl vermeldt men in de connectiestring ook de naam van de databank. In het voorbeeld van de JavaDB (Derby) databank is dit “planten”.

De getConnection() functie kan een SQLException opleveren wanneer de databank bijvoorbeeld niet geopend kan worden. Dit is een checked exception in Java. Dat wil zeggen dat we ze ofwel moeten opvangen, ofwel moeten we ze declareren. Belangrijk bij een Connection object is ook dat we niet mogen vergeten om ze te sluiten. We kunnen hiervoor een try…finally constructie gebruiken, maar vanaf Java 7 is try-with-resources meer aangewezen:

```
try(Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", “app”,
“app”){
…
}catch (SQLException ex) {
  Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
```

De gebruikersnaam en het paswoord voor de databank zijn in dit voorbeeld “app” en “app”. Vandaar dat we die ook meegeven in de getConnection()-functie. We hadden ook een Properties object kunnen meegeven met die informatie.

3. Datasources

In een application server zullen de connecties beheerd worden door de application server, niet door de applicatie. In dat geval zullen we gebruik maken van een DataSource. Een DataSource laat toe om gebruik te maken van *connection pooling*. Bij connection pooling wordt er een pool van connecties voorzien. Wanneer de applicatie een connectie nodig heeft, wordt één connectie uit de pool toegewezen aan de applicatie. Wanneer de applicatie de connectie "sluit", wordt ze teruggegeven aan de pool.

We zullen ons hier beperken tot hoe een application server een Datasource zal implementeren. Een datasource wordt door een application server geregistreerd in een name service: de Java Naming and Directory Interface (JNDI). Dat is niet de verantwoordelijkheid van de ontwikkelaar van de applicatie, maar in dit geval zullen we dit zelf doen om te laten zien wat er achter de schermen gebeurt. Een voorbeeld met connection pooling komt niet aan bod in deze syllabus. Voor de implementatie hiervan verwijzen we naar het Apache DBCP project. (Apache commons, 2015)

Aangezien we geen application server hebben, hebben we ook geen JNDI service. We zullen die dus zelf moeten voorzien. Een JNDI service registreert objecten onder een bepaalde naam en laat toe om die objecten op te zoeken via de naam. We zullen hiervoor een HashMap gebruiken. We implementeren hier een heel eenvoudige JNDI service die in de praktijk niet zal werken. De namen in JNDI vormen immers een hiërarchie zoals de package structuur van Java. In dit voorbeeld zullen we een hiërarchie hebben van 1 niveau.

Een klasse die als JNDI service dienst moet doen, moet de Context interface implementeren:

```
public class MyInitialContext implements Context {
    private final Map<String, Object> namen = new HashMap<>();

    @Override
    public Object lookup(String name) throws NamingException {
        Object o = namen.get(name);
        if (o == null) throw new NamingException("De naam " + name + "is
niet gevonden");
        return o;
    }
    @Override
    public void bind(String name, Object obj) throws NamingException {
        try{
           namen.put(name, obj);
        }catch(NullPointerException | IllegalArgumentException ex){
            throw new NamingException("Naam kan niet geregistreerd
worden");
        }
    }
…
}
```

We hebben hier alleen de twee methodes geïmplementeerd die belangrijk zijn voor ons:

1. de mogelijkheid om een object te registreren onder een naam (bind);
2. de mogelijkheid om een object op te vragen op basis van een naam (lookup).

Een tweede klasse die moet voorzien worden door de application server is een ContextFactory: een object dat een context kan teruggeven:

```
public class MyInitialContextFactory implements InitialContextFactory{

    @Override
    public Context getInitialContext(Hashtable<?, ?> environment) throws
NamingException {
        return new MyInitialContext();
    }
}
```

De enige methode die hier voorzien moet worden voor de InitialContextFactory interface is getInitialContext(). We geven hier het InitialContext object terug. De “environment”-parameter wordt normaal doorgegeven aan het InitialContext object. In ons geval doen we dat niet.

Om de initialcontext te gebruiken, moeten we aangeven welke klasse gebruikt moet worden als factory. We doen dat via een Properties object. Vervolgens kunnen we een InitialContext object maken en een DataSource object registreren:

```
Properties prop = new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY,
"jdbcapp.MyInitialContextFactory");
Context ctx = new InitialContext(prop);
DataSource ds = createDataSource();
ctx.bind("jdbc/planten", ds);
```

De registratie van het Datasource object gebeurt normaal ook door de application server, net zoals het creëren van het object:

```
    private static ClientDataSource40 createDataSource() {
        ClientDataSource40 ds = new ClientDataSource40();
        ds.setServerName("localhost");
        ds.setPortNumber(1527);
        ds.setDatabaseName("planten");
        ds.setUser("app");
        ds.setPassword("app");
        return ds;
    }
```

Nu is al het voorbereidend werk achter de rug en kunnen we het datasource object gebruiken:

```
DataSource dsc = (DataSource) ctx.lookup("jdbc/planten");
try (Connection conn = dsc.getConnection();
   …
}
```

Het grote verschil met de DriverManager is dat we hier geen parameters meer moeten meegeven om de connectie. Dat is al gebeurd, meestal door de application server.

4. Statements gebruiken

Eénmaal dat we een connectie hebben, kunnen we SQL statements doorsturen naar de databank. Hiervoor kunnen we het Statement object gebruiken. In het volgende voorbeeld maken we een tabel aan in een databank:

```
try (Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", "app",
"app");
   Statement s = conn.createStatement()) {
   s.executeUpdate("CREATE TABLE klanten (id int primary key, naam
varchar(30))");
}
```

Net zoals het Connection object moet het Statement object ook afgesloten worden. Daarom nemen we het op in het try-with-resources blok. We kunnen meerdere AutoCLoseable objecten combineren in één try-with-resources-blok door ze te scheiden door een puntkomma.

Het Statement object dat we hier gebruikt hebben is geschikt wanneer een command maar éénmaal moet uitgevoerd worden. Voor het CREATE TABLE commando dat we hier als voorbeeld hebben gebruikt, is dat het geval. Maar wanneer we vervolgens meerdere records willen toevoegen, zullen we meerdere malen een INSERT-commando moeten voeren. Die verschillende INSERT commando's zullen alleen verschillen van elkaar door de waarden die voor elke record anders zullen zijn. In SQL kunnen we parameters gebruiken. Het SQL commando wordt een soort van template met *placeholders*. Op de plaats van de *placeholders* kunnen we de waarden zetten die ingevuld moeten worden.

Dit heeft twee voordelen:

1. Heel wat databanken kunnen SQL commando's met parameters compileren zodat de gecompileerde versie meerdere malen kan gebruikt worden. Wanneer we geen parameters gebruiken, zal de databank voor elke uitvoering het commando opnieuw moeten compileren.
2. Het alternatief voor parameters is dat we SQL commando's samenstellen door strings te concateneren: "INSERT INTO klanten (id, naam) VALUES (" + id + ", '" + naam + "')". Buiten het feit dat het niet altijd eenvoudig is om dit soort van strings samen te stellen, zet dit ook de poort wagenwijd open voor SQL Injection aanvallen.

Wanneer we een statement meerdere malen moeten uitvoeren, kunnen we gebruik maken van een PreparedStatement object. De SQL string van een PreparedStatement object zal vraagtekens bevatten op de plaats waar de waarden moeten worden ingevuld:

```
try (Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", "app",
"app");
   PreparedStatement ps = conn.prepareStatement("INSERT INTO klanten (id,
naam) VALUES (?, ?)")) {
     for (Map.Entry<Integer, String> klant : klanten.entrySet()) {
       ps.setInt(1, klant.getKey());
```

```
            ps.setString(2, klant.getValue());
            ps.executeUpdate();
        }
    }
```

In dit voorbeeld is klanten een HashMap met id's en namen. Voordat we het preparedstatement kunnen uitvoeren, moeten we eerst waarden invullen voor de placeholders. Hiervoor worden de setXXX()-methodes gebruikt.

5. Resultset

De meest voor de hand liggende manier om gegevens op te vragen uit de databank is via een ResultSet. Wanneer we voor een Statement (of een PreparedStatement) de methode executeQuery uitvoeren (met een SELECT-statement), krijgen we een resultSet terug:

```
try (Connection conn = dsc.getConnection();
     Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery("select * from klanten")) {
     while (rs.next()) {
        int id = rs.getInt("id");
        String naam = rs.getString("naam");
        System.out.printf("%d: %s\n", id, naam);
     }
}
```

De getXXX()-methodes van de resultset gebruiken we om de waarden van de huidige record op te vragen. Als argument geven we een kolomnummer mee (vertrekkend van 1) of de naam van de kolom.

Een ResultSet kan altijd van voor naar achter overlopen worden met behulp van de next()-mehode. Dat is de mogelijkheid TYPE_FORWARD_ONLY. We kunnen de mogelijkheden van de resultset opvragen via de database metadata:

```
DatabaseMetaData meta = conn.getMetaData();
if (meta.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY)){
   System.out.println("Support voor TYPE_FORWARD_ONLY");
}
if (meta.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)){
   System.out.println("Support voor TYPE_SCROLL_INSENSITIVE");
}
if (meta.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE)){
   System.out.println("Support voor TYPE_SCROLL_SENSITIVE");
}
```

De test op TYPE_FORWARD_ONLY is overbodig omdat elke databank resultsets van dit type ondersteunt. Een TYPE_SCROLL_INSENSITIVE ResultSet laat toe dat we ook terugkeren naar een vorige rij via de previous()-methode. We kunnen ook naar een absolute positie gaan met behulp van de methode absolute(int positie). Een positieve waarde verwijst naar een rijnummer waarbij de eerste rij de positie 1 krijgt. Een negatieve waarde begint te tellen vanaf de laatste rij: absolute(-1) zal de resultset op de laatste rij zetten. ResultSet.absolute(0) zet de resultset voor de eerste rij. We zullen eerst next() moeten aanroepen voordat we de gegevens van een rij kunnen opvragen.

Een TYPE_SCROLL_INSENSITIVE ResultSet zal een snapshot nemen van de data op het moment dat het statement wordt uitgevoerd. Wanneer de onderliggende data in de databank veranderen, zullen we die wijzigingen niet zien in de ResultSet. Dat is wel het geval voor een ResultSet die TYPE_SCROLL_SENSITIVE ondersteunt.

De concurrency van een resultset bepaalt of we ook aanpassingen kunnen doen in de databank via de resultset. De default toestand is CONCUR_READ_ONLY. Dat wil zeggen dat de databank niet kan worden aangepast via de resultset. CONCUR_UPDATABLE houdt in dat we de gegevens in de databank kunnen wijzigen via de resultset. We kunnen de concurrency mogelijkheden opvragen via supportsResultSetConcurrency(type, concurrency).

In het volgende voorbeeld willen we een ResultSet die gebruikt kan worden voor aanpassingen en ook absoluut positioneren ondersteunt. Vandaar dat we dit meegeven bij het aanroepen van de prepareStatement methode. Via de updateXXX() methodes kunnen we waarden van een rij aanpassen en via updateRow kunnen we de wijzigingen wegschrijven naar de databank:

```
try (Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", "app",
"app");
   PreparedStatement ps = conn.prepareStatement("SELECT * FROM klanten
WHERE ID=?",ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);) {
      ps.setInt(1, 2);
      try (ResultSet rs = ps.executeQuery()) {
          rs.absolute(1);
          rs.updateString("naam", "Josje");
         rs.updateRow();
      }
   }
```

We kunnen een resultset ook gebruiken wanneer we nieuwe records willen toevoegen:

```
try (Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", "app",
"app");
  Statement st = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
  ResultSet rs = st.executeQuery("SELECT * FROM klanten")) {
    rs.moveToInsertRow();
    rs.updateInt("id", 4);
    rs.updateString("naam", "Hanne");
    rs.insertRow();
    rs.beforeFirst();
 }
```

Voordat we de gegevens van de nieuwe record kunnen toevoegen, moeten we ons eerst positioneren op een speciale rij, namelijk de insertrij. Dat gebeurt via de methode moveToInsertRow(). Eénmaal op die rij kunnen we de nieuwe gegevens invullen zoals we dat ook bij een wijziging zouden doen. Om de nieuwe rij weg te schrijven in de databank, gebruiken we de methode insertRow(). Wanneer we de ResultSet achteraf nog willen

gebruiken is het een goed idee om niet op de insertrow te blijven staan. Vandaar dat we ons voor de eerste rij positioneren. De methode beforeFirst() doet eigenlijk hetzelfde als absolute(0).

6. Batch updates

Wanneer we meerdere records moeten wijzigen of toevoegen, bijvoorbeeld bij het importeren van gegevens, is het dikwijls efficiënter om alle wijzigingen in één groep door te sturen. In het geval van een preparedstatement is dit zeker het geval omdat het statement maar eenmaal wordt doorgestuurd, tezamen met de gegevens voor alle records.

Om meerdere statements in éénmaal door te zenden, kunnen we een batch update gebruiken. Om het toevoegen van gegevens nog te versnellen, kunnen we alle updates ook in één transactie zetten. Een JDBC driver staat standaard in autocommit mode. Dat wil zeggen dat voor het uitvoeren van elk statement een transactie wordt gestart en na het uitvoeren van het statement de transactie wordt gecommit. Het quasi continue starten en committen van een transactie bij elke wijziging vertraagt het uitvoeren van de statements. Het werkt sneller wanneer we bij het begin van de reeks statements een transactie starten en de transactie committen wanneer het laatste statement is uitgevoerd. Het nadeel hierbij kan zijn dat alle statements ongedaan gemaakt worden wanneer er één statement mislukt. Maar dit zou natuurlijk ook een voordeel kunnen zijn wanneer we willen dat er maar twee mogelijke einduitkomsten zijn: ofwel is alles toegevoegd aan de databank, ofwel is er niets toegevoegd aan de databank.

Om de connectie uit de standaard autocommit mode te halen, gebruiken we de methode setAutoCommit(false) op de connectie.

```
 try (Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", "app",
"app");
     PreparedStatement ps = conn.prepareStatement("INSERT INTO klanten (ID,
naam) VALUES(?,?)")){
     conn.setAutoCommit(false);
     ps.setInt(1, 5);
     ps.setString(2, "Marthe");
     ps.addBatch();
     ps.setInt(1, 6);
     ps.setString(2, "Klaasje");
     ps.addBatch();
     int[] updateCounts = ps.executeBatch();
     for(int i:updateCounts){
        System.out.println("Batch update: " + i);
     }
     conn.commit();
 }
```

Door na het instellen van de parameters de methode addBatch() op te roepen, voegen we de parameters toe aan de batch. In dit voorbeeld zullen er twee records worden toegevoegd aan de databank. De array updateCount die we terugkrijgen van executeBatch geeft voor elk statement terug hoeveel records er zijn gewijzigd. De gegevens worden pas definitief weggeschreven in de databank na het uitvoeren van de commit() methode.

We kunnen zien wat er gebeurt wanneer één van de updates niet lukt door een dubbele id in te geven. Wanneer er iets misloopt, kunnen we best ook een rollback uitvoeren. Wanneer we dat niet doen krijgen we de SQLException:

```
java.sql.SQLException: Cannot close a connection while a transaction is
still active.
```

Maar dat hangt natuurlijk af van de specifieke databank. De code zou er dan als volgt kunnen uitzien:

```
try (Connection conn =
DriverManager.getConnection("jdbc:derby://localhost:1527/planten", "app",
"app");
   PreparedStatement ps = conn.prepareStatement("INSERT INTO klanten (ID,
naam) VALUES(?,?)")){
      conn.setAutoCommit(false);
      ps.setInt(1, 5);
      ps.setString(2, "Marthe");
      ps.addBatch();
      ps.setInt(1, 5);
      ps.setString(2, "Klaasje");
      ps.addBatch();
      try{
         int[] updateCounts = ps.executeBatch();
         for(int i:updateCounts){
            System.out.println("Batch update: " + i);
         }
         conn.commit();
       }catch(BatchUpdateException ex){
         conn.rollback();
         for(int updateCount: ex.getUpdateCounts()){
           System.out.println("Update count " + updateCount);
         }
       }
}
```

De BatchUpdateException die we opvangen bevat voor elk succesvol uitgevoerd commando het aantal gewijzigde records in de array die we terugkrijgen van getUpdateCounts. Wanneer een statement mislukt is (zoals hier het geval is wegens de dubbele id), is de waarde gelijk aan Statement.EXECUTE_FAILED (of -3).

7. Rowsets

7. A. Inleiding

Rowsets vormen een wrapper rond een ResultSet en voegen dus extra mogelijkheden toe. Er zijn 5 RowSet interfaces gedefinieerd binnen de standaard Java omgeving:

- JdbcRowSet: voegt scroll- en update-mogelijkheden toe aan een ResultSet. Dit heeft natuurlijk alleen maar zin wanneer de ResultSet dit niet zelf ondersteunt. JdbcRowSet is tegenwoordig niet meer zo belangrijk.
- CachedRowSet: de basis van de disconnected rowsets. Een ResultSet (en een JdbcRowSet) kan alleen gebruikt worden wanneer er een connectie is met een

databank. De data in een CachedRowSet blijft toegankelijk wanneer de connectie gesloten is. De data kan niet alleen gelezen worden zonder databank connectie, maar ook gewijzigd. Nadat alle wijzigingen zijn gebeurd in de CachedRowSet kan er opnieuw een connectie worden gemaakt met een databank om de wijzigingen op te slaan. Aangezien er dikwijls een bepaalde tijd ligt tussen het inlezen van de data en het wegschrijven van de wijzigingen, is er ook een mechanisme voorzien om te controleren of de data in de databank ondertussen niet veranderd is.

- WebRowSet: een CachedRowSet die geserialiseerd kan worden als XML document (en ook kan ingelezen worden van een XML document)
- JoinRowSet: is een WebRowSet die zonder connectie met de databank een SQL JOIN kan implementeren
- FIlteredRowSet: een WebRowSet waarop filters gezet kunnen worden, zonder dat men een connectie met de databank moet hebben of een query moet gebruiken.

Om een Rowset te creëren, kunnen we vanaf Java 7 gebruik maken van een RowSetFactory. Een alternatief is dat we gebruik maken van de constructor van de rowset, maar de implementaties van de verschillende RowSet interfaces behoren niet tot de Java standaard en de namen zouden in de toekomst kunnen veranderen. Door gebruik te maken van een factory object, koppelen we de applicatie niet aan een specifieke RowSet implementatie klasse. Een Rowsetfactory object maken we als volgt aan:

```
RowSetFactory factory = RowSetProvider.newFactory();
```

- Welke factory er nu specifiek wordt aangemaakt, wordt achter de schermen bepaald. We zouden kunnen stellen dat er soort van *dependency injection* mechanisme wordt gebruikt. Wanneer de system property javax.sql.rowset.RowSetFactory bestaat zal de klassenaaam die daaraan gekoppeld is gebruikt worden om een rowsetfactory te creëren.
- In een volgende stap wordt er gezocht naar een implementatie via de Service Loader API. Een klasse (jarbestand) met een bestand META-INF/services/javax.sql.rowset.RowSetFactory en als inhoud de volledige naam van de klasse zal gebruikt worden als factory.
- Tenslotte is er nog de default implementatie van het platform.

In deze tekst behandelen we alleen de CachedRowSet en de FilteredRowSet. Voor de mogelijkheden van de andere rowsets verwijzen we naar de documentatie van Oracle.

7. B. CachedRowSet

Een rowsetfactory kan gebruikt worden om alle soorten rowsets aan te maken, dus ook de CachedRowSet. Net zoals bij een ResultSet of een JdbcRowSet kunnen we gegevens inladen uit een databank. We hebben daarvoor een Connection en een SQL-commando nodig. Het verschil tussen een CachedRowSet en een ResultSet (of een JdbcRowSet) is dat een CachedRowSet de connectie zal sluiten na het inlezen. We kunnen het object gebruiken zonder dat we een connectie met de databank moeten openhouden:

```
CachedRowSet crs = factory.createCachedRowSet();
```

```
crs.setCommand("SELECT * FROM klanten");
try(Connection conn = getConnection()){
  crs.execute(conn);
}
crs.absolute(3);
crs.updateString("naam", "Josje");
crs.updateRow();
try(Connection conn = getConnection()){
  crs.acceptChanges(conn);
}
```

We vragen hier de lijst van alle klanten op. Om dat te doen hebben we een connectie met de databank nodig. Volgens de javadocs zou het connection object gesloten moeten worden, maar in de praktijk blijkt dat niet het geval te zijn. Daarom sluiten we de connectie hier via een try-with-resources. Aangezien we de CachedRowSet achteraf kunnen gebruiken, is het tevens een bewijs dat we geen geopende connectie nodig hebben om met de rowset te werken.

We kunnen de rowset gebruiken zoals een ResultSet. In dit voorbeeld doen we een aanpassing van een naam. Een belangrijk verschil met een ResultSet is dat de methode updateRow() niets aanpast in de databank. Dat is logisch aangezien er geen connectie is.

Om de gegevens effectief weg te schrijven naar de databank moeten we acceptChanges() aanroepen. Hiervoor hebben we weer een databank connectie nodig. Ook deze connectie moeten we weer zelf sluiten.

7. C. CachedRowSet en optimistic locking

Connecties met een databank worden best zo kort mogelijk gehouden. Wanneer we via een ResultSet aanpassingen doen in een databank, is bij een korte connectietijd de kans op aanpassingen door andere gebruikers heel klein. Bij een disconnected rowset is de kans reëel dat er ondertussen al aanpassingen gebeurd zijn. Een CachedRowSet maakt standaard gebruik van *optimistic locking*. Bij het wegschrijven van een rij wordt gecontroleerd of de data ondertussen niet gewijzigd zijn door een andere gebruiker. Hiervoor is de SyncProvider verantwoordelijk.

Wanneer we tussen het inlezen van de data en het wegschrijven van de gewijzigde data in de databank een aanpassing doen, zullen we een SyncProviderException krijgen:

```
CachedRowSet crs = factory.createCachedRowSet();
crs.setKeyColumns(new int[]{1});
crs.setCommand("SELECT * FROM klanten");
try (Connection conn = getConnection()) {
  crs.execute(conn);
}
crs.absolute(3);
crs.updateString("naam", "Joske");
crs.updateRow();
updateKlanten(factory);
try(Connection conn = getConnection()){
  try {
    crs.acceptChanges(conn);
```

```
  } catch (SyncProviderException ex) {
    System.out.println("Iemand anders heeft de gegevens aangepast");
  }
}
```

Na updateRow(), maar voor acceptChanges() veranderen we de gegevens in de databank via updateKlanten. Deze methode verandert de naam van de derde rij in "Josje". Bij het uitvoeren van acceptChanges() krijgen we een SyncProviderException omdat de gegevens ondertussen gewijzigd zijn.

Wanneer we de wijziging willen afhandelen, kunnen we via de SyncProviderException het SyncResolver object opvragen. We kunnen deze afhandeling echter alleen uitvoeren wanneer we bij de CachedRowSet keycolumns hebben gedefinieerd (setKeyColumns).

In de volgende code gaan we eerst naar de rij in de dataset waar er een conflict was (nextConflict()). Vervolgens kijken we of het een update conflict was. Aangezien we geen rijen hebben verwijderd of toegevoegd, kan het alleen een update conflict zijn.

Eénmaal dat we de rij hebben, moeten we ook weten in welke kolom het conflict is opgetreden. De methode getConflictValue(kolomnummer) geeft de waarde in de databank terug wanneer die ondertussen gewijzigd is. Wanneer de waarde niet gewijzigd is, geeft getConflictValue() null terug. Wanneer we de rij en de kolom hebben, kunnen we de gewijzigde waarde opvragen in het CachedRowSet object. In dit voorbeeld kiezen we ervoor de gewijzigde waarde toch te bewaren in de databank. Dat gebeurt via setResolvedValue(). Een SyncResolver is eigenlijk een resultset object. Voor de connectie met de databank gebruikt de SyncResolver dezelfde connectie als de acceptChanges() methode van de CachedRowset. Dat maakt het moeilijk om de afhandeling van het conflict in "usertime" te doen (=aan de gebruiker vragen welke waarde er in de databank bewaard moet worden). In de praktijk wordt deze code niet veel gebruikt maar ze toont wel de mogelijkheden die de SyncResolver klasse biedt.

```
SyncResolver resolver = ex.getSyncResolver();
while (resolver.nextConflict()) {
  if (resolver.getStatus() == SyncResolver.UPDATE_ROW_CONFLICT) {
    int rij = resolver.getRow();
    crs.absolute(rij);
    int aantalCols = crs.getMetaData().getColumnCount();
    for (int i = 1; i <= aantalCols; i++) {
      Object inDatabank = resolver.getConflictValue(i);
      if (inDatabank != null) {
        Object inApplicatie = crs.getObject(i);
          System.out.printf("In de databank: %s, in de applicatie: %s.\n",
inDatabank, inApplicatie);
          resolver.setResolvedValue(i, inApplicatie);
      }
    }
  }
 }
```

7. D. FilteredRowSet

Een FIlteredRowSet is een CachedRowSet waarop we een filter kunnen zetten zodat alleen een deel van de gecachte records zichtbaar zullen zijn. Die filter moet een klasse zijn die de Predicate interface implementeert:

```
class MaxIdFilter implements Predicate{
  private int maxId;
  private static final int COLID=1;
  private static final String COLNAME="ID";

  public MaxIdFilter(int maxId) {
    this.maxId = maxId;
  }
  @Override
  public boolean evaluate(RowSet rs) {
    try {
      ResultSetMetaData rsmd = rs.getMetaData();
      for(int i=1; i<=rsmd.getColumnCount();i++){
        if (! evaluate(rs.getObject(i), i)) return false;
      }
      return true;
    } catch (SQLException ex) {
      Logger.getLogger(MaxIdFilter.class.getName()).log(Level.SEVERE, null,
ex);
      return false;
    }
  }
  @Override
  public boolean evaluate(Object value, int column) throws SQLException {
    if (column != COLID) return true;
      return((Integer)value <= maxId);
    }
  @Override
  public boolean evaluate(Object value, String columnName) throws
SQLException {
    if (!columnName.equalsIgnoreCase(COLNAME)) return true;
      return((Integer)value <= maxId);
    }
}
```

Elk van de evaluate() methodes moet true teruggeven wanneer de informatie aan de filter voldoet en false wanneer dat niet het geval is. In dit geval controleert de filter of de id van een rij niet groter is dan een maximumwaarde. We kunnen de filter als volgt gebruiken:

```
FilteredRowSet frs = factory.createFilteredRowSet();
frs.setCommand("SELECT * FROM klanten");
try (Connection conn = getConnection()) {
  frs.execute(conn);
}
MaxIdFilter filter = new MaxIdFilter(2);
frs.setFilter(filter);
while (frs.next()) {
  int id = frs.getInt(1);
  String naam = frs.getString(2);
  System.out.printf("%d: %s\n", id, naam);
}
```

Via setFilter kunnen we het filterobject doorgeven aan de FilteredRowSet. Wanneer we de rowset overlopen, worden alleen de rijen getoond die voldoen aan de filtervoorwaarde. We zouden ook meerdere filters kunnen combineren door setFilter meerdere keren uit te voeren. Alleen de rijen die aan alle filters voldoen, zullen getoond worden.

Filters worden niet alleen gebruikt om de getoonde rijen te filteren. De filter is ook van toepassing voor een insert, een delete en een update. In het volgende voorbeeld werken we verder met de filteredrowset. We proberen een nieuwe rij toe te voegen, maar omdat ID=4 niet voldoet aan de filtervoorwaarden, krijgen we een SQLException bij de aanroep van updateInt("ID", 4).

```
frs.moveToInsertRow();
try {
  frs.updateInt("ID", 4);
  frs.updateString("naam", "Hanne");
  frs.insertRow();
  frs.beforeFirst();
  try (Connection conn = getConnection()) {
    frs.acceptChanges(conn);
  }
} catch (SQLException ex) {
  System.out.println(ex.getMessage());
}
```

Hoofdstuk 5. Multithreading

1. Inleiding

In een computersysteem maken we een onderscheid tussen processen en threads. Een proces is wat meestal een programma wordt genoemd, alhoewel die één-op-één relatie niet altijd waar hoeft te zijn. Maar wanneer we een programma starten, starten we een nieuw proces. In een besturingssysteem kun je op één of andere manier een lijst opvragen van alle processen die worden uitgevoerd.

Bij het starten van een proces wordt er altijd ook minstens één thread opgestart. De instructies van een programma worden binnen een thread uitgevoerd door een processor core. Een processor core kan maar één thread tegelijkertijd uitvoeren. Het is de verantwoordelijkheid van het besturingssysteem om de processortijd te verdelen over de verschillende threads.

Threads zouden dus gezien kunnen worden als processen-binnen-processen of, anders gezegd, een soort mini-programma's (binnen een programma). Er is echter een belangrijk verschil tussen een thread en een proces. Processen worden strikt van elkaar gescheiden. Ze kunnen bijvoorbeeld onmogelijk aan elkaars geheugen. Processen kunnen dus alleen in elkaars weg zitten wanneer ze met een externe resource willen werken, bijvoorbeeld twee programma's die tegelijkertijd hetzelfde bestand willen openen. Wanneer we twee processen met elkaar willen laten communiceren om bijvoorbeeld data uit te wisselen, moeten we gebruik maken van Inter Process Communication (IPC) mechanismen. Lezen en schrijven naar een gedeeld bestand is daar een primitief voorbeeld van.

Threads worden uitgevoerd binnen een proces en delen het geheugen van het proces. En dat zorgt ervoor dat we erg moeten opletten wanneer twee threads dezelfde geheugenlocatie kunnen aanspreken. Een eenvoudig statement zoals:

```
aantal++;
```

lijkt op een ondeelbaar of atomair statement zodat het in één keer zal worden uitgevoerd door een thread. Maar dit statement wordt vertaald naar een aantal machinecode constructies. En na het uitvoeren van elk van die constructies kan het besturingssysteem beslissen dat de huidige thread moet vervangen worden door een andere thread. Wanneer die op hetzelfde moment dit statement ook uitvoert, krijgen we onverwachte resultaten.

Een multithreaded programma schrijven en testen is niet zo gemakkelijk. Een programma kan perfect werken op de ene computer en in de problemen komen op een snellere of tragere machine. Bij het delen van het geheugen tussen de verschillende threads hebben we wel één voordeel. Alleen het *heap*-geheugen wordt gedeeld tussen de threads. Elke thread heeft zijn eigen stack en de lokale variabelen in een functie hebben geen last van meerdere threads.

De Java Virtual Machine zal altijd meerdere threads bevatten. Naast het programma dat wordt uitgevoerd is er ook een thread die verantwoordelijk is voor de *garbage collection*. Daar moet men als ontwikkelaar geen rekening mee houden. In deze cursustekst hebben we het alleen over de meerdere threads binnen één proces. We behandelen eerst de "primitieve" functies die gebruikt kunnen worden voor multithreading programma's. Daarna bekijken we de "hogere" objecten en functies die het werken met threads gemakkelijker maken.

2. Basis mulithreading

De basisklasse voor multithreading is Thread. In deze klasse en in de klasse Object zitten de methodes die we nodig hebben voor het werken met multithreading systemen. Wanneer we een klasse willen definiëren die via meerdere threads kan worden uitgevoerd, moet die de Runnable interface implementeren. In de Runnable interface is er maar één methode gedeifnieerd, namelijk run(). Hierin moet alle code staan die in de thread moet worden uitgevoerd.

Om de thread te laten stoppen, bestaat er nog steeds een methode .stop(), maar die is *deprecated*. Ze wordt dus best niet meer gebruikt. Het is aangewezen om de interrupt() methode te gebruiken. In de thread zelf kunnen we via Thread.interrupted() controleren of de thread onderbroken werd.

De volgende klasse bevat een run() methode die als aparte thread kan worden uitgevoerd. In die methode controleren we onder meer of de thread onderbroken is. Wanneer de thread zou onderbroken worden tijdens Thread.sleep(5) (doe gedurende 5 milliseconden niets), zal er een InterruptedException optreden. Om de interrupt in beide situaties eenvormig op te vangen, throwen we ook een InterruptedException wanneer interrupted() true is:

```
class MyThread implements Runnable {
  private String naam;
  public MyThread(String naam) {
    this.naam = naam;
  }
  @Override
  public void run() {
    try {
      for (int i = 0; i < 10; i++) {
        if (Thread.interrupted()){
          throw new InterruptedException("Onderbroken");
        }
        System.out.printf("Thread %s zegt 'hallo' \n", naam);
        Thread.sleep(5);
      }
    } catch (InterruptedException ex) {
      System.out.printf("Thread %s is onderbroken: %s\n", naam,
ex.getMessage());
    }
 }
}
```

In het volgende hoofdprogramma maken we twee Runnable objecten aan die elk in een aparte thread worden uitgevoerd. Om ze te starten gebruiken we de start() methode van de klasse Thread. Om ervoor te zorgen dat de hoofdthread pas verder gaat wanneer de eerste thread gestopt is, gebruiken we de join()-methode. Na de interrupt van de tweede thread wachten we niet tot de thread effectief gestopt is (via join). De laatste message van de hoofdtread kan dus verschijnen vóór de laatste message van thread2. Een Java programma is pas afgelopen wanneer alle threads gestopt zijn.

```
public static void main(String[] args) throws InterruptedException {
  Thread t1 = new Thread(new MyThread("thread1"));
  Thread t2 = new Thread(new MyThread("thread2"));
  t1.start();
  Thread.sleep(100);
  t2.start();
  t1.join();
  t2.interrupt();
  //t2.join();
  System.out.println("einde hoofd thread");
}
```

Zolang de beide threads zoals in dit voorbeeld volledig onafhankelijk van elkaar kunnen worden uitgevoerd, is er niets op tegen om de Thread klasse te gebruiken. Voor meer ingewikkelde situaties zijn high-level klassen meer aangewezen:

1. Twee threads kunnen dezelfde variabele aanspreken en wijzigen. Via het *lock* keyword zouden we kunnen voorkomen dat twee threads de waarde gelijktijdig aanspreken, maar Atomic Variables zijn waarschijnlijk een gemakkelijkere oplossing.
2. Een webapplicatie start een nieuwe thread voor elke request die binnenkomt. Wanneer er teveel requests binnenkomen, zullen er teveel threads worden aangemaakt waardoor we het systeem lamleggen. Via een threadpool kunnen we een vaste reeks threads voorzien. Elke request zal afgehandeld worden door een vrije thread uit de pool, maar wanneer er geen vrije threads meer zijn, komt de request in een wachtrij terecht.
3. Wanneer we grote arrays moeten verwerken, kan het op multiprocessor systemen handig zijn dat we de array opsplitsen in stukken om die stukken gelijktijdig op verschillende processoren te laten verwerken. Vanaf Java 8 zijn er een aantal parallelXXX() methodes in de Arrays klasse waarmee die parallelle verwerking gemakkelijk kan uitgevoerd worden.

3. Atomic variables

Een ogenschijnlijk niet opdeelbaar statement zoals

```
teller++;
```

bestaat toch uit verschillende stappen. Zolang we maar met één thread bezig zijn, zijn deze details niet belangrijk. Maar wanneer verschillende threads dit statement tegelijkertijd kunnen uitvoeren komen we in de problemen. In feite bestaat het uit drie stappen:

1. De huidige waarde van *teller* wordt opgevraagd.
2. De waarde wordt verhoogd.
3. De verhoogde waarde wordt opnieuw in *teller* bewaard.

Wanneer twee threads tegelijkertijd de huidige waarde opvragen, bijvoorbeeld 5, ze vervolgens verhogen naar 6 en de nieuwe waarde bewaren in *teller*, zal teller gelijk zijn aan 6. Maar eigenlijk zou de waarde 7 moeten zijn, want het statement is tweemaal uitgevoerd. We moeten dus erg opletten met dit soort van statements in een multithreading omgeving.

In oudere Java code moest dit opgelost worden met het *synchronized* keywoord, tegenwoordig kunnen we gebruik maken van de Atomicxxx klassen. In de volgende klasse Teller, maken we zowel gebruik van een gewone integer als van een AtomicInteger die allebei verhoogd kunnen worden:

```
class Teller{
    private AtomicInteger atomicTeller = new AtomicInteger();
    private int teller;
    public void verhoogAtomic(){
        atomicTeller.incrementAndGet();
    }
    public int getAtomicValue(){
        return atomicTeller.get();
    }
    public void verhoog(){
        teller++;
    }
    public int getValue(){
        return teller;
    }
}
```

Wanneer we de beide methodes oproepen via twee threads, zouden beide tellers achteraf de waarde 2000 moeten hebben:

```
public static void main(String[] args) throws InterruptedException {
  Teller teller = new Teller();
  Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
      for (int i=0;i<1000;i++) {
        teller.verhoog();
        teller.verhoogAtomic();
      }
    }
  });
  Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
      for (int i=0;i<1000;i++) {
        teller.verhoog();
        teller.verhoogAtomic();
      }
    }
  });
```

```
  t1.start();
  t2.start();
  t1.join();
  t2.join();
  System.out.println("De waarde van teller is "+teller.getValue());
  System.out.println("De atomic waarde van teller is " +
teller.getAtomicValue());
}
```

Afhankelijk van de snelheid van de processoren, zien we echter dat beide waarden kunnen verschillen en dat alleen de verhoging met de AtomicInteger correct werkt, bijvoorbeeld:

```
De waarde van teller is 1942
De atomic waarde van teller is 2000
```

Dat komt omdat incrementAndGet() ervoor zorgt dat twee threads nooit tegelijkertijd de waarde kunnen verhogen.

4. Executors

Executors laten toe om met threadpools te werken. Wanneer we zelf een Thread creëren, wil dat zeggen dat we een taak (run()) koppelen aan een thread. Wanneer de taak klaar is, is ook de thread klaar. Er is dus een één-op-één relatie tussen thread en taak. Maar wanneer de thread klaar is, is er misschien een andere taak die door die thread behandeld zou kunnen worden. Wanneer we zelf met de Thread klasse werken, kunnen we de thread niet hergebruiken om een andere taak uit te voeren éénmaal dat de run()-methode afgelopen is.

Wanneer we een threadpool gebruiken, worden er een aantal threads voorzien en worden de taken toegekend aan de threadpool. De threads in de pool voeren de taken uit en wanneer een taak klaar is, wordt een volgende taak afgehandeld door de thread.

Er worden drie soorten executors voorzien bij het Java platform:

- Executor: laat toe om een taak uit te voeren;
- ExecutorService: laat toe om een taak uit te voeren en achteraf een resultaat op te vragen via een Future
- ScheduledExecutorService: een ExecutorService die toelaat om een taak later of op geregelde tijdstippen uit te voeren.

In het volgende voorbeeld gebruiken we een Threadpool om via aparte threads op een inefficiënte manier een aantal fibonacci nummers te berekenen. Aangezien we de berekende getallen achteraf willen opvragen, moeten we een ExecutorService gebruiken. De klasse moet de Callable<T> interface implementeren om gebruikt te worden door een ExecutorService (Runnable voor een Executor):

```
class MyFib implements Callable<Integer> {
  private final int fibgetal;
  public MyFib(int fibgetal) {
    this.fibgetal = fibgetal;
```

```
  }
  private int fibonacci(int n) {
    if (n == 0) {
      return 0;
    } else if (n == 1) {
      return 1;
    } else {
      return fibonacci(n - 1) + fibonacci(n - 2);
    }
  }
  @Override
  public Integer call() throws Exception {
    return fibonacci(fibgetal);
  }
}
```

De call()-methode vervangt de run()-methode van Runnable. In het hoofdprogramma maken we een ExecutorService. We mogen niet vergeten om die af te sluiten. Omdat de Executor de threadpool beheerd en een Java programma pas stopt wanneer alle threads gestopt zijn, moeten we de threads stoppen alvorens het programma af te sluiten. Dat doen we via de shutdown() methode van de Executor service.

```
public static void main(String[] args) throws InterruptedException,
ExecutionException {
  ExecutorService exec = Executors.newFixedThreadPool(2);
  long start = System.nanoTime();
  Future<Integer> f1 = exec.submit(new MyFib(39));
  Future<Integer> f2 = exec.submit(new MyFib(39));
  Future<Integer> f3 = exec.submit(new MyFib(39));
  Future<Integer> f4 = exec.submit(new MyFib(40));
  while(!f4.isDone());
  long end = System.nanoTime();
  System.out.printf("Dit duurde %d milliseconden.\n", (end-start)/1000000);
  System.out.println("Eerste: " + f1.get());
  System.out.println("Tweede: " + f2.get());
  System.out.println("Derde: " + f3.get());
  System.out.println("Vierde: " + f4.get());
  exec.shutdown();
  System.out.println("Einde");
}
```

Wanneer we dit programma uitvoeren op een dual core computer zien we een substantiële verbetering in snelheid wanneer we een Threadpool van 2 threads maken ten opzichte van een Threadpool van 1 thread.

Hoofdstuk 6. Java Desktop GUI

1. Inleiding

Er worden tegenwoordig waarschijnlijk meer webapplicaties ontwikkeld in Java dan desktop applicaties. Dat heeft onder meer te maken met het feit dat de klassieke desktop minder belangrijk is geworden door de evolutie van de *mobile devices*. In dit hoofdstuk blijven we kort even stilstaan bij de ontwikkeling van Graphical User interfaces in een desktop omgeving.

Java heeft altijd libraries gehad om grafische desktop applicaties te ontwikkelen. De eerste grafische blibliotheek in Java was de Abstract Window Toolkit (AWT). Met de AWT waren er een aantal problemen. Ten eerste werden alle onderdelen van een grafische interface vertaald naar de onderdelen van het onderliggende platform. Het uitzicht van een applicatie (en soms zelfs de uitvoering) konden merkbaar verschillen tussen twee platformen. En dat ging natuurlijk in tegen het Java idee van “write once, run everywhere”. Aangezien de look-and-feel van de applicatie volledig bepaald werd door het onderliggende platform, had men hier als ontwikkelaar weinig controle over. Omdat AWT componenten (buttons, edit boxen, radiobuttons, …) ontwikkeld waren in de code van het onderliggende platform (*native code*) spreekt men ook over *heavyweight components*.

Vanaf Java 1.2 heeft men een nieuwe GUI bibliotheek toegevoegd: Swing. Swing maakt nog steeds gebruik van AWT. Maar de Swing componenten zelf zijn geschreven in Java. Daarom noemt men ze *lightweight components*. Men heeft dus meer controle over hoe een component er zal uitzien. Dat wordt mede in de hand gewerkt via de *Pluggable look and feel (PLAF)*. PLAF laat toe om hetzelfde uitzicht (look-and-feel) te definiëren voor de verschillende platformen.

Java FX is de nieuwste GUI omgeving. Net zoals Swing is Java FX lightweight en volgt het de Model-View-Controller architectuur. Maar er zijn een aantal zaken vergemakkelijkt:

- Mogellijkheid om de *Graphical User Interface* te ontwerpen in FXML. In AWT en Swing kon men een user interface alleen maar samenstellen door Java code te schrijven. Men zou dat de *imperative approach* kunnen noemen. (maak een component met die eigenschappen, voeg die toe aan die parent, …). FXML is een taal zoals HTML. We declareren welke componenten er zijn met hun eigenschappen en we zetten ze op de juiste plaats in het document. Wanneer we dit combineren met de Scene Builder tool, wordt het maken van Graphical User Interface eenvoudig.
- Gebruik van CSS. De vergelijking met HTML gaat verder. Want om de opmaak van een window te definiëren, kunnen we gebruik maken van CSS.
- Animatie is een swing een manueel proces. Wanneer we een object willen laten bewegen over het scherm, moeten we de verschillende stappen één voor één uittekenen. In JavaFX kunnen we een animatie definiëren: beginsituatie, parameters

die moeten veranderen, naar welke waarde dat ze moeten veranderen en hoe lang dat moet duren. Dit is ook weer een voorbeeld van *imperative* vs *declarative.*

2. Eenvoudig voorbeeld van Swing

De hoofdpagina van een swing applicatie moet afgeleid zijn van javax.swing.JFrame:

```
public class HoofdJFrame extends javax.swing.JFrame {
   public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new HoofdJFrame().setVisible(true);
            }
        });
    }
    …
}
```

In de main methode starten we een thread (Runnable) op via de invokeLater() methode van EventQueue. Het probleem met grafische applicaties is dat de componenten niet ontwikkeld zijn voor *concurrent* gebruik. Ze kunnen problemen opleveren wanneer ze vanuit verschillende threads worden aangepast. Daarom moeten aanpassingen aan properties van componenten (tekst van een tekstveld, kleur van een knop) in de Event Dispatching Thread worden uitgevoerd. Dat is de thread die de berichten afwerkt die naar de applicatie worden gestuurd zoals het klikken van een knop, het indrukken van een toets op het toetsenbord,…. Het tonen van het hoofdframe moet ook in die thread worden uitgevoerd en dat is waar invokeLater voor zorgt. De thread met de Runnable code wordt toegevoegd aan de queue van de Event Dispatching Thread en zal worden uitgevoerd wanneer alle andere events zijn afgewerkt.

Bij de creatie van het HoofdJFrame (dus nog voor het getoond wordt via setVisible(true)), moeten we de componenten toevoegen. Hiervoor zouden we de volgende code kunnen gebruiken:

```
jButton1 = new javax.swing.JButton();
jTextField1 = new javax.swing.JTextField();
jLabel1 = new javax.swing.JLabel();


setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
getContentPane().setLayout(new java.awt.FlowLayout());

jButton1.setText("Click Me!");
getContentPane().add(jButton1);

jTextField1.setColumns(20);
getContentPane().add(jTextField1);

jLabel1.setName(""); // NOI18N
getContentPane().add(jLabel1);

pack();
```

Na het initialiseren van de drie componenten die we op het frame willen zetten (button, tekstveld en label), zorgen we ervoor dat de applicatie stopt wanneer de gebruiker op de sluitknop klikt (setDefaultCloseOperation). Vervolgens definiëren we een layoutmanager voor het frame. In dit geval is dit een FlowLayout. De componenten zullen zich gedragen zoals de woorden in een tekstverwerker.

Eenmaal dat we een layout hebben, kunnen we de juiste properties invullen voor de drie componenten en ze toevoegen aan de content pane. Op het einde zorgen we ervoor dat elke component de juiste grootte krijgt, rekening houdend met de componenten die aanwezig zijn in het jFrame. (pack()).

Om code toe te voegen die moet worden uitgevoerd wanneer de gebruiker op de knop klikt, voegen we een actionListener toe:

```
jButton1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        String naam = jTextField1.getText();
        if ("".equals(naam)) naam="niemand";
        String begroeting = String.format("Dag %s", naam);
        jLabel1.setText(begroeting);
    }
});
```

We kunnen de waarde van de properties van de componenten opvragen via de getters en wijzigen via de setters.

3. Eenvoudig voorbeeld van Java FX

Een JavaFX applicatie moet afgeleid zijn van javafx.application.Application:

```
public class Main extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Parent root =
FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));

        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

We kunnen eventueel de init()- en stop()-methodes overschrijven van javafx.applicationApplication wanneer we code hebben die moet worden uitgevoerd voordat de eigenlijke JavaFX user interface opstart of die moet worden uitgevoerd nadat de applicatie is beëindigt. Beide methodes zijn standaard leeg. De init()-methode wordt niet in

de thread van de GUI uitgevoerd. Men mag hier geen user interface code schrijven. De stop()-methode wordt wel uitgevoerd in de GUI thread.

De launch()-methode van Application start de FX applicatie. Van hieruit zullen init(), start() en stop() worden opgeroepen. In de start()-methode hoort de code thuis die de user interface genereert. In dit voorbeeld maken we gebruik van een fxml bestand. We laden dit via getResources(). Een fxml-bestand beschrijft een *scene* in JavaFX termen. Een *scene* kunnen we vergelijken met de inhoud van een window. Een scene hoort thuis in een container: de *stage*.

In een fxml-document beschrijven we hoe het window er uit zal zien. We doen dit in een pane. Hiermee bepalen we de layout van het window In dit geval is dat een *anchorpane*. In een anchorpane wordt elke component verankerd op een bepaalde positie.

```
<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
xmlns="http://javafx.com/javafx/8.0.60" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="myfxml.FXMLDocumentController">
    <children>
         <Button fx:id="button" layoutX="126" layoutY="90"
onAction="#handleButtonAction" text="Click Me!" />
        <Label fx:id="label" layoutX="126.0" layoutY="140.0" minHeight="16"
minWidth="69" />
      <TextField fx:id="txtNaam" layoutX="126.0" layoutY="115.0"
promptText="Naam" />
    </children>
</AnchorPane>
```

In dit voorbeeld bevat het fxml document een button, een tekstveld en een label. Aan de button is een *action* gekoppeld. Die action is gedefinieerd in een controller class waarvan de naam gedefinieerd is in het *fx:controller* attribuut van <AnchorPane>:

```
public class FXMLDocumentController implements Initializable {

    @FXML
    private Label label;
    @FXML
    private TextField txtNaam;

    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
        String naam = txtNaam.getText();
        if ("".equals(naam)) naam="niemand";
        String begroeting = String.format("Dag %s", naam);
        label.setText(begroeting);
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }

}
```

Wanneer die klasse Initializable implementeert, kan ze code bevatten die wordt uitgevoerd nadat het fxml-bestand geïnitialiseerd is. We verwijzen naar de elementen in het fxml document door de naam van de variabele gelijk te stellen aan het id-attribuut. De objecten *label* en *txtnaam* zullen geïnitialiseerd worden door de FXMLLoader. Zoals het hoort zijn de member variabelen private gedeclareerd. Om ervoor te zorgen dat de FXMLLoader ze kan initialiseren, krijgen ze de annotation @FXML.

Hoofdstuk 7. Java configuratie

1. Inleiding

Aangezien java bestanden worden uitgevoerd binnen een *java virtual machine*, kunnen we de uitvoering van de code beheren door bepaalde configuratie-instellingen aan te passen in de *virtual machine.*

In dit hoofdstuk bekijken we de structuur van JAR bestanden en enkele opties die we kunnen meegeven bij het uitvoeren van de Java Virtual Machine.

2. JAR bestanden

De meeste Java applicaties worden uitgevoerd als JAR (Java ARchive) bestand. Een JAR-bestand is een gecomprimeerd (gezipt) bestand dat de .class bestanden in de juiste directory structuur bevat tezamen met andere resources die de applicatie nodig heeft.

Een JAR-bestand heeft een aantal voordelen:

- Het kan digitaal worden ondertekend, zodat gebruikers zeker kunnen zijn wie het JAR-bestand heeft gemaakt. In sommige omgevingen mogen alleen goedgekeurde JAR-bestanden worden uitgevoerd. Die goedkeuring kan samenhangen met de producent van de code. Alleen code van de producenten X, Y of Z mag worden uitgevoerd.
- De installatie wordt sterk vereenvoudigd doordat men geen afzonderlijke .class bestanden in de juiste directory structuur moet kopiëren. Dit geldt zowel voor applicaties als voor libraries. Het is ook een belangrijk voordeel wanneer code moet worden gedownload.
- Een JAR-bestand kan versie informatie bevatten. Packages in een JAR-bestand kunnen *sealed* worden. Dat wil zeggen dat alle .class bestanden in het package in hetzelfde JAR-bestand moeten staan.

We kunnen een JAR-bestand maken vanuit geïntegreerde programmeeromgevingen, maar we kunnen ook gebruik maken van het *jar*-commando:

```
jar –cf jarbestand inputbestanden
```

- De optie –c wil zeggen dat we een JAR-bestand willen creëren.
- De optie –f wil zeggen dat het resultaat moet worden weggeschreven naar het bestand dat volgt (jarbestand)
- De inputbestanden is een lijst van bestanden of directory gescheiden door een spatie. Er kunnen wildcards gebruikt worden (*). Wanneer een directory wordt meegegeven, wordt de volledige directory met inhoud toegevoegd aan het JAR-bestand.

Elk JAR-bestand moet een *Manifest* bestand bevatten. Dat is een bestand met als naam MANIFEST.MF dat in de META-INF directory staat. Het is een tekstbestand dat uit sleutel-waarde paren bestaat, bijvoorbeeld:

```
Manifest-Version: 1.0
Implementation-Title: myfxml
Implementation-Version: 1.0
```

Belangrijk bij een MANIFEST bestand is dat het niet mag eindigen met een lege lijn. Het laatste karakter moet een *newline* of een *carriage return* zijn.

Bij het maken van een JAR-bestand wordt er altijd een standaard MANIFEST bestand gegenereerd tenzij we de opie -M meegeven. We kunnen zelf extra informatie in het MANIFEST bestand steken via –m. Het commando wordt dan:

```
jar –cfm jarbestand manifest-items inputbestanden
```

Een entry die we meestal nodig hebben in het MANIFEST bestand is de Main-Class. Wanneer we een jar bestand willen uitvoeren:

```
java –jar JarBesntand
```

moet de Java Virtual Machine weten wat het startpunt of de startklasse is van de applicatie. Een JAR-bestand zal immers meer dan één klasse bevatten. Via de MANIFEST entry:

```
Main-Class: org.betavzw.test.Main
```

zal de main()-methode in de klasse Main in het package org.betavzw.test worden uitgevoerd wanneer het JAR-bestand wordt opgestart. In plaats van zelf een MANIFEST bestand te maken, kunnen we de Main-Class entry ook laten invullen bij het maken of updaten van het JAR-bestand via de schakeloptie –e

```
jar –cfe jarbestand inputbestanden org.betavzw.test.Main.class
```

Een MANIFEST bestand kan ook informatie bevatten over het classpath. Via de Class-Path entry kunnen we de andere JAR-bestanden aangeven die dienst zullen doen als bibliotheek:

```
Class-Path: jarbestand1 jarbestand2 directory/jarbestand3
```

3. Digitaal ondertekenen van een JAR bestand

Door code digitaal te ondertekenen wordt ze betrouwbaarder. Gebruikers kunnen zien wie de code ondertekend heeft en dus ook wie ze geschreven heeft. Want wanneer code eenmaal ondertekend is, kan ze niet meer gewijzigd worden. Een digitaal ondertekend bestand dat achteraf gewijzigd is, zal geen geldige ondertekening meer bevatten.

Om code digitaal te ondertekenen, moet men een certificaat hebben dat uitgegeven is door een betrouwbare *Certification Authority*. Betrouwbaar wil in dit geval zeggen, dat de *Certification Authority* vertrouwd wordt door de computer van de gebruiker. Naast dat certificaat moet men ook beschikken over de *private key* die bij dat certificaat hoort. Die private sleutel is erg belangrijk en mag nooit uit handen gegeven worden.

In dit voorbeeld maken we gebruik van de mogelijkheden van de Java Development Kit. We zullen geen gebruik maken van een *Certifcation Authority.* Het commando om public en private keys te genereren is *keytool*. Hiermee kunnen we een keystore aanmaken (mystore). Om een certificaat aan te maken, moeten we een alias definiëren en informatie over de houder van het certificaat:

```
C:\...\java advanced>keytool -genkeypair -alias signFiles -keystore mystore
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Joske Vermeulen
What is the name of your organizational unit?
  [Unknown]:  Opleiding
What is the name of your organization?
  [Unknown]:  Beta VZW
What is the name of your City or Locality?
  [Unknown]:  Hoboken
What is the name of your State or Province?
  [Unknown]:  Antwerpen
What is the two-letter country code for this unit?
  [Unknown]:  BE
Is CN=Joske Vermeulen, OU=Opleiding, O=Beta VZW, L=Hoboken, ST=Antwerpen,
C=BE correct?
  [no]:  yes
Enter key password for <signFiles>
        (RETURN if same as keystore password):
```

Er is nu een bestand *mystore* aangemaakt. Er is ook een sleutelpaar gegenereerd. Wanneer we een certificaat nodig hebben dat ondertekend is door een Certification Authority, moeten we een *certificate signing request* genereren:

```
C:…\java advanced>keytool -certreq -alias signFiles -keystore mystore -file
certreq.csr
Enter keystore password:
```

Het bestand certreq.csr moet opgestuurd worden naar de *Certification Authority*, tezamen met de nodige bewijzen van identiteit. In dit voorbeeld werken we echter verder met het oorspronkelijke certificaat. Dit certificaat is niet ondertekend door een trusted Certification Authority. Het is een *self-signed certifcate*.

```
C:\...\java advanced\myfxml\dist>jarsigner -keystore ..\..\mystore -
signedjar smyfxml.jar myfxml.jar signFiles
Enter Passphrase for keystore:
jar signed.

Warning:
The signer certificate will expire within six months.
```

```
No -tsa or -tsacert is provided and this jar is not timestamped. Without a
timestamp, users may not be able to validate this jar after the signer
certificate's expiration date (2016-10-31) or after any future revocation
date.
```

Nadat het certificaat ondertekend en geïmporteerd is, kunnen we het gebruiken om een jar-bestand te ondertekenen:

```
C:\...\java advanced>jarsigner -keystore ..\..\mystore -signedjar
smyfxml.jar myfxml.jar signFiles
```

Hierbij is “mystore” de naam van de keystore, myfxml.jar is het oorspronkelijke bestand, smyfxml.jar is het ondertekende bestand en “signFiles” is de alias van de key in de keystore.

4. Security policies

De Java Virtual Machine kan bepaalde acties toelaten of weigeren aan applicaties. Die mogelijkheden van de verschillende applicaties worden bewaard in een policy bestand. In het algemeen zijn er twee policy bestanden:

- Het systeem policy bestand in ${java.home}/lib/security/java.policy
- Het user policy bestand in ${user.home}/.java.policy

De volledige lijst van policybestanden staat in ${java.home}/lib/security/java.security:

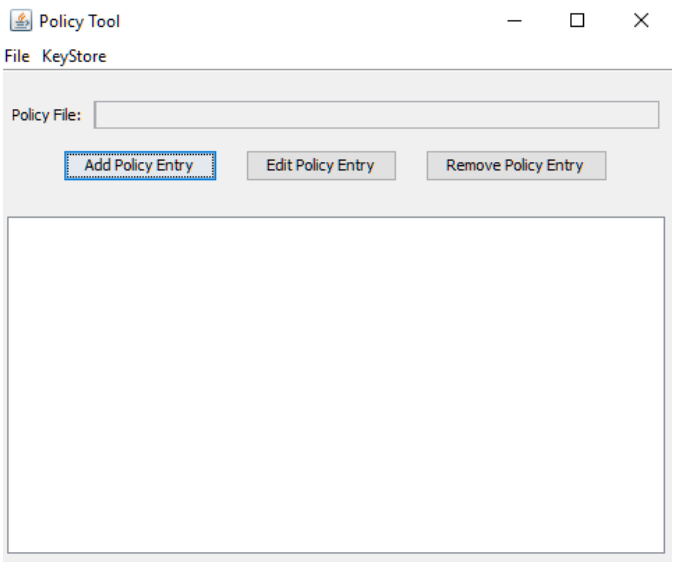
```
# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Deze security policies hebben echter geen effect op lokale applicaties die worden uitgevoerd. Wanneer we willen dat een applicatie onder deze policy valt, kunnen we dat meegeven tijdens het uitvoeren:

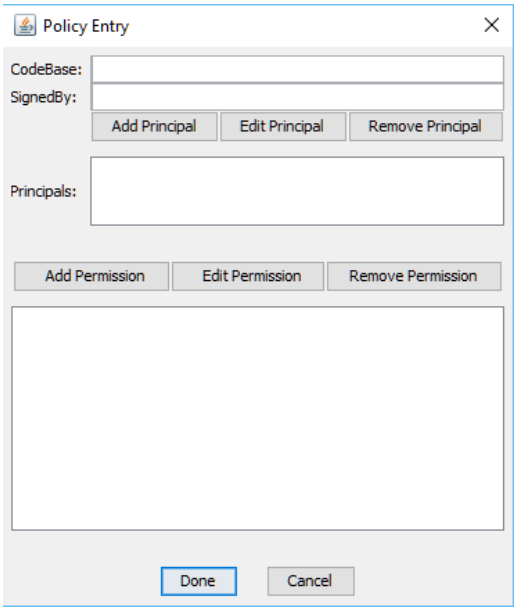
```
java -Djava.security.manager mainclassbestand
```

Voor applets die in een browser zouden worden uitgevoerd, is er een aparte security policy voorzien. Ook application servers (Websphere, JBoss, GlassFish, …) hebben hun eigen security manager die ervoor zorgt dat er geen gevaarlijke code kan worden uitgevoerd.

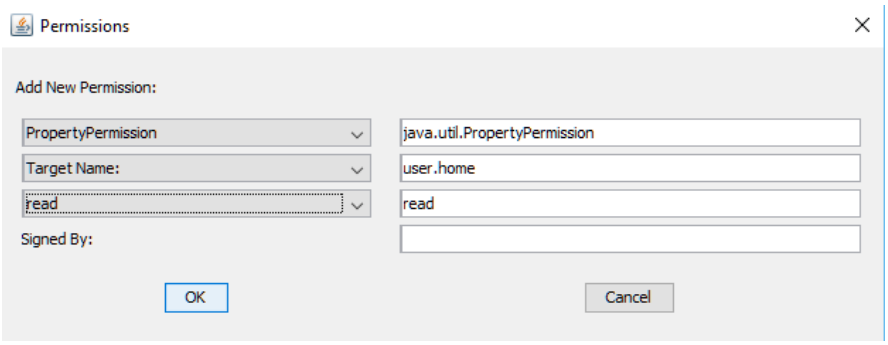
We kunnen een eigen security policy maken via de policytool:



Via Add Policy Entry kunnen we een policy toevoegen die toepasbaar zal zijn voor een bepaalde URL of voor code die ondertekend is door een bepaalde fabrikant:



De knop Add Permission laat toe om een bepaalde permissie te definiëren:



In dit voorbeeld krijgt men leesrechten op de property user.home. De applicatie zou op die manier te weten kunnen komen wat de home directory is van de gebruiker.

5. Opties bij het uitvoeren van de Java Virtual Machine

Wanneer we een applicatie opstarten, kunnen we extra opties meegeven. Eén ervan hebben we al gezien in het vorige voorbeeld. We kunnen system property waarden definiëren van de –D optie:

```
java -Djava.security.manager mainclassbestand
```

In dit voorbeeld is er geen waarde gedefinieerd voor de property . De meer algemene syntax is:

```
-Dproperty=value
```

Wanneer er spaties staan in *value* moet de waarde tussen aanhalingstekens worden gezet.

Andere interessante opties tonen wat er gebeurt tijdens het uitvoeren van het programma:

- - verbose: class toont informatie over de klassen die geladen worden
- -verbose:gc toont informatie over de garbage collection events
- -verbose:jni toont informatie over het gebruikt van *native methods*

Voor de Oracle Java Hotspot virtual machine zijn er een aantal opties die de werking van de garbage collection kunnen optimaliseren. Bij garbage collection configuratie zijn er twee mogelijke doelstellingen:

1. *Maximum Pause Time goal*: de pause time is de periode waarbij de applicatie gepauzeerd is (*Stop The World*) omdat de garbage collector bezig is. Voor interactieve applicaties probeert men deze tijd zo laag mogelijk te houden omdat de gebruikers de indruk zouden kunnen krijgen dat de applicatie niet meer reageert. We kunnen dit definiëren via -XX:MaxGCPauseMillis=<nnn>. Hierbij is <nnn> de maximum tijd in milliseconden dat een garbage collection sweep mag duren. Wanneer dit niet lukt, zal de heap verkleint worden zodat garbage collection sweeps elkaar sneller op zullen volgen (en dus minder lang zullen duren).
2. *Throughput goal:* bij de throughput heeft men het over de totale tijd die besteed wordt aan garbage collection in vergelijking met de tijd die besteed wordt aan het uitvoeren van de applicatie. Voor batch processen is dat belangrijker dan de maximum pause time. De throughput goal wordt gedefinieerd via -XX:GCTimeRatio=<nnn>. De waarde <nnn> wordt ingevuld in de formule 1/(1+n). Een waarde van 19 wil zeggen dat 1/20 van de tijd wordt besteed aan garbage collection. Wanneer het niet lukt om de throughput goal te halen, zal de grootte van de generaties worden verhoogd.

Wanneer beide doelstellingen behaald zijn, zal de JVM de heap kleiner maken (footprint goal) totdat één van beide goals niet meer behaald werden.

Bibliografie

Apache commons. 2015. Commons DBCP. *Apache commons.* [Online] 6 augustus 2015. [Citaat van: 26 juli 2016.] https://commons.apache.org/proper/commons-dbcp/.

Hall, Andrew. 2009. Thanks for the memory, Linux. *IBM developerWorks.* [Online] 21 April 2009. [Citaat van: 25 Juli 2016.] http://www.ibm.com/developerworks/library/j-nativememory-linux/.

Langer, Angelika. Java Generics FAQs - Under The Hood Of The Compiler. *Angelika Langer.* [Online] [Citaat van: 1 september 2016.] http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ100.

WInterberg, Benjamin. 2014. Java 8 Stream tutorial. *Benjamin Winterberg.* [Online] 31 juli 2014. [Citaat van: 2016 augustus 2016.] http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/.

De meest recente versie van deze referentielijst kan geraadpleegd worden op http://www.betavzw.org/wnopleidingen/java_advanced