

# **Programmeren in Java, de volgende stap**

Prof. K. Coolsaet

Universiteit Gent

*Vakgroep Toegepaste Wiskunde, Informatica en*

*Statistiek*

**2023–2024**



# Inhoud

<b>1. Bijkomende aspecten van de programmeertaal Java</b>	<b>1</b>
1.1. Opsomtypes . . . . .	1
1.2. Records . . . . .	7
1.3. Statische binnenklassen . . . . .	10
1.4. Niet-statische binnenklassen . . . . .	14
1.5. Functionele interfaces en lambda's . . . . .	18
1.6. Lambda's in de collectiebibliotheek . . . . .	24
1.7. Java streams . . . . .	27
1.8. Klassen, interfaces en methodes met typeparameters . . . . .	32
<b>2. Programmeren 'zonder if'</b>	<b>39</b>
2.1. Meervoudige selecties vermijden . . . . .	39
2.2. Meervoudige selectie van gedrag . . . . .	44
2.3. Het <i>visitor</i> -patroon . . . . .	50
<b>3. Persistente informatie</b>	<b>55</b>
3.1. Tekstbestanden lezen en schrijven . . . . .	56
3.2. <i>BufferedReader</i> en <i>PrintWriter</i> . . . . .	57
3.3. Opvangen van uitzonderingen . . . . .	60
3.4. Bestanden lezen uit het class path . . . . .	63
3.5. Het pakket <code>java.nio.file</code> . . . . .	65
3.6. Netwerkverbindingen opzetten en gebruiken . . . . .	67
3.7. Eigenschaftsbestanden . . . . .	70
3.8. XML met behulp van JDOM . . . . .	73
3.9. JSON met behulp van Jackson . . . . .	76

<b>4. JavaFX: aanvullingen</b>	<b>81</b>
4.1. Scènes bouwen <i>zonder</i> FXML . . . . .	81
4.2. Een eigen component definiëren . . . . .	86
4.3. Gelijkaardige componenten . . . . .	87
4.4. Een eigen <i>Button</i> -klasse . . . . .	92
4.5. Componenten opnieuw gebruiken . . . . .	95
4.6. Internationalisatie . . . . .	98
4.7. Drag en drop . . . . .	105
<b>5. Model/view/controller</b>	<b>111</b>
5.1. Model, view en controller . . . . .	111
5.2. Het model . . . . .	113
5.3. Het bladzijdelabel . . . . .	115
5.4. De vier knoppen . . . . .	117
5.5. De partnerklasse . . . . .	118
5.6. Views en controller samen in het partnerobject . . . . .	120
5.7. JavaFX-eigenschappen . . . . .	121
5.8. Een JavaFX-eigenschap als model . . . . .	125
5.9. Eigenschappen verbinden . . . . .	126
<b>6. Databanken aanspreken met JDBC</b>	<b>131</b>
6.1. Drivers en verbindingen . . . . .	132
6.2. Opdrachten . . . . .	134
6.3. Batch-bewerkingen . . . . .	139
6.4. Automatisch gegenereerde sleutels . . . . .	141
<b>7. Draden in JavaFX</b>	<b>143</b>
7.1. Draden in Java . . . . .	143
7.2. Draden en JavaFX . . . . .	145
7.3. De klasse <i>Task</i> . . . . .	149
7.4. Het tonen van voortgang . . . . .	152
7.5. Een taak annuleren . . . . .	153

<b>8. Complexe componenten</b>	<b>157</b>
8.1. De combo box en de cell factory . . . . .	157
8.2. Een boomdiagram . . . . .	164
8.3. Het selectiemodel . . . . .	167
8.4. Een eenvoudige tabel . . . . .	169
8.5. Cell factories voor tabellen . . . . .	172
8.6. Cell value factories . . . . .	177
8.7. Editeerbare tabellen . . . . .	179
<b>9. Databanktoegang abstraheren</b>	<b>189</b>
9.1. Data access objects . . . . .	189
9.2. Data access context en data access provider . . . . .	192
9.3. Implementatie van de data access-bibliotheek . . . . .	194
9.4. Filteren met een fluent interface . . . . .	198
<b>A. Databanken en SQL</b>	<b>203</b>



# Woord vooraf

Als beginnend Java-programmeur blijf je wellicht toch nog een beetje op je honger zitten. Je hebt nu een goed idee van de basisprincipes van de programmeertaal Java en van objectgericht programmeren in het algemeen maar eigenlijk ben je nog niet klaar voor het ‘echte’ werk: toepassingen die ook in de reële wereld praktisch bruikbaar zijn.

Zoals je aan de titel van deze nota’s kunt zien, zetten we nu een volgende stap. Als rode draad doorheen de tekst leren we je hoe je in Java toepassingen maakt met een grafische gebruikersinterface (*graphical user interface*, *GUI*, spreek uit ‘goe-ie’). We geven je ook een eerste inleiding tot het werken met bestanden en databanken.

Daarnaast maken we ook van de gelegenheid gebruik om enkele van de meer gevorderde aspecten van de programmeertaal te introduceren: binnenklassen, opsomtypes, records, zogenaamde ‘lambda’s en generieke klassen en methodes.

Onze hoofdbedoeling is niet om je te overstelpen met encyclopedische weetjes over allerhande Java-bibliotheken, maar om je te laten kennis maken met enkele courante ontwerptechnieken voor het schrijven van middelgrote Java-programma’s.

Je weet ondertussen wel dat een goed Java-programma uit heel wat klassen bestaat en dat het niet altijd eenvoudig is om die onderling met elkaar te laten samenwerken. Het is daarom heel belangrijk om voldoende tijd te besteden aan het ontwerp van de juiste klassenstructuur. We hopen je hierbij te kunnen helpen.

De broncode voor de voorbeelden die in deze nota’s worden gebruikt, kan je terugvinden op (en ophalen van) <https://github.ugent.be/kcoolsae/ObjProg><sup>1</sup>.

KC, januari 2024

---

<sup>1</sup>Lezers die geen toegang hebben tot deze site, kunnen zich rechtstreeks wenden tot de auteur — liefst per e-mail.





# 1. Bijkomende aspecten van de programmeertaal Java

In een basiscursus Java kan je niet alle aspecten van de programmeertaal behandelen. In de praktijk betekent dit vaak dat men zich beperkt tot wat er in de eerste versies van Java mogelijk was, en de ‘nieuwigheden’ uit latere versies voorlopig links laat liggen. Nochtans zijn er sinds de introductie van Java in 1995 heel wat elementen aan de programmeertaal toegevoegd die ook voor een beginner reeds interessant kunnen zijn.

In dit hoofdstuk proberen we deze achterstand een stukje in te halen.

## 1.1. Opsomtypes

[ De Java-broncode van onderstaande voorbeelden vind je in het pakket (de *package*) `be.ugent.objprog.extra`. ]

Kortweg is een opsomtype een klasse waarvan de objecten reeds *at compile time* kunnen worden opgesomd. Elk van die objecten is op voorhand reeds aangemaakt en heeft een vaste naam. Maar dit wordt wellicht duidelijker met een voorbeeld.

In een simulatieprogramma kunnen voorwerpen verplaatst worden in vier verschillende richtingen (oost, zuid, west, noord). We geven deze vier richtingen intern aan met de getallen 0, 1, 2 en 3, maar om het programma bevattelijker te maken, willen we een manier vinden om naar de richtingen te verwijzen met hun namen. In oudere versies van Java zou men hiervoor vier gehele constanten introduceren:

```
public static final int OOST = 0;  
public static final int ZUID = 1;  
public static final int WEST = 2;  
public static final int NOORD = 3;
```

Een veld *richting* dat één van deze vier richtingen kan bevatten, wordt dan gedeclareerd en gebruikt als geheel getal:

```
private int richting = NOORD;
```

Deze manier van werken heeft enkele nadelen:

- Je kan zonder problemen elk mogelijk geheel getal toekennen aan *richting* ook als dit getal niet overeenkomt met een echte windrichting (bijvoorbeeld 5 of -3). De compiler kan niet zien dat een toewijzing zoals *richting=aantal* wellicht verkeerd is.
- Wanneer je een windrichting afdruckt met *System.out.println*, dan verschijnt er gewoon een getal. Je moet extra werk verzetten om een richting af te beelden als één van de woorden ZUID, NOORD, OOST of WEST. Daarbij komt nog dat een richting door een primitief type wordt voorgesteld, en dat we dus ook *toString* niet zomaar kunnen overschrijven.

Het is daarom beter om een windrichting voor te stellen als een opsomtype, bijvoorbeeld met de naam *Windrichting*. Dit is hoe je dit type definieert:

```
public enum Windrichting {  
    ZUID, NOORD, OOST, WEST  
}
```

Het veld *richting* wordt nu gedeclareerd op de volgende manier:

```
private Windrichting richting = Windrichting.NOORD;
```

Een opsomtype is eigenlijk een bijzonder soort klasse. Waar je anders **class** zou verwachten, staat nu het sleutelwoord **enum**.

In zijn meest eenvoudige vorm bevat de definitie van een opsomtype enkel maar een opsomming van een aantal constanten. Elk van deze constanten is een object van de overeenkomstige klasse dat automatisch wordt aangemaakt. Het is onmogelijk om later nog nieuwe objecten van dit type aan te maken. De opdracht '**new** *Windrichting* ()' werkt niet. Dit heeft als (gelukkig) gevolg dat je opsomtypes steeds op een betrouwbare manier met '=' kan vergelijken en dus *equals* niet hoeft te gebruiken.

Merk op dat de compiler nu wel een fout zal aanrekenen bij een toewijzing zoals '*richting = aantal*', windrichtingen zijn nu niet langer gehele getallen maar

objecten van de klasse *Windrichting*. Ook geeft de methode *toString* van een **enum**-type de naam van de constante terug. Opsomtypes erven bovendien automatisch enkele zeer bruikbare methodes van de klasse *Enum*. (We verwijzen naar de elektronische documentatie van deze klasse voor meer informatie.)

Het feit dat de constanten uit een opsomtype geen gehele getallen zijn, is op het eerste zicht misschien ook een nadeel. Stel bijvoorbeeld dat we voor elke windrichting de naam willen opslaan van de overeenkomstige Latijnse benaming. Bij de originele implementatie maken we gewoon een tabel aan en gebruiken de windrichting als index in die tabel:

```
String[] latijnseNaam = {  
    "Subsolanus", "Auster", "Favonius", "Septentrio"  
};  
...  
String naam = latijnseNaam[richting];
```

Als *richting* van het type *Windrichting* is, dan kan dit echter niet meer. Er bestaan verschillende manieren om dit op te lossen.

Het gemakkelijkste is om de methode *ordinal()* van *Enum* gebruiken. Die geeft het volgnummer terug in de lijst die het opsomtype definieert. Dus:

```
String naam = latijnseNaam[richting.ordinal()];
```

Het gebruik van *ordinal* is echter geen goed idee, omdat dit voor problemen zorgt wanneer je later de volgorde van de opsomming zou veranderen, of nog andere elementen toevoegt. Je moet dan overal in je programma de corresponderende tabellen aanpassen — en in een groot project zal je al eens gemakkelijk zo'n tabel over het hoofd zien.

Het is beter om de tabel te vervangen door een *map* met als sleutel het opsomtype en als waarde de latijnse naam:

```
Map<Windrichting,String> map =  
  
map.put (Windrichting.OOST, "Subsolanus");  
map.put (Windrichting.ZUID, "Auster");  
map.put (Windrichting.WEST, "Favonius");  
map.put (Windrichting.NOORD, "Septentrio");  
...  
String naam = map.get (richting);
```

Java voorziet een bijzonder soort *Map*, een *EnumMap*, die even efficiënt is als de tabellen uit het vroegere systeem:

```
Map<Windrichting,String> map = new EnumMap<>(Windrichting.class);
```

De parameter van de constructor van *EnumMap* is een zogenaamd *klassenobject*. Dit is een object dat intern (*at run time*) gebruikt wordt om een (*compile time*) klasse voor te stellen. Voor elke klasse *K* is er één dergelijk object, genoteerd als '*K.class*'. We zullen dergelijke klassenobjecten later nog in andere contexten ontmoeten.

Een tweede mogelijkheid maakt gebruik van het feit dat opsomtypes klassen zijn, en dus elementen zoals *ZUID* en *NOORD* echte objecten. Net zoals bij gewone klassen, kan een opsomtype velden, constructoren en methodes bevatten.

In het bijzonder kunnen we in *Windrichting* het veld *latijnseNaam* declareren, met corresponderende getter *getLatijnseNaam*:

```
public enum Windrichting {  
    ... // zie verder  
  
    private String latijnseNaam;  
  
    private Windrichting (String latijnseNaam) {  
        this.latijnseNaam = latijnseNaam;  
    }  
  
    public String getLatijnseNaam() {  
        return this.latijnseNaam;  
    }  
}
```

Je kan dan de Latijnse naam gewoon ophalen van het object:

```
String naam = richting.getLatijnseNaam();
```

We moeten natuurlijk nog op één of andere manier de velden invullen voor elk van de vier constanten uit *Windrichting*. Was *Windrichting* een gewone klasse, dan schreven we wellicht

```

OOST = new Windrichting ("Subsolanus");
ZUID = new Windrichting ("Auster");
WEST = new Windrichting ("Favonius");
NOORD = new Windrichting ("Septentrio");

```

maar **new** is niet toegelaten bij opsomtypes (hun constructoren zijn ook altijd *private*). In de plaats daarvan gebruikt Java een speciale notatie voor de definitie van de constanten van een opsomtype. De definitie van de klasse *Windrichting* begint namelijk op de volgende manier:

```

public enum Windrichting {
    OOST("Subsolanus"),
    ZUID("Auster"),
    WEST("Favonius"),
    NOORD("Septentrio");
    ...
}

```

Bij elke constante die we opsommen, geven we de argumenten mee voor de constructor waarmee deze constante moet worden aangemaakt.

Java doet erg zijn best om het werken met opsomtypes te vereenvoudigen. Zo krijgt elk opsomtype bijvoorbeeld automatisch een klassenmethode *values()* cadeau. Deze methode geeft een tabel terug met alle mogelijk opsomwaarden, in de volgorde waarop ze zijn gedeclareerd. In het volgende fragment drukken we bijvoorbeeld alle mogelijke windrichtingen af:

```

for (Windrichting richting: Windrichting.values()) {
    System.out.println (richting);
}

```

Dat **enum**-types meer zijn dan enkel een opsomming van constanten zullen we nog wat beter in de verf te zetten aan de hand van een tweede voorbeeld.

Een (GUI-)toepassing gebruikt een lijst van personen die we op twee verschillende manieren kunnen manipuleren: we kunnen een plaats in de lijst aanduiden, de naam van een persoon intikken en dan ofwel deze persoon *invoegen* op de aangegeven plaats in de lijst, of anders het element dat op die plaats in de lijst staat *vervangen* door de nieuwe persoon.

Welke van de twee acties wordt ondernomen, hangt af van de *modus* waarin het

programma zich bevindt, ofwel in *invoegmodus* of in *overschrijfmodus*. (Deze modus kan dan bijvoorbeeld ingesteld worden door op de INSERT-toets te drukken.)

Waar we vroeger de modus zouden voorstellen als een geheel getal (0 of 1), of iets beter nog, als een logische waarde, weten we nu dat we hier best een opsomtype gebruiken:

```
public enum Modus {  
    INVOEGEN, OVERSCHRIJVEN  
}
```

Het aanpassen van de lijst gebeurt dan met een methode zoals deze:

```
public void aanpassen (List<Persoon> lijst, int index, Persoon persoon) {  
    if (modus == Modus.INVOEGEN) {  
        lijst .add (index, persoon);  
    } else {  
        lijst .set (index, persoon);  
    }  
}
```

Opsomtypes bieden echter de mogelijkheid om dit op een meer objectgeïntende manier te doen. We definiëren een methode *aanpassen* in de klasse *Modus* die de lijst aanpast op de juiste manier en schrijven dan

```
public void aanpassen (List<Persoon> lijst, int index, Persoon persoon) {  
    modus.aanpassen (lijst, index, persoon);  
}
```

in het hoofdprogramma.

Java biedt een korte manier om aan te geven dat de methode *aanpassen* van *Modus* zich anders moet gedragen bij *INVOEGEN* als bij *OVERSCHRIJVEN*:

```
public enum Modus {  
  
    INVOEGEN {  
        public void aanpassen(List<Persoon> lijst,int index,Persoon persoon) {  
            lijst .add (index, persoon);  
        }  
    },  
}
```

```

OVERSCHRIJVEN {
    public void aanpassen(List<Persoon> lijst,int index,Persoon persoon) {
        lijst .set (index, persoon);
    }
};

public abstract void aanpassen
    (List<Persoon> lijst,int index,Persoon persoon);

}

```

Deze notatie verbergt heel wat detail: *Modus* is nu een abstracte klasse geworden, en de constanten *INVOEGEN* en *OVERSCHRIJVEN* zijn weliswaar van het type *Modus*, maar behoren elk tot een verschillende verborgen en naamloze klasse die *Modus* overschrijft, elk met haar eigen implementatie van de methode *aanpassen*.

## 1.2. Records

[ Onderstaand voorbeeld vind je in *be.ugent.objprog.mails.* ]

In één van de voorbeelden hieronder (§1.5) werken we met een klasse *Mail* waarmee e-mailberichten worden voorgesteld. Deze klasse bevat vier velden, vier corresponderende getters en één constructor:

```

public class Mail {

    private final String subject;
    private final String sender;
    private final String message;
    private final LocalDateTime time;

    public Mail (String subject, String sender, String message, LocalDateTime time) {
        this.subject = subject;
        this.sender = sender;
        this.message = message;
        this.time = time;
    }
}

```

```

    public String getSubject() {
        return subject;
    }

    // ... getters voor sender, message en time.
}

```

De klasse *LocalDateTime* stelt een combinatie van datum en tijd voor. Java biedt in het pakket *java.time* een resem klassen aan voor tijdstippen, datums, periodes, e.d. (Er wordt ten zeerste afgeraden om hiervoor de klasse *Date* te gebruiken die nog dateert vanuit de beginperiode van Java.)

Merk op dat bij de velden van *Mail* telkens het sleutelwoord **final** staat. Tot nog toe hebben we **final** enkel gebruikt bij de declaratie van een constante.

```

public static final double BETERE_PI = 3.14;

```

De toevoeging **final** kan echter ook op andere plaatsen gebruikt worden, zoals hier bij een veld (instantievariabele). Daarmee geef je aan dat je belooft enkel in de constructor aan dat veld iets toe te wijzen, maar niet in een andere methode van die klasse. In zeker zin is de waarde van dit veld dus ook constant - ze verandert niet tijdens de levensduur van het object. (IntelliJ IDEA zal voor een veld van een klasse vaak suggereren om dit final te maken, als hij merkt dat er inderdaad nooit iets aan wordt toegewezen. Het is geen slechte gewoonte om hem dat dan ook te laten doen.)

Je kan ook **final** plaatsen in de hoofding van een methode. Dat betekent dat deelklassen die methode niet mogen overschrijven. Dit wordt in de praktijk weinig gebruikt.

Op dezelfde manier kan je **final** toevoegen aan de hoofding van een klasse, en dat betekent dat die klasse geen extensies toelaat. Ook dit wordt weinig gebruikt, maar er bestaat wel een conventie dat we **final** plaatsen bij een klasse die enkel uit klassenmethoden bestaat en waarvan het niet de bedoeling is dat er objecten van gemaakt worden. De klasse *Math* is hiervan een mooi voorbeeld.

Een klasse waarvan *alle* velden **final** zijn<sup>1</sup>, heet *onveranderlijk* (Engels: *immutable*). Dergelijke klassen komen in de praktijk heel veel voor — het standaardvoorbeeld hiervan is *String*.

---

<sup>1</sup>Om echt onveranderlijk te mogen heten, moet ook het type van elke van die velden onveranderlijk zijn, anders zou je de inhoud van een object ook onrechtstreeks kunnen aanpassen via zijn velden.



De broncode van een onveranderlijke klasse heeft altijd dezelfde structuur (zogenaamde *boilerplate* code). Programmeeromgevingen zoals IDEA kunnen je helpen om heel snel een dergelijke klasse te schrijven, maar sinds Java 17 biedt de programmeertaal nu ook zelf ondersteuning hiervoor.

Vanaf nu kan je een onveranderlijke klasse *Mail* met de vier opgegeven velden, op de volgende manier heel compact definiëren

```
public record Mail (String subject, String sender,  
                  String message, LocalDateTime time) {  
}
```

(Zonder **package**- en **import**-opdrachten dus slechts twee lijnen!)

Een dergelijk ‘record’ is een gewone klasse, ze is enkel op een andere manier gedefinieerd. Je kan dus nog steeds een object van de klasse *Mail* aanmaken met **new**.

```
Mail mail =  
    new Mail ("Re: Java", "kc", "...", LocalDateTime.now());
```

Het is toegelaten om in een **record**-definitie nog bijkomende methoden te definiëren, maar in de praktijk zullen we dit niet vaak doen (en dan liever gewone klassen gebruiken.) Je kan geen bijkomende velden definiëren en je kan van een record-klasse ook niet overerven.

Er is één belangrijk verschil tussen de record-klasse *Mail* en de klasse *Mail* zoals we ze eerder hebben gedefinieerd. De accessoren in een record-klasse gebruiken niet de getter-conventies voor hun namen, maar heten hetzelfde als de velden. Om bijvoorbeeld het onderwerp van het object *mail* op te vragen, schrijf je

```
String onderwerp = mail.subject(); // Let op de notatie!
```

en niet *mail.getSubject()* zoals voorheen.

## 1.3. Statische binnenklassen

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.inner.* ]

Het is in Java toegelaten om een klassendefinitie op te nemen *binnenin* de definitie van een andere klasse. Een dergelijke klasse noemt men een *vernestelde* klasse (Engels: *nested class*) of een *binnenklasse* (*inner class*) en de klasse waarin ze is opgenomen, heet dan haar *buitenklasse* (*outer class*).

Er zijn verschillende soorten vernestelde klassen waarvan de meest eenvoudige om te begrijpen de zogenaamde *statische* binnenklasse<sup>2</sup> is. De definitie van een dergelijke klasse ziet er zo uit:

```
public class Buiten {  
    ...  
    public static class Binnen {  
        ...  
    }  
    ...  
}
```

De naam ‘statische’ binnenklasse komt van het sleutelwoord **static** dat hier gebruikt wordt.

Er is heel weinig verschil tussen een statische binnenklasse en een gewone klasse. Het belangrijkste is de manier waarop die klasse benoemd wordt. Willen we bijvoorbeeld een object aanmaken van de klasse *Binnen*, dan schrijven we **new Buiten.Binnen()** — behalve wanneer we dit object aanmaken vanuit de klasse *Buiten* zelf, want dan mag je dit afkorten tot **new Binnen()**.

Een tweede verschil is dat binnenklassen ook **private** kunnen zijn. In dat geval is de klassennaam slechts zichtbaar binnenin de buitenklasse (wat nog niet betekent dat er geen objecten van die klasse in de rest van het programma kunnen verzeild raken).

Statische binnenklassen worden meestal gebruikt wanneer de objecten van die klasse hoofdzakelijk betekenis hebben in de context van hun buitenklasse, maar niet daarbuiten. Een typisch voorbeeld hiervan zijn de zogenaamde *data transfer*

---

<sup>2</sup>Officieel heet dit een statische vernestelde klasse en wordt de term ‘binnenklasse’ voorbehouden voor de andere soorten binnenklasse (cf. §1.4). Het belangrijkste woord is hier ‘statisch’.

*objects* (DTOs), objecten die enkel gebruikt worden om waarden te groeperen en mee te geven als parameters of retourwaarde van een functie.

We willen bijvoorbeeld een methode schrijven die de som bepaalt van de vierkantswortels van alle positieve getallen in een gegeven lijst en tegelijk ook telt hoeveel getallen in de lijst negatief zijn. We zouden hiervoor twee afzonderlijke functies kunnen schrijven, één voor de som van de vierkantswortels en één die telt hoeveel negatieve getallen er zijn, maar eigenlijk is dit zonde van het werk: dan moeten we dezelfde lijst twee keer overlopen.

De implementatie van een dergelijke functie is absoluut niet moeilijk, de vraag is enkel hoe we de resultaten zullen teruggeven: de som (een **double**) samen met het aantal (een **int**).

We zouden beide waarden bijvoorbeeld in een tabel van twee *Objects* kunnen stoppen (of van *Numbers*, of zelfs van **doubles**) maar dit resulteert in code die moeilijk leesbaar is en onderhoudbaar. Het beste is om hier een kleine klasse aan te maken die beide resultaten tegelijk kan bevatten.

En omdat we deze klasse wellicht enkel voor dit doel zullen gebruiken, ligt een statische binnenklasse voor de hand.

```
public class MijnKlasse {  
  
    public static class Resultaat {  
  
        private double som;  
        private int aantal;  
  
        public Resultaat (double som, int aantal) {  
            this.som = som;  
            this.aantal = aantal;  
        }  
  
        public double getSom() {  
            return this.som;  
        }  
  
        public int getAantal() {  
            return this.aantal;  
        }  
    }  
}
```

```

public Resultaat mijnFunctie (List<Double> lijst) {
    int aantal = 0;
    double som = 0.0;
    for (double element: lijst) {
        if (element >= 0.0) {
            som += Math.sqrt(element);
        } else {
            aantal ++;
        }
    }
    return new Resultaat (som, aantal);
}

...
}

```

Dit is trouwens een ideale gelegenheid om een record te gebruiken:

```

public class MijnKlasse {

    public static record Resultaat(double som, int aantal) {
    }

    public Resultaat mijnFunctie(List<Double> lijst) {
        // .. zelfde broncode als voorheen
    }

    ...
}

```

(Het sleutelwoord **static** mag bij een record-binnenklasse worden weggelaten. Records mogen immers niet gebruikt worden als niet-statische binnenklasse.)

Zoals al aangegeven, worden DTOs ook gebruikt om parameters te groeperen. In de volgende functie kijken we of twee gebieden in een rechthoekig rooster elkaar overlappen: voor elk gebied geven we het rijnummer en kolomnummer van de linker bovenhoek op en de breedte en hoogte (in aantallen kolommen en rijen). We kunnen de volgende signatuur hiervoor gebruiken:

```

public boolean overlappen (int rij1, int kolom1, int hoogte1, int breedte1,
                           (int rij2, int kolom2, int hoogte2, int breedte2)

```

Acht parameters is nogal zwaar op de hand (en nog goed dat het niet over een driedimensionaal rooster gaat), daarom is een DTO hier opnieuw een goed alternatief:

```
public record Gebied (int rij, int kolom, int hoogte, int breedte) {  
}  
  
public boolean overlappen (Gebied gebied1, Gebied gebied2) {  
    ...  
}
```

Opgelet! Als je nog meer methodes hebt die dergelijke gebieden als parameter nemen, is het misschien beter om van *Gebied* toch een ‘echte’ klasse te maken en de methodes naar die klasse te verhuizen. (De functie *overlappen* wordt dan misschien *overlaptMet*, met één parameter.)

Tot slot willen we ook nog vermelden dat statische binnenklassen ook toegelaten zijn binnen interfaces en records, en dat ook interfaces binnen andere klassen (of interfaces, of records) mogen gedefinieerd worden.

De volgende klasse bevat bijvoorbeeld een methode *vertaalLijst* die een lijst van woorden omzet naar een nieuwe lijst met de vertalingen van die woorden.

```
public class Vertaling {  
    public interface Vertaler {  
        String vertaal(String woord);  
    }  
    ...  
    public List<String> vertaalLijst  
        (List<String> woordenLijst, Vertaler vertaler) {  
        List<String> resultaat = new ArrayList<> ();  
        for (String str: woordenLijst) {  
            resultaat.add (vertaler.vertaal(str ));  
        }  
        return resultaat;  
    }  
}
```

(Voor **interface** hoeft er hier geen **static** te staan.)

De volledige naam van de interface is *Vertaling.Vertaler*. Wanneer je de methode *vertaalLijst* gebruikt, moet je niet alleen een lijst van woorden als argu-

ment meegeven, maar ook een object van een (wellicht zelfgeschreven) klasse die *Vertaling.Vertaler* implementeert — en dus een methode *vertaal* bevat die een string omzet naar een andere string.

Ook opsomtypes kunnen statische binnenklassen zijn:

```
public class MijnKlasse {  
    public enum Richting {  
        OOST, ZUID, WEST, NOORD;  
    }  
  
    public Punt verplaats (Punt origineel, Richting richting) {  
        ...  
    }  
}
```

(Opnieuw mag je hier het sleutelwoord **static** weglaten vóór **enum**.)

## 1.4. Niet-statische binnenklassen

Naast de statische binnenklassen bestaat er nog een tweede soort vernestelde binnenklassen die in de praktijk veel meer toegepast worden. Hun definitie ziet er hetzelfde uit als bij statische binnenklassen, alleen is het woord **static** nu weggelaten:

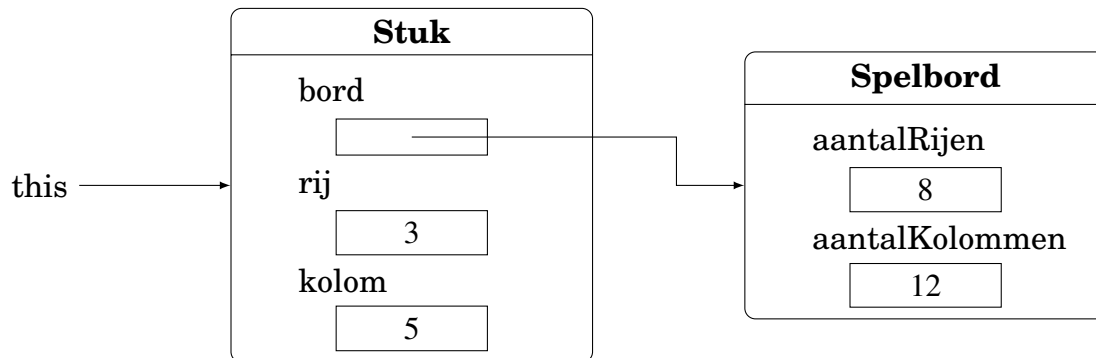
```
public class Buiten {  
    ...  
    public class Binnen {  
        ...  
    }  
    ...  
}
```

(De binnenklasse kan ook **private** zijn, of zelfs **protected**.)

We illustreren de betekenis van deze binnenklassen met behulp van een voorbeeld: we introduceren de klassen *Stuk* en *Spelbord* waarmee stukken op een spelbord worden voorgesteld. Elk spelbord heeft een aantal rijen en kolommen en deze aantallen zijn opgeslagen in de velden *aantalRijen* en *aantalKolommen*

van de klasse *Stuk*. Elk stuk heeft een veld *bord* dat naar het spelbord verwijst waarop dit stuk zich bevindt, en (o.a.) de velden *kolom* en *rij* die aangeven waar het stuk zich op het bord bevindt.

Onderstaand objectdiagram schetst een stuk dat zich op rij 3 en kolom 5 bevindt van een spelbord van 8 rijen en 12 kolommen.



De volgende methode (uit de klasse *Stuk*) verplaatst het stuk één positie naar onder, tenzij het stuk zich reeds aan de onderkant van het bord bevindt:

```

public void naarOnder () {
    if (rij + 1 < bord.getAantalRijen()) {
        rij ++;
    }
}

```

Omdat stukken nauw zijn verbonden met het spelbord waarop ze zich bevinden, hebben we hier de optie om *Stuk* als binnenklasse te definiëren van *Spelbord*.

Merk op hoe de definitie van *naarOnder* hierdoor verandert:

```

public class Spelbord {

    private int aantalRijen;

    private int aantalKolommen;

    ...

    public class Stuk {

```

```

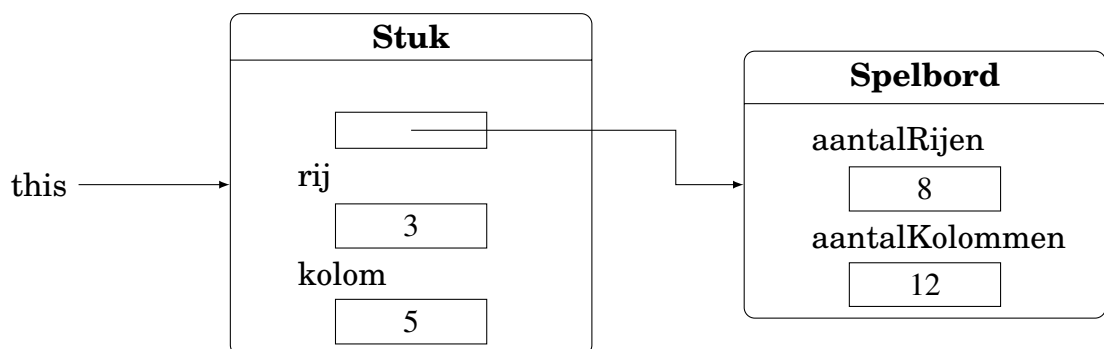
private int rij;

private int kolom;

public void naarOnder () {
    if (rij + 1 < aantalRijen) {
        rij ++;
    }
}
...
}

```

Een object van een (niet-statische) binnenklasse heeft automatisch toegang tot alle velden van de buitenklassen, zelfs als ze **private** zijn. Het corresponderende objectdiagram ziet er bijna identiek uit als in het vorige geval:



De verwijzing van het stuk naar zijn spelbord is nu *impliciet*: je hebt geen veld *bord* meer nodig om de velden en methodes van het corresponderende spelbord aan te spreken<sup>3</sup>.

Wanneer je een nieuw *Stuk*-object aanmaakt in een methode uit de klasse *Spelbord*, dan wordt de impliciete referentie ingevuld met het spelbord waarop die methode werd aangeroepen. Doe je dit vanuit een andere klasse in je programma, dan kan je het spelbord expliciet opgeven met de volgende notatie:

<sup>3</sup>In de zeldzame gevallen waar je toch een expliciete referentie nodig hebt — bijvoorbeeld wanneer binnen- en buitenklasse velden hebben met dezelfde naam — kan je de notatie *Spelbord.this.aantalRijen* gebruiken. Dit kan echter enkel vanuit de binnenklasse *Stuk* zelf.



```

Spelbord spelbord = new Spelbord(...);
Stuk stuk = spelbord.new Stuk(...);

```

In de praktijk komt dit echter weinig voor (en wijst dan vaak op een slecht software-ontwerp).

Als tweede voorbeeld van een binnenklasse tonen we een implementatie van de interface *Vertaler* van blz. 13. De buitenklasse heet *Vertaling* en bevat een gesofisticeerde methode waarmee een woord van een gegeven taal wordt vertaald naar een andere taal:

```

private String vertaalVanNaar (String woord,
                               String vanTaal, String naarTaal)

```

We willen nu deze methode gebruiken, samen met de methode *vertaalLijst* van blz. 13, om een lijst van woorden te vertalen van het Nederlands naar het Frans. We hebben dus een klasse nodig die de interface *Vertaler* implementeert en de bewuste vertaling uitvoert. Waarom niet een binnenklasse?

```

public class Vertaling {
    ...
    private class VertalerNlFr implements Vertaler {
        public String vertaal(String woord) {
            return vertaalVanNaar (woord, "NL", "FR");
        }
    }

    // Vertaalt een woordenlijst van Nederlands naar Frans
    public List<String> vertaalNlFr (List<String> woordenLijst) {
        return vertaalLijst (woordenLijst, new VertalerNlFr());
    }
}

```

Merk op dat de binnenklasse *VertalerNlFr* rechtstreeks gebruikt maakt van de methode *vertaalVanNaar* van de buitenklasse.

Een situatie zoals bovenstaande zullen we verder nog regelmatig tegenkomen: een binnenklasse die alleen maar dient om een bepaalde interface te implementeren, essentieel maar uit enkele lijnen bestaat en waarvan er slechts op één plaats in het programma een object wordt aangemaakt.

Voor deze toepassing biedt Java een bijzondere soort binnenklasse — *anonieme binnenklasse* genaamd. De speciale notatie die hiervoor gebruikt wordt, combineert de definitie van de klasse onmiddellijk met het aanmaken van een object van die klasse en doet dit zonder dat je een naam voor die klasse hoeft te verzinnen. (Een naam heeft hier toch weinig nut aangezien je in de rest van het programma nooit naar die klasse zult verwijzen.)

Met een anonieme binnenklasse ziet de vertaalfunctie van hierboven er zo uit:

```
public List<String> vertaalNlFr (List<String> woordenLijst) {  
    return vertaalLijst(woordenLijst, new Vertaler() {  
        public String vertaal(String woord) {  
            return vertaalVanNaar (woord, "NL", "FR");  
        }  
    });  
}
```

Dit ziet er op het eerste gezicht uit als de creatie van een nieuw *Vertaler*-object met een gewone oproep van **new** — alhoewel **new** in Java nooit met een interface kan gebruikt worden. Let echter op de accolades die er onmiddellijk op volgen. Tussen die accolades staat er een volledige klassendefinitie.

De anonieme klasse die hier wordt aangemaakt, voldoet vanzelf aan de interface *Vertaler*. In plaats van op interfaces, kan je een anonieme klasse ook baseren op een gewone (eventueel abstracte) klasse. De anonieme klasse wordt dan automatisch een uitbreiding van die basisklasse.

In de volgend paragraaf zien we dat er een nog kortere notatie bestaat voor dit soort anonieme klassen.

## 1.5. Functionele interfaces en lambda's

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.mails*. ]

Er zijn in versie 8 van Java heel wat nieuwigheden geïntroduceerd en één van de belangrijkste daarvan is de introductie van zogenaamde '*lambda's*' — een verkorte notatie voor anonieme binnenklassen met slechts één methode.

We illustreren dit concept aan de hand van een voorbeeld. We willen een lijst van e-mailberichten sorteren op verschillende manieren, volgens onderwerp, volgens

naam van de afzender, volgens tijdstip waarop het bericht is verstuurd, enz. We gebruiken de record-klasse *Mail* uit §1.2 om een e-mailbericht voor te stellen.

Zoals je weet<sup>4</sup> heeft de klasse *Collections* een methode *sort* waarmee je lijsten kunt sorteren.

```
public static void sort(List<T> list)
```

Deze methode werkt echter enkel voor types *T* die de interface *Comparable<T>* implementeren. Een dergelijke interface implementeren heeft echter enkel zin wanneer de objecten een ‘natuurlijke’ volgorde bezitten, zoals bij getallen of tijdstippen of strings. Voor *Mail*-objecten zijn er echter verschillende zinvolle manieren om ze te rangschikken.

Voor dergelijke gevallen voorziet *Collections* een tweede sorteermethode:

```
public static <T> void sort(List<T> list, Comparator<T> c)
```

en ook *List* heeft een methode die hetzelfde doet:

```
public void sort(Comparator<E> c)
```

Beide methodes dienen om een lijst te sorteren en gebruiken daar bij een hulpobject, *comparator* genoemd, die het sorteeralgoritme toelaat om van twee objecten de volgorde te bepalen.

Om dus onze lijst van e-mailberichten te sorteren, schrijven we

```
List<Mail> list = ...  
list.sort(mailComparator);
```

waarbij *mailComparator* een object is dat de volgorde tussen twee mail-objecten kan bepalen. We kunnen een comparator schrijven die twee e-mailberichten vergelijkt aan de hand van hun afzender, een andere comparator die in de plaats het onderwerp gebruikt, of het tijdstip, enz.

Een comparator voor objecten van het type *T* moet voldoen aan de interface *Comparator<T>*. In de praktijk betekent dit dat hij de volgende methode moet implementeren:

---

<sup>4</sup>Zie cursus Datastructuren en Algoritmen 1.

```
int compare(T o1, T o2);
```

Deze methode moet twee objecten van het type *T* vergelijken en een negatief getal teruggeven wanneer het eerste object kleiner is dan het tweede (volgens de volgorde die typisch is voor deze specifieke comparator), een positief getal wanneer het eerste object groter is dan het tweede, en 0 wanneer ze als gelijk moeten beschouwd worden. (Merk op, twee objecten kunnen door de comparator als gelijk beschouwd worden zonder daarom gelijk te zijn — e-mailberichten met dezelfde afzender bijvoorbeeld.) De methode *compare* heeft m.a.w., hetzelfde contract als de gelijknamige methode in *Comparable*, maar heeft een andere vorm.

Voor ons voorbeeld moeten we dus (o.a.) een comparatorklasse schrijven die twee mail-objecten met elkaar vergelijkt aan de hand van het tijdstip waarop ze verstuurd zijn. Van deze klasse moeten we één enkel object maken. Paragraaf §1.4 leert ons dat dit een uitstekende gelegenheid voor een *anonieme binnenklasse*:

```
list .sort(new Comparator<Mail>() {  
    @Override  
    public int compare(Mail o1, Mail o2) {  
        return o2.time().compareTo(o1.time());  
    }  
});
```

(Merk op dat we hier *time()* gebruiken en niet *getTime()*. *Mail* is immers een record-klasse.)

De notatie voor anonieme binnenklassen is al een heel stuk korter dan voor reguliere klassen maar er is nog een korter alternatief, op voorwaarde dat de anonieme klasse een zogenaamde functionele interface implementeert.

We noemen een interface een *functionele interface* wanneer zij slechts één methode bevat. *Comparator* is een functionele interface omdat ze enkel de methode *compare* bevat<sup>5</sup>.

Het comparatorvoorbeeld kan nu tot één lijn worden gereduceerd:

```
list .sort( (o1,o2) -> o2.time().compareTo(o1.time()) );
```

---

<sup>5</sup>De werkelijke regel is iets meer gecompliceerd: *Comparator* bevat ook de methode *equals*, maar die wordt niet meegerekend omdat dit een methode is van *Object*. Bovendien kunnen in de meer recente versies van Java interfaces ook klassenmethoden en ‘default’methoden bevatten — en *Comparator* heeft er daarvan een hele boel — maar ook zij tellen niet mee.

Het argument van *sort* wordt een ‘*lambda*’ genoemd<sup>6</sup> of ook wel een *anonieme functie*.

De anonieme binnenklasse uit het vertalervoorbeeld van blz. 18 kan op dezelfde manier mooi afgekort worden tot

```
public List<String> vertaalNlFr3 (List<String> woordenLijst) {  
    return vertaalLijst(woordenLijst,  
                        woord -> vertaalVanNaar (woord, "NL", "FR"));  
}
```

(Merk op dat we de haakjes mogen weglaten rond de parameter van een lambda als die slechts één parameter bezit.)

Lambda's vragen heel wat werk van de compiler. Hij moet uit één lijn immers de volledige anonieme klasse kunnen reconstrueren. Omdat *list* slechts één sort-methode heeft, weet de compiler dat het argument van *sort* een comparator moet zijn. Omdat *Comparator* een functionele interface is, weet hij dat hij een methode *compare* nodig heeft, in dit geval met twee parameters van het type *Mail*, en dit bepaalt het type van *o1* en *o2*.

Soms is de context voor de compiler niet zo duidelijk, en moeten we hem een handje helpen. We kunnen bijvoorbeeld het type van *o1* en *o2* expliciet opgeven

```
list.sort(  
    (Mail o1, Mail o2) -> o2.time().compareTo(o1.time())  
);
```

en in hoge nood kunnen we zelfs het type van de functionele interface aanduiden met behulp van een cast:

```
list.sort(  
    (Comparator<Mail>)  
    (Mail o1, Mail o2) -> o2.time().compareTo(o1.time())  
);
```

(wat in dit voorbeeld echter totaal overbodig is).

---

<sup>6</sup>De term komt uit de zogenaamde *lambda-calculus*, een belangrijk concept uit de theoretische informatica.

Het corpus van een lambda hoeft niet noodzakelijk één enkele lijn te zijn. Onderstaande lambda is bijvoorbeeld een comparator die eerst de namen van de afzenders van twee e-mailberichten met elkaar vergelijkt, en wanneer beide namen dezelfde zijn, de volgorde bepaalt aan de hand van het onderwerp:

```
list.sort(
    (o1, o2) -> {
        int cmp = o1.sender().compareTo(o2.sender());
        if (cmp == 0) {
            return o1.subject().compareTo(o2.subject());
        } else {
            return cmp;
        }
    }
);
```

Net zoals we je zouden aanraden om geen anonieme binnenklassen te schrijven waarvan de methodes meer dan enkele lijnen lang zijn, is het ook beter om lambda's altijd kort te houden. Moet je lambda meer doen dan enkele eenvoudige opdrachten, dan kan je die best in een afzonderlijke methode verpakken.

Bekijk bijvoorbeeld de volgende (klassen)methode die twee e-mailonderwerpen met elkaar vergelijkt door in de eerste plaats elke 'Re: ' erin te negeren, en pas wanneer de e-mailonderwerpen voor de rest gelijk zijn, ook naar hun lengte te kijken:

```
public class Comparators {

    public static int compareSubject(Mail mail1, Mail mail2) {

        String subject1 = mail1.subject();
        while (subject1.startsWith("Re: ")) {
            subject1 = subject1.substring(4);
        }

        String subject2 = mail2.subject();
        while (subject2.startsWith("Re: ")) {
            subject2 = subject2.substring(4);
        }
    }
}
```

```

    int cmp = subject1.compareTo(subject2);
    if (cmp == 0) {
        return mail1.subject().length() - mail2.subject().length();
    } else {
        return cmp;
    }
}
...
}

```

Willen we deze methode in een lambda gebruiken, dan schrijven we

```
list.sort( (o1, o2) -> Comparators.compareSubject(o1,o2) );
```

en omdat dit soort lambda's heel courant voorkomt, kunnen ze nog verder worden ingekort, op de volgende manier:

```
list.sort( Comparators::compareSubject );
```

Dit heet een *methodereferentie* (Engels: *method reference*.)

Ook in de standaardbibliotheken vinden we veel toepassingen van lambda's. We geven hieronder één voorbeeld, in paragrafen §1.6 en §1.7 volgen er meer.

Vaak is het zo dat we in een comparator twee complexe objecten vergelijken door er eerst bepaalde informatie uit te extraheren en deze dan te vergelijken: voor een *Person*-object bekijken we bijvoorbeeld enkel de naam, voor een *Mail*-object enkel het tijdstip. Als je weet hoe je de relevante gegevens moet extraheren dan is het schrijven van de gepaste comparator slechts routine. En wat routine is, kan evengoed door de computer zelf gedaan worden...

Om die reden bestaat er een klassenmethode *Comparator.comparing* met de volgende signatuur:

```
public Comparator<T> comparing(Function<T,U> extractor);
```

Deze methode maakt een nieuwe comparator (voor objecten van het type *T*) aan de hand van een gegeven object *extractor* dat weet hoe het uit een object van het type *T* een object van het type *U* moet halen. (De klasse *U* moet wel 'comparable' zijn.)

*Function* is een functionele interface uit de standaardbibliotheek die gebruikt wordt voor lambda's die een parameter van het type *T* omzetten naar een parameter van het type *U*. Willen we dus twee e-mailberichten vergelijken aan de hand van hun tijdstippen, dan hebben we een lambda nodig die uit een e-mailbericht het tijdstip haalt:

```
list.sort( Comparator.comparing (m -> m.time()) )
```

En opnieuw kunnen we een kortere notatie gebruiken:

```
list.sort( Comparator.comparing (Mail::time) )
```

(De dubbele dubbelepuntnotatie werkt dus ook voor objectmethoden.)

Nu rangschikken we de berichten van oud naar nieuw. We gebruiken de methode *reverse* om dit om te keren:

```
list.sort( Comparator.comparing (Mail::time).reverse() )
```

De methode *reverse* zet een comparator om naar een nieuwe comparator waarvan het resultaat van de *compare*-methode het omgekeerde is als bij de originele comparator. (Dit is een zogenaamde *default methode* van de interface *Comparator* en elke comparator krijgt deze automatisch cadeau.)

## 1.6. Lambda's in de collectiebibliotheek

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.extra.* ]

Er zijn heel wat bibliotheekklassen die van lambda's gebruik maken. Hierboven zagen we dit reeds in de context van comparatoren, maar ook in de collectiebibliotheek (lijsten en maps) zijn er een aantal nuttige methodes die functionele interfaces gebruiken voor hun parameters.

Je herinnert je ongetwijfeld nog dat het verwijderen van elementen uit een lijst niet zo evident is, omdat je aan een lijst niets mag veranderen terwijl je ze overloopt. Dit kan je oplossen door een *iterator* te gebruiken, maar de resulterende programmacode is niet altijd zeer leesbaar. Gelukkig bestaat er nu voor elke collectie de methode *removeIf(..)* die als parameter een *predicaat* neemt dat aangeeft



welke elementen uit een lijst mogen worden verwijderd — m.a.w., een lambda die een lijstelement als parameter neemt en een boolean als waarde.

In onderstaand voorbeeld nemen we een lijst van gehele getallen en verwijderen er alle *oneven* getallen uit:

```
List<Integer> list = ...;  
list.removeIf(i -> i % 2 == 1);
```

Een andere methode die voor elke collectie bestaat, heet *forEach* en doet ongeveer hetzelfde als een *for each*-lus. Hieronder drukken we alle elementen van een lijst onder elkaar af

```
list.forEach(System.out::println);
```

Toegegeven, de methode *forEach* biedt niet zoveel voordeel t.o.v. de *for each*-lus zelf, behalve misschien dat ze wel heel erg kort noteert wanneer we een methode-referentie kunnen gebruiken zoals hierboven. (Merk trouwens op dat de methode *forEach* niet kan toegepast worden op een array.)

Nog een methode die ons toelaat om het gebruik van een iterator te vermijden, is de volgende:

```
list.replaceAll(i -> i/2);
```

Hier vervangen we elk getal in de lijst door de helft van dit getal. (Deze methode bestaat enkel voor lijsten, niet voor verzamelingen.)

Ook aan de klasse *Map* zijn heel wat methodes toegevoegd. We vermelden er hier slechts enkele.

De methode *forEach* loopt over de ganse inhoud van de map en past er een bepaalde actie op toe. Dit keer wordt de actie voorgesteld door een functie met *twee* parameters, de sleutel en de waarde:

```
Map<String,Integer> map = ...;  
map.forEach( (k,v) -> System.out.println (k + " : " + v) )
```

Een andere methode van *Map* met interessante toepassingen is *computeIfAbsent*. Zoals de naam suggereert, kan je hiermee een element uit de map opvragen en aangeven hoe het element moet worden berekend als het zich nog niet in de map

bevindt. Is de berekende waarde bovendien verschillend van **null** dan wordt ze automatisch in de map ingevoegd.

Dit kan rechtstreeks toegepast worden bij het zogenaamde *caching* — het opslaan van resultaten van langdurige bewerkingen zodat wanneer hetzelfde resultaat nog eens nodig is, het gewoon kan worden opgevraagd en niet meer opnieuw hoeft worden berekend — maar wij geven een voorbeeld van een andere manier waarop deze methode kan worden gebruikt.

Stel dat we van elk woord in een gegeven lijst willen weten hoeveel keer het in die lijst voorkomt? Dit kan je doen door de woorden als sleutel te gebruiken in een map, en het aantal keer dat het woord voorkomt als waarde. Je vult de map dan bijvoorbeeld op de volgende manier op.

```
Map<String,Integer> map = ...;
for (String word: list) {
    if (map.containsKey(word)) {
        map.put(word, map.get(word) + 1);
    } else {
        map.put(word, 1);
    }
}
```

Merk op dat je hier twee gevallen afzonderlijk moet behandelen: al naar gelang je het woord al eerder hebt ontmoet, of niet. Met *computeIfAbsent* schrijf je in de plaats het volgende:

```
Map<String,Integer> map = ...;
for (String word: list) {
    map.put(word, map.computeIfAbsent(word, k -> 0) + 1);
}
```

(Merk op dat de waarde die we hier berekenen, niet afhangt van de sleutel *k*. In een dergelijk geval kan je vaak beter de methode *getOrDefault* van *Map* gebruiken. Meer informatie vind je in de elektronische documentatie.)

Een andere situatie waarvoor *computeIfAbsent* nuttig kan zijn, is wanneer je een map hebt die met elke sleutel niet één maar mogelijks meerdere waarden laat overeenkomen. Waar we in het bovenstaande voorbeeld de mapwaarde met 0 initialiseerden, gebruik je nu een lege lijst (of verzameling).

```

private Map<String,List<Person>> map;

public void registerValue(String key, Person value) {}
    map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);
}

```

(Dit kan je niet in één lijn oplossen met *getOrDefault*.)

## 1.7. Java streams

Streams<sup>7</sup> bieden een alternatief voor collecties om gegevensreeksen voor te stellen. Je hebt inderdaad streams van gehele getallen, streams van strings, streams van objecten van allerlei types, enz., maar waar we bij collecties in de eerste plaats denken aan hoe gegevens worden opgeslagen en efficiënt kunnen worden opgevraagd, staan bij streams vooral de bewerkingen centraal die op de elementen worden uitgevoerd.

Voor de eenvoud kan je een stream beschouwen als een object waaraan je een aantal elementen één voor één kan opvragen<sup>8</sup>. Een stream gedraagt zich een beetje als een lijst die je alleen met een *for each*-lus kunt bewerken. De lijst wordt echter niet noodzakelijk volledig in het geheugen opgeslagen, en in de plaats van een *for each*-lus gebruik je allerlei methodes met functionele parameters (lambda's) om aan te geven wat er met de elementen moet gebeuren. Een voorbeeld zal dit wellicht verduidelijken.

In onderstaand fragment bepalen we de som van de opeenvolgende kwadraten  $1^2$ ,  $2^2$ , ...,  $10^2$ . Hiertoe creëren we een stream die de getallen 1, 2, ..., 10 genereert, transformeren die tot een nieuwe stream met de kwadraten van die getallen en bepalen we tenslotte de som van de elementen uit de resulterende stream:

```

int som = IntStream.range(1,11).map(i -> i*i).sum();

```

Dit fragment illustreert de typische manier waarop we streams gebruiken — in een zogenaamde *pijplijn*. Een pijplijn bestaat uit drie delen.

<sup>7</sup>Niet te verwarren met de invoer/uitvoer-streams die we behandelen in §3.1. Wanneer er verwarring mogelijk is, noemen we de streams uit dit hoofdstuk *Java streams*.

<sup>8</sup>Dit is een aanzienlijke vereenvoudiging: één van de eigenschappen van de nieuwe streams-bibliotheek is precies dat er ook *parallelle* streams bestaan waarbij er meerdere elementen tegelijk worden verwerkt. We gaan daar in deze inleidende nota's echter niet verder op in.

- Het eerste deel is de *bron* van de pijplijn, hier aangemaakt met de methode *range* van *IntStream*.
- Daarna volgen nul of meer *intermediaire* bewerkingen. Deze zetten telkens een inkomende stream om naar een uitgaande stream door elementen te transformeren en/of te selecteren. Hier gebruiken we de methode *map* om elk element van de stream om te zetten in zijn kwadraat.
- De laatste methode-oproep in een stream-pijplijn is een *terminale* of *eind*-bewerking. Deze produceert een eindresultaat (een geheel getal, een string, een lijst, een verzameling, ..., maar niet een stream) of doet iets met de elementen dat geen resultaat oplevert (ze afdrukken bijvoorbeeld). Hier bepalen we de som van de elementen met de methode *sum*.

Er bestaan verschillende manieren om een bron aan te maken. Heel vaak creëert men een stream vanuit een bestaande collectie, met de methode *stream()*. In onderstaand fragment gebruiken we dit bijvoorbeeld om van een lijst van woorden enkel deze af te drukken die minder dan 4 letters hebben.

```
List<String> woorden = ...;
woorden.stream()
    .filter (w -> w.length() < 4)
    .forEach(System.out::println);
```

De intermediaire bewerking *filter* maakt een nieuwe stream aan met alleen de element uit de originele stream waarvoor het opgegeven predicaat een **true** teruggeeft. (De eindbewerking *forEach* doet wat je ervan zou verwachten.)

Merk op dat *filter* niet eerst een nieuwe lijst aanmaakt met daarin alle woorden van lengte kleiner dan 4 om ze daarna door te geven aan *forEach*. De streams-bibliotheek doet immers haar best om dit soort geheugengebruik te beperken: elk woord dat door *filter* wordt doorgelaten, wordt meteen doorgegeven aan *forEach* (en dus afgedrukt) vóór het volgende woord door *filter* wordt bekeken.

Er zijn nog heel wat andere methodes om nieuwe streams aan te maken: met *Arrays.stream* en *Stream.of* maak je een stream op basis van een array, de methode *IntStream.range* die we hierboven al hebben gebruikt, bouwt een stream van opeenvolgende gehele getallen, de klasse *Random* heeft een methode *ints* voor een stream van willekeurige getallen, enz. Je kan ook een stream opvragen die de opeenvolgende lijnen van een bestand teruggeeft (cf. §3.5).

De meest gebruikte intermediaire bewerking is wellicht *map*, waarmee je de stream element per element kan omzetten naar een andere stream. Een map kan

een element ook omzetten naar een element van een ander type dan het oorspronkelijke, bijvoorbeeld een stream van personen naar een stream van voornamen.

Bij omzetten van objecten van en naar primitieve types moeten we opletten. Hieronder drukken we 10 willekeurige cijfers af, als Nederlandse woorden:

```
String[] NAMEN = { "nul", "één", "twee", "drie", "vier",  
                  "vijf", "zes", "zeven", "acht", "negen" };  
...  
new Random().ints(20)  
    .mapToObj(i -> NAMEN[Math.abs(i) % 10])  
    .forEach(System.out::println);
```

Merk op dat we hier niet *map* gebruiken, maar *mapToObj*. De reden hiervoor is dat Java onderscheid maakt tussen verschillende soorten streams. De interface *Stream* wordt gebruikt voor streams waarvan de elementen objecten zijn (t.t.z., bij referentietypes). Voor elementen van een primitief type behoren streams tot de interface *IntStream*, *DoubleStream*, enz.

In bovenstaand voorbeeld beginnen we met een *IntStream*. We willen die omzetten naar een (object-) *Stream*. De methode *map* zet een stream altijd om naar hetzelfde type (een *IntStream* naar een *IntStream*, een *Stream* naar een *Stream*, maar in dit laatste geval mogen de types van de elementen wel verschillend zijn) en vandaar dat we een aangepaste methode *mapToObj* nodig hebben.

Omgekeerd zijn er ook methodes *mapToInt*, *mapToDouble*, ... wanneer we elementen van een stream willen omzetten naar elementen van een primitief type.

We tellen bijvoorbeeld de lengtes op van alle strings in een gegeven lijst:

```
List<String> woorden = ...  
int totaal = woorden.stream().mapToInt(String::length).sum();
```

Een ander veelgebruikte intermediaire bewerking is *filter* waarmee je elementen uit een lijst kan verwijderen — of beter gezegd, selecteren, want de lambda die je als argument van *filter* meegeeft, bepaalt welke elementen er in de nieuwe stream worden opgenomen en niet welke er worden weggegooid (cf. blz. 28).

Ook interessant is *flatMap*. Hiermee zet je elk element uit de oorspronkelijke stream om naar nul of meer elementen die dan allemaal achter elkaar in de resulterende stream worden opgenomen.

Als voorbeeld tonen we hoe je een stream van lijnen kan omzetten naar een stream van woorden waaruit die lijnen bestaan. Als parameter neemt *flatMap*

een functie (hier *words*) die één lijn opsplijt en die teruggeeft als een (korte) *stream* van woorden.

```
private void Stream<String> words(String line) {  
    return Arrays.stream(line.split(" "));  
}
```

```
Stream<String> linesStream = ...  
lineStream.flatMap(this::words).forEach(...)
```

Naast *forEach* ondersteunt de streams-bibliotheek nog heel wat andere eindbewerkingen. We kunnen die grosso modo onderverdelen in twee groepen: de *reductiebewerkingen* en de *collectiebewerkingen*.

Zoals de naam al aangeeft, reduceren reductiebewerkingen de volledige stream tot één enkel resultaat. Voorbeelden van reductie-bewerkingen zijn *count* die het aantal elementen van de stream teruggeeft, of *sum*, *min*, *max* die de som, minimum en maximum van alle stream-elementen bepaalt. (Deze laatste werken enkel bij *IntStream* en *DoubleStream*.)

Er bestaat ook een algemene *reduce*-methode waarmee je zelf kan bepalen hoe je de stream-elementen reduceert. In het volgende voorbeeld zetten we een stream van woorden om tot één lange string die alle woorden achter elkaar bevat, gescheiden door mintekens.

```
Stream<String> wordsStream = ...;  
String result = wordsStream.reduce(" ", (t,e) -> t + "-" + e);
```

(Merk op dat dit resultaat ook nog vooraan een minteken zal bevatten, dat we dan achteraf kunnen weghalen met *result.substring(1)*.)

De methode *reduce* neemt als (tweede) parameter een functie die aan de hand van een tussenresultaat *t* en een element *e* uit de stream een nieuw tussenresultaat berekent. De stream past deze functie herhaaldelijk toe totdat het laatste stream-element is verwerkt, en dat wordt het tussenresultaat het eindresultaat. Om dit mogelijk te maken, moet de stream van een initieel ‘tussen’resultaat kunnen starten. Dit geef je mee als eerste parameter van *reduce*.

Je kan de methode *sum* ook nabootsen met *reduce*, op de volgende manier

```
int sum = intStream.reduce(0, (t,e) -> t + e);
```

Collectiebewerkingen nemen een volledige stream en verzamelen haar elementen in één of andere datastructuur (een lijst, een verzameling, ...). Je zou deze bewerkingen ook als een reductiebewerkingen kunnen implementeren, maar dit is niet altijd efficiënt omdat je telkens nieuwe collecties moet aanmaken als opeenvolgende tussenresultaten. In plaats daarvan gebruikt de streams-bibliotheek één enkele vaste collectie die stap per stap wordt aangevuld.

De meest eenvoudige collectiebewerkingen zijn *toArray* en *toList* waarmee je de elementen van een stream verzamelt in een array of een lijst. Hieronder gebruiken we de *flatMap* van blz. 30 om uit een lijst van lijnen een lijst van alle woorden te distilleren:

```
public List<String> listOfWords(List<String> lines) {  
    return lines.stream().flatMap(this::words).toList();  
}
```

Meer ingewikkelde collectiebewerkingen doe je met de methode *collect* die als parameter een *Collector*-object vraagt. Er bestaan heel wat standaardcollectors die hiermee gebruikt kunnen worden. Hier beperken we ons tot één voorbeeld, de *joining*-collector waarmee strings uit een stream tot één lange string kunnen worden samengevoegd, gescheiden door een gegeven string:

```
String result = wordStream.collect(Collectors.joining("-"));
```

Achter de schermen wordt hier een *StringBuilder* gebruikt, wat dit een stuk (geheugen)efficiënter maakt dan onze eerdere oplossing met een reductiebewerking.

We hebben met deze voorbeelden slechts een klein tipje van de sluier opgelicht over wat je allemaal met streams kan doen. Meer informatie vind je in de elektronische documentatie.

We eindigen met een kleine waarschuwing: overdrijf niet met het gebruik van streams. Het is inderdaad zo dat heel veel van wat je anders met lussen en **if**-opdrachten zou doen, nu ook kan programmeren met streams, filters en collectors. Het blijft echter belangrijk dat je code nog leesbaar is, ook voor programmeurs die niet zo goed op de hoogte zijn van de laatste finesses in functioneel programmeren. Java is en blijft in de eerste plaats een objectgerichte programmeertaal.

## 1.8. Klassen, interfaces en methodes met typeparameters

Generieke klassen en interfaces, zoals *List<..>* en *HashMap<...,..>*, zijn niet meer weg te denken uit onze Java-programma's. In deze paragraaf tonen we hoe je zelf een dergelijke klasse of interface kan programmeren die één of meerdere types als parameters neemt.

Als voorbeeld ontwerpen we een klasse *Optional<T>* waarmee je ofwel een waarde van de klasse *T* kan voorstellen, ofwel het feit dat een dergelijke waarde ontbreekt<sup>9</sup>. Het type *Optional<..>* kan je bijvoorbeeld gebruiken als retourwaarde van een zoekoperatie om aan te geven dat je wat je zocht niet hebt gevonden:

```
public Optional<Persoon> persoonMetNaam (String naam)
```

De belangrijkste methodes van *Optional* zijn *isPresent*, waarmee je kan opvragen of een *Optional*-object een echte waarde voorstelt (indien **true**) of het feit dat er geen echte waarde is (indien **false**), en *get* waarmee je de waarde kan ophalen (en die een uitzondering opgooit als er geen echte waarde is).

De implementatie van een generieke klasse verschilt heel weinig van die van een gewone klasse, je plaatst enkel de typeparameter bij de naam van de klasse, tussen scheve haken. (Meestal gebruikt men voor de typeparameter een naam die uit één hoofdletter bestaat, zoals *S*, *T* of *U*.)

```
public class Optional<T> {  
  
    private boolean present;  
    private T value;  
  
    public Optional () { // een 'lege' optional  
        this.present = false;  
    }  
}
```

---

<sup>9</sup>Er bestaat een klasse *Optional* in de standaardbibliotheek (pakket *java.util*) met dezelfde functionaliteit als hier maar met bijkomende methodes en een implementatie die efficiënter is dan de onze. (Opmerking: de constructoren die we hier introduceren zijn privé in *java.util.Optional*. Meer informatie vind je in elektronische documentatie.)

Je vraagt je misschien af wat het nut is van deze klasse. Je kan toch ook **null** gebruiken om aan te geven dat een waarde ontbreekt? Tegenwoordig bestaat de tendens om het gebruik van **null** te vermijden, om diverse redenen waar we nu niet dieper op ingaan — tik anders eens 'Why is null bad?' in je favoriete zoekmachine.



```

public Optional (T value) {
    this.value = value;
    this.present = true;
}

public boolean isPresent() {
    return present;
}

public T get () {
    if (present) {
        return value;
    } else {
        throw new NoSuchElementException("No value present");
    }
}
}

```

Binnenin de klasse kan je de typeparameter (hier *T*) gebruiken zoals elke andere klasse — of liever gezegd interface, want ‘**new T()**’ kan je niet schrijven (zie verder). Je kan *T* gebruiken als type van een parameter in de hoofding van een methode, als retourtype van een methode, als type van een variabele of als argument van een ander generiek type, m.a.w., je kan bijvoorbeeld ook een *List<T>* gebruiken binnen in de geparametriseerde klasse. (Arrays van het type *T[]* kan je echter beter vermijden, zie verder.)

We kunnen bijvoorbeeld onderstaande methode *filter* toevoegen aan *Optional* waarmee je een optional ‘leeg maakt’ tenzij de waarde aan een bepaalde eigenschap voldoet.

```

public Optional<T> filter (Predicate<T> predicate) {
    if (!present) {
        return this;
    } else if (predicate.test(value)) {
        return this;
    } else {
        return new Optional<>();
    }
}

```

(De interface *Predicate*<*T*>, uit *java.util*, heeft één methode *test* die een object van het type *T* als argument neemt en een **boolean** retourneert.)

Bij *Optional*<..*>*, net zoals bij de meeste generieke klassen die je in de praktijk al hebt ontmoet, zijn er geen restricties op wat je kan invullen als type-argument *T*. Tussen de scheve haken kan je elke klasse of interface plaatsen (maar *geen* primitieve types). Dit is echter niet altijd wat je wil.

Voor het volgende voorbeeld gebruiken we een interface *GameCharacter* om personages uit een spel voor te stellen. Deze interface wordt door verschillende klassen geïmplementeerd, bijv. *Wizard*, *Dwarf*, *Hobbit*, afhankelijk van het type personage. Deze interface bezit onder andere de methode *getName()* waarmee de naam van een personage wordt teruggegeven.

We wensen nu een nieuwe generieke klasse *Guild* te maken waarin we personages van een bepaald type kunnen opzoeken en registreren aan de hand van hun naam. Bij een *Guild*<*Wizard*> kan je enkel tovenaars registreren, bij een *Guild*<*Dwarf*> enkel dwergen, enz.

```
Guild<Wizard> wizardGuild = ...  
Wizard wizard = ...  
wizardGuild.register (wizard);
```

...

```
Optional<Wizard> opt = wizardGuild.get("Gandalf");
```

Op het eerste zicht lijkt de volgende code een goede implementatie van de generieke klasse *Guild*:

```
public class Guild<T> {  
  
    private Map<String,T> names;  
  
    public Guild () {  
        this.names = new HashMap<>();  
    }  
  
    public void register (T character) {  
        names.put (character.getName(), character); // compileert niet  
    }  
}
```

```

    public Optional<T> get (String name) {
        T character = names.get(name);
        if (character == null) {
            return new Optional();
        } else {
            return new Optional(character);
        }
    }
}

```

De compiler zal echter een foutmelding geven bij de definitie van de methode *register*, meer specifiek bij de oproep van *getName*.

Omdat het type *T* elke mogelijke klasse kan voorstellen, kunnen we niet garanderen dat de parameter *character* van *register* op dat moment wel degelijk een methode *getName* bezit en daarom signaleert de compiler hier een fout. Gelukkig laat Java ook toe om in de hoofding van een generieke klasse op te geven dat de parametertypes aan bepaalde voorwaarden moeten voldoen, een gegeven klasse uitbreiden of een gegeven interface implementeren.

Opdat onze *Guild*-klasse zou compileren, hoeven we enkel de hoofding te vervangen door

```

public class Guild<T extends GameCharacter> {

```

(Merk op dat we hier ‘**extends**’ moeten gebruiken, ook al is *GameCharacter* een interface.)

Doordat we deze restrictie opleggen aan *T* zal de methode *register* wel compileren: Java weet nu immers dat *T* aan de interface *GameCharacter* voldoet en dus met zekerheid een methode *getName* bevat.

Naast generieke klassen of interfaces laat Java ook toe om generieke *methodes* te definiëren. De volgende methode retourneert een willekeurige string uit een lijst van strings, tenzij de lijst leeg is<sup>10</sup>.

---

<sup>10</sup>In dit voorbeeld schrijven we een klassenmethode, maar ook gewone (instantie-)methodes kunnen generiek gemaakt worden.

```

public static Optional<String> getRandomElement (List<String> list) {
    if (list .isEmpty()) {
        return new Optional();
    } else {
        return new Optional(list.get(RG.nextInt(list.size())));
    }
}

```

Willen we in de plaats een willekeurige persoon uit een lijst van personen, of een willekeurige *Double* uit een lijst van kommagetallen, dan hoeven we in bovenstaande hoofding enkel maar de naam *String* te vervangen door het betreffende type. Om knip en plakwerk te vermijden kan je echter ook het volgende schrijven:

```

public static <T> Optional<T> getRandomElement (List<T> list) {
    ...
}

```

Merk op dat er nu een bijkomende ‘<*T*>’ moet staan vlak voor het retourtype van de methode

In enkele gevallen kan de notatie vereenvoudigd worden. De volgende methode, *verwijdert* een willekeurig element uit een gegeven lijst:

```

public static <T> void removeRandomElement (List<T> list) {
    if (! list .isEmpty()) {
        list .remove(RG.nextInt(list.size()));
    }
}

```

Hier wordt de parameter *T* slechts één keer gebruikt, namelijk in de parameterlijst van de methode. In dat geval kan je een vraagteken gebruiken als een soort jokerteken (Engels: *wildcard*):

```

public static void removeRandomElement (List<?> list) {
    ...
}

```

Jokertekens kunnen ook gecombineerd worden met **extends**, zoals in de volgende methode die de som kan bepalen van zowel een lijst van gehele getallen als van kommagetallen:

```

public static double sum(List<? extends Number> list) {
    double result = 0.0;
    for (Number number : list) {
        result += number.doubleValue();
    }
    return result;
}

```

Zowel *Integer* als *Double* voldoen namelijk aan de interface *Number* die onder andere een methode *doubleValue* declareert.

Onze voorbeelden zijn nog vrij eenvoudig, er is met generieke klassen en methodes heel wat meer mogelijk. In de praktijk zal je echter vooral generieke klassen *gebruiken* en het heel weinig nodig hebben om er zelf te ontwerpen en te implementeren, ook al omdat je vaak op een andere manier hetzelfde resultaat kunt bereiken.

Doordat deze begrippen niet van in het begin in Java zijn ingevoerd, zijn ze minder krachtig dan je misschien zou hopen — we hebben al vermeld dat je geen ‘**new** *T()*’ kan schrijven en dat werken met arrays van het type *T[]* allerlei problemen met zich meebrengt. Dit komt door de manier waarop de compiler ermee omgaat: parametertypes verdwijnen namelijk uit een klasse zodra ze gecompileerd is. Dit noemt men *type erasure*.

Concreet zal onze klasse *Optional*<*T*> na compilatie gewoon *Optional* heten, en zal de methode *get()* de volgende hoofding krijgen

```

public Object get()

```

met *Object* als retourtype, in plaats van *T*.

Als we dus **new** *T()* zouden mogen schrijven, dan zou dat uiteindelijk vertaald worden naar **new** *Object()*, wat een object zou opleveren van de verkeerde klasse. Ook een array van het type *T[]* wordt dan een array van het type *Object[]*, wat opnieuw problemen oplevert, aangezien subtypering niet overdraagt naar arrays — ook als is *String* een subtype van *Object*, toch is *String[]* geen subtype van *Object[]*.

Bij parametertypes met een **extends**-clausule wordt het type niet vervangen door *Object* maar door het type dat achter de **extends** staat. De methode *register* uit *Guild* krijgt dus uiteindelijk de volgende signatuur:

**public void** *register* (*GameCharacter* *character*)

Een ander vervelend bijeffect van *type erasure* treedt soms op bij methodes met dezelfde naam maar andere parametertypes. Veronderstel even dat we in een bepaalde klasse de volgende twee methodes willen definiëren

```
public List<T> multiplyVectors (List<T> links, List<Double> rechts) {  
    ...  
}
```

```
public List<T> multiplyVectors (List<Double> links, List<T> rechts) {  
    ...  
}
```

Na *type erasure* krijgen beide methodes exact dezelfde signatuur:

```
public List multiplyVectors (List links, List rechts)
```

en dat is niet toegelaten. Merk trouwens op dat dit niets te maken heeft met het feit dat *T* een parametertype is, dit geeft evengoed problemen wanneer we *T* vervangen door een concreet type, zoals bijvoorbeeld *Integer*.

## 2. Programmeren ‘zonder if’

Zoals je reeds in de inleiding hebt kunnen lezen, willen we je in deze nota's in de eerste plaats een aantal technieken bijbrengen voor het schrijven van middelgrote programma's op een objectgerichte manier en komt de implementatie van grafische gebruikersinterfaces slechts op de tweede plaats.

In dit korte hoofdstuk bespreken we daarom eerst enkele meer algemene programmeertechnieken en *good practices*.

[ Onderstaande voorbeelden vind je in `be.ugent.objprog.ifless.correct` en `be.ugent.objprog.ifless.wrong`. ]

### 2.1. Meervoudige selecties vermijden

Wellicht heb je in je programma's al vaak gebruik gemaakt van zogenaamde *meervoudige selecties* : ketens van **if-else**-opdrachten, of de meer compacte **switch**-opdrachten, waarmee je verschillende code uitvoert in verschillende gevallen. Inderdaad vormen deze structuren dikwijls een handig instrument voor de programmeur. Het veelvuldig gebruik ervan duidt echter heel vaak op een (objectgeïntereerd) programmaontwerp dat niet optimaal is.

In de komende pagina's illustreren we met behulp van een aantal voorbeelden hoe je meervoudige selecties kunt vermijden, en hoewel het misschien wat te ver gaat om te beweren dat een goed ontworpen programma nooit een meervoudige selectie hoeft te bevatten, loont het beslist de moeite om steeds een bewuste inspanning te leveren om het programmaontwerp zodanig aan te passen dat je die meervoudige selectie toch niet nodig hebt.

We maken onderscheid tussen twee gevallen: eerst bespreken we de meervoudige selecties waarbij het de bedoeling is om in de verschillende takken van de selectie andere gegevens te gebruiken, en daarna het meer ingewikkelde geval waarbij ook het gedrag van het programma van tak tot tak verschilt.

In een eerste voorbeeld willen we, al naargelang de dag van de week, het juiste dagloon bepalen van een arbeider in een bedrijf:

```
private double basisloon;
...

public double dagloon (int dag) {
    if (dag == 0) {
        return basisloon * 2.3; // zondag
    }
    else if (dag == 6) {
        return basisloon * 1.7; // zaterdag
    }
    else if (dag == 1) {
        return basisloon * 1.1; // maandag
    }
    else {
        return basisloon;
    }
}
```

Het enige verschil tussen de verschillende takken van deze meervoudige selectie is de factor waarmee het basisloon moet vermenigvuldigd worden. (Het **else**-geval correspondeert met een factor 1.0.)

Deze selectie van gegevens kan je vermijden door in de plaats een tabel (array) te gebruiken:

```
private double basisloon;
...

private static final double[] LOONFACTOR
    = { 2.3, 1.1, 1.0, 1.0, 1.0, 1.0, 1.7 };

public double dagloon (int dag) {
    return basisloon * LOONFACTOR[dag];
}
```

Dit maakt het trouwens meteen ook veel gemakkelijker om aanpassingen te doen. Stel dat je plots ook op woensdag meer loon moet uitbetalen dan hoef je in onze tweede versie slechts één getalletje te veranderen, terwijl je vroeger een volledige **if-else**-clausule moest toevoegen.



Soms is een tabel niet voldoende, maar gebruik je beter een (hash)map. In het volgende voorbeeld herkent het programma bepaalde benamingen van kleuren. Intern moet je echter de overeenkomstige objecten van het type *Color* gebruiken. Je hebt dus een functie nodig die de kleurennaam omzet naar het overeenkomstige object.

```
public Color stringToColor (String naam) {  
    if (naam.equals("geel")) {  
        return Color.YELLOW;  
    } else if (naam.equals("oranje")) {  
        return Color.ORANGE;  
    } else if {  
        ...  
    }  
    else {  
        throw new IllegalArgumentException ("Kleur onbekend");  
    }  
}
```

Met een map los je dit zo op:

```
private Map<String,Color> colorMap = new HashMap<>();  
...  
colorMap.put ("geel", Color.YELLOW);  
colorMap.put ("oranje", Color.ORANGE);  
...  
public Color stringToColor (String naam) {  
    Color result = colorMap.get (naam);  
    if (result == null) {  
        throw new IllegalArgumentException ("Kleur onbekend");  
    } else {  
        return result;  
    }  
}
```

(Merk op dat er nog steeds een **if** nodig is om het uitzonderingsgeval te herkennen.)

Op het eerste zicht lijkt de tweede versie bijna even lang als de eerste, en minstens zo ingewikkeld. Het is nu echter zeer gemakkelijk om een nieuwe kleurennaam toe te voegen of zelfs om het programma te ‘internationaliseren’, t.t.z. de

kleurennamen te laten afhangen van de taal van de gebruiker — gebruik gewoon een andere map.

Bovendien bestaat er in de meer recente versies van Java een kortere manier om een map aan te maken en alvast met enkele waarden in te vullen:

```
private Map<String, Color> colorMap = Map.of(
    "geel", Color.YELLOW,
    "oranje", Color.ORANGE,
    "rood", Color.RED
);
```

Een dergelijke map is wel van het type ‘alleen lezen’. Er bestaan ook gelijkaardige methodes *List.of(..)* om een lijst aan te maken waarvan je de inhoud al op voorhand kent.

Het volgende voorbeeld is minder eenvoudig. De methode *beweeg* beweegt een object in een opgegeven richting door zijn *x*- en *y*-coördinaten overeenkomstig aan te passen. We gebruiken het opsomtype *Windrichting* uit paragraaf §1.1 om een richting aan te geven.

We geven twee versies: een versie met **ifs** en een versie met **switch**<sup>1</sup>.

```
private int x;
private int y;

public void beweeg (Windrichting richting) {
    if (richting == OOST) {
        x ++;
    } else if (richting == ZUID) {
        y --;
    } else if (richting == WEST) {
        x --;
    } else { // richting moet nu NOORD zijn
        y ++;
    }
}
```

(We hebben een ‘static import’ van *Windrichting* gebruikt om de windrichtingen te kunnen afkorten tot *OOST*, *ZUID*, ..., i.p.v. *Windrichting.OOST*, ...)

---

<sup>1</sup>We gebruiken hier de nieuwere vorm van de **switch**-opdracht. Omdat de boodschap van dit hoofdstuk is om meervoudige selecties te vermijden, doen we dit zonder veel uitleg.

```

public void beweeg(Windrichting richting) {
    switch (richting) {
        case OOST -> x++;
        case ZUID -> y--;
        case WEST -> x--;
        case NOORD -> y++;
    }
}

```

(Bij een **switch** met opsomtypes is een *static import* niet nodig.)

In beide gevallen kan je een tabel gebruiken om de meervoudige selectie te vermijden, of liever gezegd: twee tabellen.

```

private int x;
private int y;

private static int[] DX = { 1, 0, -1, 0 };
private static int[] DY = { 0, -1, 0, 1 };

public void beweeg (Windrichting richting) {
    x += DX[richting.ordinal()];
    y += DY[richting.ordinal()];
}

```

Dit is al een stuk beter: het wordt nu heel wat eenvoudiger om bijvoorbeeld bijkomende richtingen (zuidoost, noordnoordwest, ...) in te voeren. In het vorige hoofdstuk hebben we echter afgeraden om *ordinal* te gebruiken en suggereren we om in de plaats een *EnumMap* te introduceren. In dit specifieke geval stellen we echter voor om het opsomtype *Windrichting* aan te passen op de volgende manier:

```

public enum Windrichting {

    OOST (1, 0),
    ZUID (0, -1),
    WEST(-1, 0),
    NOORD(0, 1);

    private int dx;
    private int dy;
}

```

```

private Windrichting(int dx, int dy) {
    this.dx = dx;
    this.dy = dy;
}

public int getDx() {
    return dx;
}

public int getDy() {
    return dy;
}
}

```

Met deze nieuwe implementatie van *Windrichting* wordt *beweeg* heel eenvoudig:

```

public void beweeg(Windrichting richting) {
    x += richting.getDx();
    y += richting.getDy();
}

```

In een ‘echt’ programma zou het wellicht nuttig zijn om de twee coördinaten *x* en *y* te groeperen tot één enkel (record-)object, bijvoorbeeld van een type *Vector*. Je kan dan een *getVector* toevoegen aan *Windrichting* die een dergelijke vector teruggeeft.

## 2.2. Meervoudige selectie van gedrag

Bij de voorbeelden hierboven bestonden de takken van de meervoudige selectie uit lijnen code die slechts in één enkele waarde van elkaar verschilden (in het derde voorbeeld was dit enigszins vermomd en ging het om twee waarden). Wat als dit niet het geval is, wat als elke tak compleet verschillende dingen doet? Kan je dan nog een meervoudige selectie vermijden? Laat dit nu precies zijn waar objectgericht programmeren goed in is!

Stel dat een programma een cirkel, driehoek of vierkant moet tekenen, afhankelijk van of nu de C-, de D- of de V-toets werd ingedrukt. De meest voor de hand

liggende manier om dit te doen, is wellicht de volgende:

```
if (toets == 'C') {  
    tekenCirkel ();  
} else if (toets == 'D') {  
    tekenDriehoek ();  
} else if (toets == 'V') {  
    tekenVierkant ();  
} ...
```

Dit is een meervoudige selectie waarin elke tak een ander gedrag vertoont.

Er zijn verschillende manieren om hetzelfde op een objectgerichte manier te verwezenlijken, telkens met het bijkomend voordeel dat het niet zo moeilijk is om nieuwe toetsen toe te voegen die nieuwe vormen tekenen. We kiezen er hier voor om interface (of bovenklasse) *Vormtekenaar* te definiëren met drie implementaties (of afgeleide klassen) die met de specifieke vormen overeenkomen.

```
public interface Vormtekenaar {  
    void tekenVorm ();  
}  
  
public class CirkelTekenaar implements Vormtekenaar {  
    public void tekenVorm () {  
        // tekent een cirkel  
    }  
}  
  
public class DriehoekTekenaar implements Vormtekenaar {  
    public void tekenVorm () {  
        // tekent een driehoek  
    }  
}  
  
public class VierkantTekenaar implements Vormtekenaar {  
    public void tekenVorm () {  
        // tekent een vierkant  
    }  
}
```

We gebruiken nu de toetscode om de juiste vormtekenaar te selecteren en laten

die zijn overeenkomstige vorm tekenen. En natuurlijk doen we dit niet met een meervoudige selectie, maar gebruiken we een map:

```
private Map<Character, Vormtekenaar> map = Map.of(
    'C', new Cirkeltekenaar(),
    'D', new Driehoektekenaar(),
    'V', new Vierkanttekenaar()
);
...
// teken de vorm die overeenkomt met de toets:
map.get(toets).tekenVorm();
```

De drie verschillende vormtekenaarklassen kunnen ook binnenklassen zijn, of zelfs lambda's:

```
private Map<Character, Vormtekenaar> map = Map.of(
    'C', this::tekenCirkel,
    'D', this::tekenDriehoek,
    'V', this::tekenVierkant
);
...

private void tekenCirkel() {
    ...
}

private void tekenDriehoek() {
    ...
}

private void tekenVierkant() {
    ...
}
```

Voor het volgende voorbeeld keren we terug naar de interface *GameCharacter* en de klassen *Wizard*, *Dwarf*, *Hobbit* uit §1.8. We wensen van elk element in een tabel (array) van *GameCharacters* af te drukken over welk type speler het precies gaat.

We zouden dit bijvoorbeeld kunnen doen op de volgende manier:

```

for (GameCharacter c : tabel) {
    if (p instanceof Wizard) {
        System.out.println ("Tovenaar");
    } else if (p instanceof Dwarf) {
        System.out.println ("Dwerg");
    } ...
}

```

Niet alleen de meervoudige selectie zou ondertussen een belletje moeten doen rinkelen, maar ook het veelvuldig gebruik van **instanceof**: we benutten hier niet voldoende de kracht van het objectgericht programmeren.

De meest elegante oplossing is hier wellicht om aan de interface *GameCharacter* een methode *getTypeName* toe te voegen die een string teruggeeft. We implementeren die methode dan voor elke klasse op een andere manier:

```

public class Wizard {
    public String getTypeName() {
        return "Tovenaar";
    }
    ...
}

```

en dan wordt onze meervoudige selectie volledig overbodig:

```

for (GameCharacter c : tabel) {
    System.out.println (c.getTypeName());
}

```

Wat als we het omgekeerde willen doen — als we aan de hand van de naam het object willen creëren?

```

GameCharacter c;
if (naam.equals("Tovenaar")) {
    c = new Wizard ();
} else if (naam.equals("Dwerg")) {
    c = new Dwarf ();
} else if ...

```

Om hier de meervoudige selectie te vermijden, gebruikt men vaak zogenaamde *factories*, objecten met als enig doel andere objecten aan te maken.

Hier hebben we een *WizardFactory* nodig om tovenaars te maken, een *DwarfFactory* voor dwergen, enz. Elk van deze factory-klassen voldoet aan een gemeenschappelijke interface:

```
public interface GameCharacterFactory {  
    GameCharacter createGameCharacter ();  
}
```

De implementaties van *WizardFactory*, *DwarfFactory*, ... zijn heel eenvoudig:

```
public class WizardFactory implements GameCharacterFactory {  
    public GameCharacter createGameCharacter () {  
        return new Wizard ();  
    }  
}  
  
public class DwarfFactory implements GameCharacterFactory {  
    public GameCharacter createGameCharacter () {  
        return new Dwarf ();  
    }  
}
```

(Het retourtype van *createGameCharacter* is telkens *GameCharacter*. Je hoeft dit niet te specialiseren naar *Wizard*, *Dwarf*, ..., maar dat mag wel.)

Van elke factory maken we één object aan en we plaatsen dit in een (hash)map.

```
Map<String,GameCharacterFactory> factories = Map.of(  
    "Tovenaar", new WizardFactory(),  
    "Dwerg", new DwarfFactory(),  
    ...  
);
```

Let op het type van de elementen die in de map zijn opgeslagen: dit is *GameCharacterFactory*, de interface waaraan alle individuele factories voldoen.

Uiteindelijk wordt onze meervoudige selectie dan vervangen door één lijn:



```
GameCharacter c = factories.get(naam).createGameCharacter ();
```

En opnieuw kan je deze implementatie een stuk verlichten door lambda's in te voeren:

```
Map<String, GameCharacterFactory> factories = Map.of(  
    "Tovenaar", Wizard::new,  
    "Dwerg", Dwarf::new,  
    ...  
);  
...  
GameCharacter c = factories.get(naam).createGameCharacter();
```

Let op de speciale dubbele dubbelepuntnotatie voor de referentie naar **new**.

Je hoeft zelfs de interface *GameCharacterFactory* niet in te voeren, want er bestaat al een functionele interface *Supplier*<*GameCharacter*> die hiervoor kan dienen:

```
Map<String, Supplier<GameCharacter>> factories = Map.of(  
    "Tovenaar", Wizard::new,  
    "Dwerg", Dwarf::new,  
    ...  
);  
...  
GameCharacter c = factories.get(naam).get();
```

In plaats van *createGameCharacter* gebruik je nu *get*.

Voor dit eenvoudig voorbeeld zijn factories misschien een beetje zwaar op de hand, maar dit is wel een patroon dat je in de praktijk regelmatig ontmoet.

## 2.3. Het *visitor*-patroon

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.visitors*. ]

De oplossing die we op blz. 47 hebben gebruikt om voor elk personage de type-naam terug te geven — het introduceren van een nieuwe methode *getTypeName* in de interface *GameCharacter* — kan niet altijd worden toegepast: soms heb je de broncode niet van de klassen die je moet aanpassen of zijn er andere redenen om die klassen met rust te laten. Er bestaat echter een standaardontwerptechniek waarmee je dit kan oplossen, het zogenaamde *visitor-patroon*.

Deze techniek laat toe om voor een bestaande klassenhiërarchie gedrag te implementeren dat verschilt van klasse tot klasse, zonder daarvoor de klassen zelf nog te moeten aanpassen. Dit doet men op de volgende manier:

- Introduceer een nieuwe ‘visitor’-interface met een specifieke *visit*-methode voor elke klasse in de hiërarchie.
- Voeg aan elk van de klassen uit de hiërarchie een methode *accept* toe die een dergelijke visitor als argument neemt en die naar de corresponderende *visit*-methode delegeert.

Concreet gaan we in het spelpersonagevoorbeeld op de volgende manier te werk. Als eerste stap definiëren we de interface *CharacterVisitor*:

```
public interface CharacterVisitor<R> {  
    R visitDwarf (Dwarf dwarf);  
    R visitWizard (Wizard wizard);  
    R visitHobbit (Hobbit hobbit);  
}
```

Deze interface bevat een *visit*-methode voor elk van de klassen die de interface *GameCharacter* implementeert.

Merk op dat *CharacterVisitor* een generieke interface is (§1.8) met een typeargument *R*. Dit is het type van de objecten die door de verschillende *visit*-methodes zullen worden geretourneerd<sup>2</sup>.

---

<sup>2</sup>In het ‘officiële’ visitor-patroon zijn de *visit*-methodes procedures in plaats van functies en heb je dit typeargument niet nodig. Onze manier van werken blijkt echter een handige uitbreiding. In het officiële patroon hebben ook alle *visit*-methodes dezelfde naam (kortweg *visit*), maar dat leidt vaak tot verwarring.

De interface *GameCharacter* krijgt nu een methode *accept* met de volgende (generieke) signatuur<sup>3</sup>.

```
public interface GameCharacter {  
    ...  
    <R> R accept (CharacterVisitor<R> visitor);  
}
```

Deze methode moet zo geïmplementeerd worden dat elke klasse de corresponderende methode van *visitor* oproept:

```
public class Dwarf extends GameCharacter {  
    ...  
  
    public <R> R accept (CharacterVisitor<R> visitor) {  
        return visitor.visitDwarf(this);  
    }  
}  
  
public class Wizard extends GameCharacter {  
    ...  
  
    public <R> R accept (CharacterVisitor<R> visitor) {  
        return visitor.visitWizard(this);  
    }  
}  
  
public class Hobbit extends GameCharacter {  
    ...  
  
    public <R> R accept (CharacterVisitor<R> visitor) {  
        return visitor.visitHobbit(this);  
    }  
}
```

Hoe vragen we nu de typenaam op van een personage met behulp van dit visitor-patroon? We schrijven een nieuwe visitorklasse *GetTypeName*, op de volgende manier.

---

<sup>3</sup>Het visitor-patroon kan dus alleen gebruikt worden wanneer dit bij het oorspronkelijke ontwerp is voorzien. Zonder toegang tot de broncode van alle klassen, kan je er achteraf geen *accept*-methode meer aan toevoegen.

```

public class GetTypeName implements CharacterVisitor<String> {

    public String visitDwarf(Dwarf dwarf) {
        return "Dwerg";
    }

    public String visitWizard(Wizard wizard) {
        return "Tovenaar";
    }

    public String visitHobbit(Hobbit hobbit) {
        return "Hobbit";
    }
}

```

Heb je nu een variabele *character* van het type *GameCharacter*, dan bekom je de typenaam als volgt:

```

String result = character.accept(new GetTypeName());

```

(Het kan nuttig zijn om eens stap voor stap te doorlopen wat er precies gebeurt wanneer bijvoorbeeld *character* een object is van de klasse *Wizard*, en te zien dat er hier wel degelijk "Tovenaar" in *result* wordt gestopt.)

Om de notatie te vereenvoudigen, voeg je eventueel de volgende klassenmethode toe aan *GetTypeName*.

```

public static String of(GameCharacter character) {
    return character.accept(new GetTypeName());
}

```

en dan vraag je de typenaam van een personage op met

```

String result = GetTypeName.of(character);

```

In dit voorbeeld retourneren de *visit*-methodes een waarde. Wat doe je wanneer je geen retourwaarde nodig hebt, m.a.w., *visit*-procedures in plaats van *visit*-functies? Hiervoor voorziet Java het type *Void* (met hoofdletter). Dit is een type dat enkel **null** toelaat als waarde en dat precies voor dit soort toepassingen in het leven is geroepen.

De volgende visitor drukt bijvoorbeeld een begroeting af die specifiek is voor elk personage:

```
public class PrintGreeting implements CharacterVisitor<Void> {  
  
    public Void visitDwarf(Dwarf dwarf) {  
        System.out.println("Hiho!");  
        return null;  
    }  
  
    public Void visitWizard(Wizard wizard) {  
        System.out.println("Greetings! I am the mighty wizard " +  
                            wizard.getName();  
        return null;  
    }  
  
    public Void visitHobbit(Hobbit hobbit) {  
        System.out.println(  
            "Hello! My name is " + hobbit.getName() +  
            ". Did you bring anything to eat?");  
        return null;  
    }  
  
    public static void of(GameCharacter character) {  
        character.accept(new PrintGreeting());  
    }  
}
```

(De ‘**return null**’ in de *visit*-methodes moet er steeds staan, want hoewel ze zich als procedure gedragen, blijven het technisch gezien nog steeds functies.)



## 3. Persistente informatie

Vooraleer we het domein van de grafische gebruikersinterfaces betreden, behandelen we eerst een ander aspect van professioneel softwareontwerp: het gebruik van *persistente informatie* — het inlezen en uitschrijven van informatie die buiten het programma is opgeslagen (op de eigen computer of op een externe server).

We beginnen met een kort overzicht van de basisbewerkingen in Java voor het werken met invoer en uitvoer. We gaan hier niet al te diep in detail omdat deze manier van werken in de praktijk niet veel voorkomt. Het is inderdaad zo dat in een professionele Java-toepassing heel veel gegevens vanuit bestanden worden ingelezen of ernaar worden uitgeschreven, maar een programmeur zal die bestanden meestal niet zelf lijn per lijn of al zeker niet byte per byte gaan verwerken. In de plaats gebruikt hij standaardbestandsformaten en standaardbibliotheken die met dergelijke bestanden omgaan. We zullen daarvan later in dit hoofdstuk ook enkele voorbeelden geven.

In het bijzonder bespreken we twee toepassingen van persistentie die in de praktijk heel vaak voorkomen. Eerst behandelen we enkele technieken om *configuratieinformatie* te hanteren, gegevens die het programma nodig heeft maar die we niet wensen hard te coderen in de programmacode. Dergelijke informatie wordt vaak opgeslagen in een tekstbestand dat met een gewone editor kan worden aangepast, bijvoorbeeld door een systeembeheerder bij installatie van het programma. Dergelijke bestanden worden door de software enkel gelezen maar niet aangepast.

Daarnaast geven we ook een inleiding tot het gebruik van databanken in Java. Wanneer een toepassing gegevens verwerkt die blijvend moeten worden bewaard, dan kan je die best opslaan in een zogenaamde *relationele databank*<sup>1</sup>. Java voorziet de *JDBC*-bibliotheek voor communicatie met een dergelijke databank. Dit onderwerp stellen we uit tot hoofdstuk 6.

---

<sup>1</sup>Vroeger gebruikte men een databank enkel wanneer het over een grote hoeveelheid gegevens ging met ingewikkelde onderlinge verbanden. Tegenwoordig bestaat er lichtgewicht databanksoftware die performant genoeg is om het interessant te maken om voor alle projecten voor een databanktechnologie te kiezen, ook wanneer het over een klein aantal gegevens gaat.

## 3.1. Tekstbestanden lezen en schrijven

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.io*. ]

Omdat je in de praktijk niet zo vaak bestanden rechtstreeks hoeft te lezen of te schrijven, geven we hier slechts een inleidende bespreking van de standaardklassen voor in- en uitvoer in Java en beperken we ons daarbij hoofdzakelijk tot het lijn per lijn lezen en schrijven van tekstbestanden. Hiermee behandelen we slechts een heel klein gedeelte van de vele klassen, interfaces en methodes voor invoer en uitvoer die door Java worden aangeboden in de pakketten *java.io* en *java.nio*. Voor meer details kan je de elektronische documentatie (API) raadplegen of de online Java Tutorials van Oracle<sup>2</sup>.

Bij de klassen die we hieronder zullen ontmoeten, gebeurt invoer en uitvoer in Java altijd *sequentieel*. Invoer- en uitvoerkanalen worden beschouwd als gegevensstromen (we gebruiken hiervoor de Engelse term *streams*<sup>3</sup>) van en naar de Javatoepassing. Concreet betekent dit bijvoorbeeld dat we de 200ste lijn van een tekstbestand niet kunnen bekijken zonder eerst de 199 lijnen in te lezen die ervoor komen, en dat we een lijn van een tekstbestand niet zonder meer kunnen aanpassen eenmaal we die uitgeschreven hebben, behalve door het ganse bestand opnieuw uit te schrijven.

Streams zijn vaak gekoppeld aan bestanden, maar kunnen ook geassocieerd zijn met het standaardinvoer- en uitvoerkanaal (toetsenbord en ‘opdracht’venster), met tabellen in het geheugen, met webpagina’s, met netwerkverbindingen, enz.

Om tekstbestanden in te lezen en uit te schrijven, zullen wij de klassen *BufferedReader* en *PrintWriter* gebruiken. Dergelijke *readers* en *writers* houden rekening met de codering die gebruikt wordt om ‘internationale’ lettertekens voor te stellen — een ‘é’ met accent neemt op een bestand bijvoorbeeld niet altijd evenveel plaats in als een ‘e’ zonder accent. Achter een dergelijke reader of writer gaat steeds een *InputStream* of *OutputStream* schuil, die een bestand beschouwt als een opeenvolging van bytes in plaats van lettertekens<sup>4</sup>. En ook al zullen we nooit rechtstreeks lezen of schrijven van of naar een dergelijke stream, kunnen we die toch niet helemaal negeren.

De meeste invoer- en uitvoermethoden in Java genereren uitzonderingen. Omdat invoer en uitvoer voor een gedeelte buiten het programma plaatsvindt, is er nu

---

<sup>2</sup>Ook voor andere aspecten van Java zijn deze tutorials vaak een goede bron van informatie.

<sup>3</sup>Niet te verwarren met de *Java streams* uit §1.7

<sup>4</sup>We gebruiken in deze tekst het woord *letterteken* als vertaling van het Engelse *character*.



eenmaal meer kans dat er iets fout loopt. Een professioneel programmeur probeert dan ook altijd zo goed mogelijk deze uitzonderingen op te vangen, daarom zijn het ook allemaal *gecontroleerde* uitzonderingen (checked exceptions)<sup>5</sup>. We durven dat in deze tekst echter wel eens ‘vergeten’ — opdat de voorbeelden eenvoudig zouden blijven.

Invoer/uitvoer-uitzonderingen zijn van het type *IOException*. Deze klasse bezit een aantal deelklassen voor meer specifieke uitzonderingen, zoals bijvoorbeeld *FileNotFoundException* wanneer je een bestand probeert te lezen met een naam die niet bestaat.

## 3.2. *BufferedReader* en *PrintWriter*

In het volgende programma lezen we een bestand in met op elke lijn een kommagetal en drukken de som ervan af.

```
public static void main (String[] args) {
    try {
        double totaal = 0.0;
        BufferedReader reader =
            Files.newBufferedReader(Path.of(args[0])); // A
        String line = reader.readLine();             // B
        while (line != null) {
            totaal += Double.parseDouble(line);
            line = reader.readLine();                 // B
        }
        reader.close ();                             // C
        System.out.println("Som = " + totaal);
    } catch (IOException ex) {
        System.out.println("Kon bestand niet lezen:" + ex);
    } catch (NumberFormatException ex) {
        System.out.println("Bestand mag enkel getallen bevatten");
    }
}
```

In dit voorbeeld zie je al meteen een aantal belangrijke aandachtspunten bij het gebruik van *BufferedReader*.

---

<sup>5</sup>We komen hierop terug in paragraaf §3.3

1. Je hebt een object van de klasse *BufferedReader* nodig om de lijnen van een bestand te lezen. Hier (in de lijn gemarkeerd met '// A') gebruiken we daarvoor de methode *newBufferedReader* (cf. §3.5) maar er bestaan nog heel wat andere manieren om een *buffered reader* aan te maken. Door dit object aan te maken, is het bestand klaar om gelezen te worden. We noemen dit 'openen' van het bestand.
2. Deze methode *readLine* van *BufferedReader* (zie '// B') leest telkens een volledige lijn van de invoerstroom en geeft die terug als string. De waarde van *readLine* is **null** aan het einde van de invoerstroom. De **while**-lus die we hier gebruiken, met een *readLine* vlak voor de lus, en één helemaal op het einde van de lus, is een typisch patroon waarmee een *buffered reader* lijn per lijn wordt doorlopen.
3. Na afloop moet je een open bestand steeds *sluiten* (zie '// C'). Dat is vooral belangrijk bij uitvoer, omdat de kans namelijk groot is dat de laatste kilobytes die je hebt uitgeschreven anders niet worden geregistreerd. Ook wanneer je de invoerstroom vergeet te sluiten, kan dit soms nare gevolgen hebben.
4. Er kan heel wat fout gaan bij het lezen van een bestand. Daarom bevindt het ganse fragment zich binnen een **try-catch**-blok en vangen we de uitzonderingen op door een bericht af te drukken. In paragraaf §3.3 gaan we hier dieper op in.

We kunnen bovenstaand fragment gemakkelijk aanpassen om getallen van het toetsenbord (het standaard invoerkanaal) te lezen. Vervang lijn '// A' door

```
BufferedReader reader =  
    new BufferedReader (new InputStreamReader (System.in)); // A'
```

Java gebruikt de objecten *System.in*, *System.out* en *System.err* om de standaard invoer-, uitvoer en errorkanalen voor te stellen. Om historische redenen zijn dit echter geen *readers* of *writers*, maar *input* en *output streams*. Ze moeten dus nog naar *readers* en *writers* worden omgezet. Dit gebeurt met behulp van de klassen *InputStreamReader* (en *OutputStreamWriter*).

De constructor voor *InputStreamReader* neemt een willekeurige inputstream als argument en construeert daaruit een nieuwe reader. Elke invoeroperatie op een *inputstreamreader* wordt automatisch doorgegeven aan de geassocieerde inputstream.



Bij het omzetten van een *InputStream* of *OutputStream* naar een *InputStreamReader* of *OutputStreamWriter* kan je ook een *encoding* opgeven die aangeeft op welke manier de internationale lettertekens moeten worden geïnterpreteerd

```
BufferedReader reader = new BufferedReader (  
    new InputStreamReader (System.in, StandardCharsets.ISO_8859_1)  
);
```

(ISO-8859-1 is de officiële afkorting voor de codering die in grote delen van Europa wordt gebruikt en ook gekend staat als LATIN1.)

Je kan ook de standaardcodering gebruiken voor het toestel waarop het Java-programma draait

```
BufferedReader reader = new BufferedReader (  
    new InputStreamReader (System.in, Charset.defaultCharset())  
);
```

maar in dit geval is deze tweede parameter overbodig.

Om lijnen uit te schrijven naar een bestand (of andere uitvoerstroom), gebruik je *PrintWriter*. Je kan een dergelijke *print writer* aanmaken met één van de volgende constructoren:

```
public PrintWriter (Writer w);  
public PrintWriter (OutputStream out);
```

Merk op dat je in dit geval reeds over een *writer* of *output stream* moet beschikken.



Voor uitvoer naar een bestand kan je ook het volgende schrijven:

```
PrintWriter writer = new PrintWriter (padnaam);
```

De basismethodes van *PrintWriter* heten *print* en *println* en laten toe om getallen, lettertekens en in principe alle objecten in tekstvorm uit te schrijven — op dezelfde manier waarop we gewoon zijn met *System.out* te werken. (Nochtans

behoort *System.out* tot de klasse *PrintStream* en niet tot *PrintWriter*.) De *print*- en *println*-methodes hebben het bijkomende voordeel dat ze geen uitzonderingen genereren.

Daarnaast bevat *PrintWriter* een aantal *printf* methodes die nog meer flexibiliteit bieden voor geformatteerde uitvoer. Voor meer informatie verwijzen we naar de elektronische documentatie.

### 3.3. Opvangen van uitzonderingen

Als je in je programma gebruik maakt van in- of uitvoer moet je er altijd rekening mee houden dat er uitzonderingen kunnen optreden. Daarenboven moet je erop letten dat elk bestand dat wordt geopend, ook steeds wordt gesloten, zelfs wanneer er een uitzondering wordt opgegooid. Om dit op een professionele manier te doen, maak je meestal gebruik van een **try-catch-finally**-combinatie, zoals in het volgende voorbeeld:

```
BufferedReader reader
    = new BufferedReader (new InputStreamReader (System.in));
try {
    String line = reader.readLine ();
    while (line != null) {
        ...
        line = reader.readLine ();
    }
} catch (IOException ex) {
    ... // verwerk de uitzondering
} finally {
    reader.close ();
}
```

Merk op dat we hier het voorbeeld van blz. 57 al iets hebben verbeterd door de *reader* te sluiten in een **finally**-constructie. Zo wordt de *close* zeker uitgevoerd, ook wanneer er tijdens de verwerking van de lijnen iets fout gaat.

Het voorbeeld wordt iets moeilijker wanneer we niet van het standaard invoerkanal lezen, maar van een bestand met een opgegeven naam, omdat het aanmaken van de *BufferedReader* in dat geval zelf ook nog een uitzondering kan opgooien:

```

try {
    BufferedReader reader = Files.newBufferedReader(bestandsnaam);
    try {
        String line = reader.readLine();
        while (line != null) {
            ...
            line = reader.readLine();
        }
    } catch (IOException ex) {
        ... // verwerk de uitzondering
    } finally {
        if (reader != null) {
            reader.close ();
        }
    }
} catch (FileNotFoundException fnfe) {
    ... // als het bestand niet kon worden geopend
}

```

Merk trouwens op dat ook *reader.close()* een uitzondering kan opgooien — ons eerste voorbeeld was dus eigenlijk nog niet volledig. Nog erger wordt het bij databanken — zie hoofdstuk 6 — waar je zowel een *result set*, een opdracht als een verbinding moet afsluiten, met vier vernestelde **try**-blokken tot gevolg!

Gelukkig bestaat er in Java al enige tijd een nieuwe vorm van de **try**-blok die ons dit allemaal heel wat korter laat neerschrijven. Zo kan je het eerste voorbeeld nu verkort als volgt noteren:

```

try (BufferedReader reader
    = new BufferedReader (new InputStreamReader (System.in))
    ){
    String line = reader.readLine ();
    while (line != null) {
        ...
        line = reader.readLine ();
    }
} catch (IOException ex) {
    ... // verwerk de uitzondering
}

```

Na de **try** staat nu tussen ronde haakjes een opdracht die een nieuwe *bron* aanmaakt, namelijk *reader*. (In deze context is een ‘bron’ elk object dat de inter-

face *AutoCloseable* implementeert — zie de elektronische documentatie voor bijkomende details.)

Deze bron wordt automatisch gesloten na afloop van het **try**-blok, en eventuele uitzonderingen die zich voordoen bij het aanmaken of sluiten van de bron, worden netjes ‘naar boven’ doorgegeven.

Het tweede voorbeeld kunnen we dus herschrijven als

```
try ( BufferedReader reader = Files.newBufferedReader(bestandsnaam)) {  
    String line = reader.readLine();  
    while (line != null) {  
        ...  
        line = reader.readLine();  
    }  
} catch (IOException ex) {  
    ... // verwerk de uitzondering  
} catch (FileNotFoundException fnfe) {  
    ... // als het bestand niet kon worden geopend  
}
```

Merk bovendien op dat we nu beide uitzonderingen in hetzelfde **try-catch**-blok kunnen opvangen.

Je kan in een dergelijke ‘try-met-bronnen’ (*try with resources*) trouwens meer dan één bron tegelijk initialiseren (gescheiden door puntkomma’s). In het volgende voorbeeld schrijven we de inhoud van het ene bestand naar het andere:

```
try (  
    BufferedReader reader = Files.newBufferedReader(Path.of(naam1));  
    PrintWriter writer = new PrintWriter(naam2)  
) {  
    String line = reader.readLine();  
    while (line != null) {  
        writer.println(line);  
        line = reader.readLine();  
    }  
} catch (IOException ex) {  
    System.err.println("Kopiëren is mislukt: " + ex);  
}
```

### 3.4. Bestanden lezen uit het class path

Zoals je wellicht weet bestaan er officiële tweelettercodes voor alle landen van de wereld — de zogenaamde ISO-codes ('BE' voor België, 'NL' voor Nederland, 'FR' voor Frankrijk, ...). De codes zijn handig om mee te werken in een programma, maar voor de eindgebruiker is het vaak best om de volledige landsnaam af te beelden.

We willen een methode *getNameForCode* schrijven die de naam teruggeeft van het land waarvan we de tweelettercode als parameter doorgeven. Op zich is dit geen moeilijke opgave — je stopt de code/naam-paren gewoon in een *map*.

```
public String getNameForCode(String code) {  
    return NAMES.get(code);  
}
```

Omdat er ongeveer 250 landen zijn in de wereld en omdat de lijst af en toe moet worden aangepast, willen we de codes liefst niet 'hard coderen' in het programma, maar ze inlezen uit een bestand wanneer het programma opstart. Bij een bestand met de volgende inhoud:

```
AFGHANISTAN; AF  
ÅLAND ISLANDS; AX  
ALBANIA; AL  
...  
ZAMBIA; ZM  
ZIMBABWE; ZW
```

kan je de *map* op de volgende manier inlezen:

```
private void readNames(BufferedReader reader, Map<String,String> map)  
    throws IOException {  
    String line = reader.readLine();  
    while (line != null) {  
        int pos = line.lastIndexOf(' ');  
        map.put(  
            line.substring(pos+1),  
            line.substring(0, pos)  
        );  
        line = reader.readLine();  
    }  
}
```

Er rest ons enkel nog de *reader* aan te maken zoals in paragraaf §3.2. We moeten enkel nog beslissen waar we het bewuste bestand ergens zullen plaatsen, en hier wringt het schoentje, zeker als het hier om een toepassing gaat die we op meerdere computers willen installeren, misschien zelfs met verschillende besturings-systemen. Het is niet zo gemakkelijk om een uniforme plaats voor een dergelijk bestand te vinden en dan mogen we bovendien niet vergeten het bestand telkens mee te installeren met het programma.

Dergelijke externe bestanden (afbeeldingen, hulpteksten, configuratie-informatie, enz.) die onlosmakelijk met je programma zijn verbonden en die enkel maar door het programma worden ingelezen en niet gewijzigd, noemt men *bronnen* (Engels: *resources*). Java biedt een aantal handige manieren om met dergelijke bronnen om te gaan. Hierbij maken we gebruik van het zogenaamde *class path*.

Het class path is het geheel van 'locaties' waar Java op zoek gaat naar klassen. Deze locaties kunnen mappen (directories) zijn, maar ook *JAR-archieven* of plaatsen op het Internet<sup>6</sup>.

Je kan elke klasse (of eigenlijk aan zijn *class loader*) vragen om voor jou een bepaalde bron op te sporen. In de praktijk betekent dit meestal dat de bron wordt gezocht op dezelfde plaats waar ook het **.class**-bestand van die klasse is gevonden (dus ergens in het class path).

In ons voorbeeld doen we dit op de volgende manier:

```
InputStream in =  
    ISOCodes.class.getResourceAsStream("isocodes.txt");
```

De methode *getResourceAsStream* van *Class* geeft een *InputStream* terug uit het class path aan de hand van zijn naam. We zoeken het bestand *isocodes.txt* dus in dezelfde folder als de klasse *ISOCodes*. We moeten dit resultaat nu wel nog omzetten naar een *BufferedReader*:

```
reader = new BufferedReader (new InputStreamReader (in));
```

Het mooie van de zaak is dat dit ook werkt wanneer *isocodes.txt* samen met de klassenbestanden in een JAR-archief werden verpakt en dat het bestand dus automatisch mee wordt geïnstalleerd met de rest van het programma.

---

<sup>6</sup>Het class path kan je ook opgeven in de omgevingsvariabele `CLASSPATH` of als `-cp`-argument bij het opstarten van `java` vanop de opdrachtlijn. Ook IDEA biedt je de mogelijkheid om een class path in te stellen vooraleer je een programma uitvoert.



## 3.5. Het pakket `java.nio.file`

[ Je vindt de voorbeelden uit onderstaande paragraaf in *be.ugent.objprog.nio.* ]

De Java-ontwikkelaars hebben geruime tijd geleden een aantal aspecten van het werken met bestanden in Java grondig vernieuwd, in het bijzonder het wissen en hernoemen van bestanden en het ophijsten van mappen. (Aan *BufferedReader* en *PrintWriter* is er echter weinig veranderd.)

Vroeger werd dit soort handelingen verricht met behulp van de klasse *File*. Niet alleen was de naam van deze klasse verkeerd gekozen (*Filename* was beter geweest), maar ook diende ze tegelijk twee verschillende doeleinden: het werken met bestands- en padnamen en het verplaatsen of wissen van bestanden.

Het nieuwe pakket *java.nio.file* scheidt deze functionaliteiten netjes van elkaar en maakt het geheel bovendien robuuster. We zullen hier enkele van de klassen uit dit pakket kort bespreken. Voor meer informatie verwijzen we (opnieuw) naar de elektronische documentatie of de online tutorial.

Eerst en vooral is er de interface *Path* die abstractie maakt van het begrip ‘padnaam’. Deze interface heeft heel wat methodes voor het manipuleren van padnamen (bijvoorbeeld, omzetten van relatieve padnamen naar absolute, en vice versa) en doet dit op een manier die onafhankelijk is van het besturingssysteem, i.h.b., zonder dat je er als programmeur mee rekening moet houden dat in Windows achterwaartse schuine strepen worden gebruikt om delen van een padnaam van elkaar te scheiden, en in Linux en MacOS voorwaartse schuine strepen.

*Path* is trouwens een interface en geen echte klasse, en heeft dus geen constructor. Nieuwe objecten van het type *Path* moet je aanmaken met *Path.of(..)*<sup>7</sup>.

De padnaam van een bestand `text.dat` in de subdirectory `Text` van de directory `Windows`, maak je bijvoorbeeld aan op de volgende manier:

```
Path path = Path.of("Windows", "Text", "text.dat");
```

of je kan dit ook in afzonderlijke stappen

```
Path directory = Path.of("Windows", "Text");  
Path path = directory.resolve("text.dat");
```

---

<sup>7</sup>Nieuwere versies van Java laten namelijk klassenmethoden toe in interfaces.

Eenmaal je een *Path*-object hebt aangemaakt, wil je er natuurlijk ook iets mee doen, bijvoorbeeld de inhoud van het corresponderend bestand inlezen of een nieuw bestand aanmaken met de gegeven padnaam. De klasse *Files* (meervoud) biedt je hier tal van mogelijkheden.

We hebben eerder al de volgende methode uit de klasse *Files* ontmoet waarmee je een buffered reader kan maken die lijnen leest uit een bestand:

```
public static BufferedReader newBufferedReader (Path path)  
                                                    throws IOException;
```

Je kan naast een *Path* ook een *Charset* opgeven dat aangeeft welke codering Java moet gebruiken om de bytes in het bestand als lettertekens te interpreteren.

Als je de lijnen die je hebt ingelezen enkel maar in een lijst wil stoppen, dan kan je beter de volgend methode uit *Files* gebruiken.

```
public static List<String> readAllLines(Path path) throws IOException;
```

Of je kan ze verwerken als Java stream met *Files.lines*.

*Files* heeft ook een analoge methode *newBufferedWriter* waarmee je een bestand kunt openen om beschreven te worden. Wellicht wil je de buffered writer die je krijgt dan wel eerst omzetten naar een *PrintWriter* — zoals in het volgende programma dat een bestand maakt met daarin de tafels van zeven.

```
public static void main(String[] args) throws IOException {  
    Path path = Path.of("tafels-van-zeven.txt");  
    try (PrintWriter out = new PrintWriter (Files.newBufferedWriter(path))) {  
        for (int i = 1; i < 10; i++) {  
            out.println("7 x " + i + " = " + (7*i));  
        }  
    }  
}
```

Een try-met-bronnen hoeft inderdaad geen **catch**-clausule te hebben — hij heeft immers al een (verborgen) **finally**-clausule.

De klasse *Files* bezit ook methodes waarmee je bestanden kan kopiëren zonder ze zelf te moeten inlezen en weer uitschrijven, om bestanden van naam te veranderen, te verplaatsen, te verwijderen, enz.

Tot slot nog even opmerken dat er in de recente versies van Java jammer genoeg nog steeds heel wat standaardklassen zijn die nog niet aan de nieuwe invoer en uitvoer zijn aangepast — de klasse *FileChooser* uit JavaFX werkt bijvoorbeeld nog steeds met *File* en niet met *Path*. Gelukkig heeft *Path* een methode *toFile* en *File* een methode *toPath* waarmee je gemakkelijk tussen de oude en nieuwe voorstellingen kunt wisselen.

## 3.6. Netwerkverbindingen opzetten en gebruiken

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.net*. ]

In- en uitvoer over een netwerkverbinding gebruikt dezelfde in- en uitvoerstromen als die voor bestanden. Dit maakt het in Java relatief eenvoudig om programma's te schrijven die met elkaar over een netwerkverbinding communiceren. De enige bijkomende stap is het opzetten (en afsluiten) van de verbinding.

De netwerkcommunicatie die we hier bespreken, gebruikt een *client/server*-architectuur. Wanneer twee programma's een netwerkverbinding met elkaar opzetten, geldt het ene als *server* en het andere als *client*. Het server-programma (vaak een *service* genoemd, t.t.z., een dienst) wordt eerst opgestart en draait vaak in een oneindige lus die wacht totdat een client een verbinding legt. Die verbinding wordt dan gebruikt om informatie tussen client en server uit te wisselen, waarna ze wordt afgesloten en de server kan wachten op een nieuwe client. Vaak kan een server ook meerdere clients tegelijk bedienen.

Om met een netwerkdienst te connecteren heb je twee gegevens nodig: de naam (of het IP-adres) van het toestel waarop de dienst ter beschikking staat, en het 'volgnummer' van de dienst, in deze context de *poort* genoemd. Poortnummers kleiner dan 1024 zijn gereserveerd voor standaarddiensten die door heel wat machines worden aangeboden, bijvoorbeeld poort 80 voor de HTTP-dienst die jouw browser gebruikt om webpagina's op te vragen. Voor je eigen server-programma kan je zelf een poortnummer kiezen, zolang het maar groter is dan 1024 (en nog niet door een andere service op hetzelfde toestel wordt gebruikt).

Als eerste voorbeeld schrijven we een programmaatje dat een verbinding maakt met de standaard 'echo'-service<sup>8</sup> (op poort 7). Zoals de naam al aangeeft, stuurt

---

<sup>8</sup>Om dit programma uit te testen, moet je een server kennen die deze dienst aanbiedt. De meeste Linux- of MacOS-toestellen ondersteunen een dergelijk dienst, maar vaak moet je die nog inschakelen. Je kan echter ook wachten tot blz. 68 waar we zelf een echo-service zullen implementeren in Java.

een dergelijke server gewoon de informatie terug die je er naar zendt. Deze service wordt dus voornamelijk gebruikt bij testen.

Java gebruikt de klasse *Socket* om netwerkverbindingen voor te stellen. Bij het aanmaken van een socket geef je naam en poort op van de server. Daarna kan je met *getInputStream* en *getOutputStream* aan deze socket een in- en uitvoerstream opvragen waarmee je met de service kan communiceren. Wat je op de uitvoerstream plaatst, komt terecht bij de server. Wat de server terugstuurt, lees je van de invoerstream. Netwerkstromen zijn binaire stromen. Je moet ze zelf nog omzetten naar gepaste *Readers* en *Writers*.

Het volgende fragment opent een socket naar de echo-service op de eigen machine (aangegeven met de naam `localhost`), stuurt er één lijn naartoe, leest wat er terugkomt van de server en sluit de verbinding terug af.

```
try (
    Socket socket = new Socket ("localhost", 7);    // null mag hier ook
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream())
    )
) {
    out.println("Go forth and multiply!");
    System.out.println(in.readLine());

} catch (IOException e) {
    System.err.println("Er gebeurde een fout");
}
```

Merk op dat een socket en de ermee geassocieerde in- en uitvoerstromen steeds moeten afgesloten worden. Daarom gebruiken we hier een try-met-bronnen.

Zoals je ziet is het opzetten van een netwerk-client vrij eenvoudig. Bij een netwerk-server komt er iets meer te kijken. Een server gebruikt twee verschillende soorten sockets: een *server-socket* en een gewone socket.

Een server-socket maak je aan als object van de klasse *ServerSocket*. De constructor neemt het poortnummer als parameter. De belangrijkste methode van *ServerSocket* is *accept*. Die blijft wachten tot er een client contact zoekt met de server en geeft dan een (gewone) socket terug waarmee tussen client en server wordt gecommuniceerd.

```

try (ServerSocket serverSocket = new ServerSocket(port)) {

    // oneindige server-lus
    for ( ; ; ) {
        try (Socket socket = serverSocket.accept();
            OutputStream out = socket.getOutputStream();
            InputStream in = socket.getInputStream()
        ) {
            int ch = in.read();
            while (ch >= 0) {
                out.write(ch);
                ch = in.read();
            }
        }
    }

} catch (IOException ex) {
    System.err.println("Er gebeurde een fout");
}

```

Merk op dat beide sockets moeten gesloten worden na gebruik. Opnieuw doen we dit met een try-met-bronnen.

In dit voorbeeld kan de server slechts één client tegelijk aan. Een nieuwe *accept* wordt pas uitgevoerd nadat de huidige client zijn verbinding sluit (wat als effect heeft dat *in.read()* een negatief getal retourneert). In §7.1 tonen we hoe je een service schrijft die meerdere clients tegelijkertijd bedient.

Tot slot willen we nog even opmerken dat je het in de praktijk zelden nodig hebt om zelf een netwerkverbinding op te zetten. Vele standaardnetwerkdiensten worden reeds ondersteund door klassen en methodes in de Java-bibliotheek.

Als voorbeeld tonen we hoe je informatie kan ophalen vanop het web. Dit doe je met behulp van de klasse *URL*<sup>9</sup>. Je maakt een *URL*-object aan voor een specifieke URL en opent een invoerstroom met *openStream*. Je kan deze techniek ook gebruiken om te lezen van een *web service*<sup>10</sup>.

<sup>9</sup>Vaak is het zelfs nog iets eenvoudiger. Zo hoef je bij het aanmaken van een afbeelding voor JavaFX (hoofdstuk 4) enkel de URL-string op te geven als stringargument van de constructor '**new** *Image*("http://sample.com/res/flower.png")' en heb je geen *URL*-object nodig.

<sup>10</sup>Schrijven naar een *web service* is iets ingewikkelder. Hiervoor bestaat in de meer recente versies van Java de klasse *HttpClient*. Het zou ons echter te ver leiden om hier dieper op in te gaan.

Onderstaand fragment haalt de tekst op van de ‘GNU Affero General Public License’ en drukt die af.

```
String urlString = "http://www.gnu.org/licenses/agpl.txt";
try (InputStream in = new URL(urlString).openStream();
    BufferedReader rdr = new BufferedReader(new InputStreamReader(in))
) {
    rdr.lines().forEach(System.out::println);
} catch (MalformedURLException e) {
    // zal hier niet gebeuren
} catch (IOException e) {
    System.err.println("Fout bij het inlezen");
}
```

Merk op dat je de gecontroleerde uitzondering *MalformedURLException* moet opvangen. Deze wordt opgeworpen wanneer de string die je aan de constructor van *URL* meegeeft niet de juiste vorm heeft.

Er zijn ook Java-klassen en methoden die rechtstreeks *URL*-objecten als parameters aanvaarden zonder dat je zelf *openStream* hoeft op te roepen. We geven hiervan enkele voorbeelden in §3.9.

## 3.7. Eigenschapsbestanden

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.config*. ]

In de schermafdruck hieronder zie je een eenvoudig ‘about’-venstertje dat hoort bij een programma met de naam *MyApp*.



Niet alle informatie in dit venster is even ‘vast’. Wanneer je het programma bij verschillende klanten installeert, dan is wellicht het email-adres van de systeem-

administrator niet telkens hetzelfde. Dit adres is dus een duidelijke kandidaat om opgenomen te worden in een configuratiebestand (in het class path). Daarnaast willen we ook het versienummer in dit bestand opnemen.

Met wat we hierboven gezien hebben, is het niet zo moeilijk een bestandsformaat te ontwerpen waarin we deze informatie kunnen opslaan op een manier die het voor ons gemakkelijk maakt het met een programma terug in te lezen (bijvoorbeeld lijn per lijn met een *BufferedReader*). We hoeven echter het warm water niet opnieuw uit te vinden — er bestaan diverse bestandsformaten en bijbehorende Java-bibliotheken die onze taak (nog) een stuk eenvoudiger maken.

Als eerste optie bespreken we de zogenaamde *eigenschapsbestanden* (Engels: *property files*). Een eigenschapsbestand<sup>11</sup> is een tekstbestand dat een aantal zogenaamde *sleutels* met hun *waarden* verbindt (beiden zijn strings). In ons voorbeeld zou dit bestand er zo kunnen uitzien:

```
# myapps.properties

sysadmin.email = sys.admin@mycompany.net
version = 1.0-2
```

Lege lijnen en lijnen die beginnen met # worden genegeerd, andere lijnen bevatten een sleutel/waarde-combinatie met een gelijkheidsteken als scheidingsteken (een dubbele punt is ook toegelaten). Traditioneel eindigt de naam van een eigenschapsbestand op de extensie *.properties*, maar dit is niet verplicht.

De inhoud van een eigenschapsbestand kan worden ingeladen als object van het type *Properties* (uit het pakket *java.util*). Aan dit object kan je gemakkelijk de waarde van een bepaalde sleutel opvragen. Het inladen gebeurt met de methode *load*, het opvragen met *getProperty*, zoals je ziet in onderstaande broncode die onderdeel is van de klasse die het ‘about’-venstertje opbouwt:

```
Properties properties = new Properties();
properties.load(
    About.class.getResourceAsStream("myapp.properties")
);
version.setText ("Versie " + properties.getProperty("version"));
email.setText(properties.getProperty("sysadmin.email"));
```

Het eigenschapsbestand heeft de naam *myapp.properties* en staat in het class path, op dezelfde plaats als de gecompileerde versie van *About*.

---

<sup>11</sup>Het woord *eigenschap* kan in een Java-context heel wat verschillende betekenissen hebben. In dit geval heeft dit niets te maken met *getters* en *setters*.

Merk op dat het inlezen van een eigenschapsbestand een uitzondering kan opgoeien van het type *java.io.IOException*. Het programmafragment kan ook een *NullPointerException* veroorzaken wanneer de bron niet wordt gevonden.

Om helemaal correct te zijn, moeten we de invoerstroom van waaruit het eigenschapsbestand wordt gelezen, ook nog sluiten — om onduidelijke redenen doet *Properties.load* dat niet zelf. Dit kan gemakkelijk met een try-met-bronnen:

```
Properties properties = new Properties();
try (InputStream in = About.class.getResourceAsStream(
    "myapp.properties")) {
    properties.load(in);
    version.setText ("Versie " + properties.getProperty("version"));
    email.setText(properties.getProperty("sysadmin.email"));
}
```

Zoals je merkt zijn eigenschapsbestanden een zeer eenvoudig instrument om met configuratiegegevens te werken. Ze worden ook intern door de Java-bibliotheken veel gebruikt (zie ook §4.6). Soms vraagt het gebruik van dergelijke bestanden echter nog wat bijkomend werk.

In nevenstaande tabel staan de regels die aan de universiteit gebruikt worden voor het toekennen van graden. We hebben die informatie nodig in een programma dat puntenlijsten afbeeldt.

Grootste onderscheiding	17	dgo
Grote onderscheiding	15	go
Onderscheiding	13	o
Voldoening	10	v
Niet geslaagd	0	ng

Hoe kunnen we dergelijke gegevens in een eigenschapsbestand opnemen?

Merk op dat we in eigenschapsbestanden enkel stringwaarden kunnen gebruiken<sup>12</sup>. De getallen zullen dus in het programma nog van *String* naar **int** moeten worden omgezet.

Een ander aandachtspunt is dat het aantal configuratiegegevens hier niet op voorhand vastligt: in dit voorbeeld zijn er vijf verschillende graden, maar het is best mogelijk dat onze toepassing uiteindelijk gebruikt wordt in een instituut dat meer of minder graden toekent.

In de praktijk zijn er een aantal manieren waarop dergelijke informatie in een

---

<sup>12</sup>Je moet bovendien opletten met strings die andere dan Westeuropese lettertekens gebruiken. Het €-teken moet je in het eigenschapsbestand bijvoorbeeld voorstellen als `\u20ac`.



eigenschaftsbestand kan worden opgenomen. Een eerste mogelijkheid is de volgende:

```
graden.namen = Grootste onderscheiding, Grote onderscheiding, \
    Onderscheiding, Voldoening, Niet geslaagd
graden.punten = 17,15,13,10,0
graden.codes = dgo,go,o,v,ng
```

(De ‘\’ op het einde van een lijn zorgt ervoor dat ook de volgende lijn nog bij hetzelfde sleutel/waarde-paar wordt gerekend.)

In dit geval zal er nog vrij veel werk moeten gebeuren door het programma nadat de stringwaarden door *getProperty* zijn opgevraagd: je moet zelf de strings in stukjes splitsen en de overbodige spaties eruit verwijderen.

Een tweede optie gebruikt een bestand met de volgende structuur:

```
graden.aantal = 5

graden.namen.0 = Grootste onderscheiding
graden.punten.0 = 17
graden.codes.0 = dgo
...
graden.namen.4 = Niet geslaagd
graden.punten.4 = 0
graden.codes.4 = ng
```

Dit bestand is gemakkelijker te verwerken maar vergt meer werk om op te stellen.

## 3.8. XML met behulp van JDOM

Een manier om bovenstaande problemen te vermijden, is door je gegevens op te slaan in bestanden waarvan de inhoud voldoet aan de *XML-standaard*. Java kent een heel uitgebreide ondersteuning voor het verwerken van dergelijke bestanden.

Java biedt vanzelf reeds heel wat manieren om met XML om te gaan, met bibliotheken zoals *SAX*, *DOM* en *JAXB*. In deze tekst kiezen we er echter voor om een ander systeem te gebruiken, namelijk de *open source*-bibliotheek met de naam *JDOM*, omdat zij veruit de gemakkelijkste interface biedt. We zullen ons hier beperken tot het inlezen van XML-bestanden. Aanmaken en uitschrijven van XML is ook zeer eenvoudig en gebeurt op een analoge manier.

Voor het ‘about’-voorbeeld uit vorige paragraaf zouden we in plaats van een eigenschapsbestand het volgende XML-bestand kunnen gebruiken:

```
<?xml version="1.0" encoding="UTF-8"?>

<applicationConfig version="2.0.5">
  <sysAdminEmail>admin.sys@yourcompany.com</sysAdminEmail>
</applicationConfig>
```

De volledige inhoud van dit bestand, ook wel het XML-*document* genoemd, wordt in JDOM voorgesteld als één enkel object van de klasse *Document* (om precies te zijn, *org.jdom2.Document*) en kan op de volgende manier worden ingelezen:

```
Document document = new SAXBuilder().build(input);
```

Merk op dat de *build*-methode een *JDOMException* kan opgooien, en een *IOException*. Je moet die beide dus ergens opvangen.

Er bestaan diverse methodes waarmee je informatie over dit *Document*-object kunt opvragen, maar doorgaans gebruik je enkel *getRootElement*, die het *element* teruggeeft dat alle gegevens bevat. Bij dit voorbeeld is dit het *applicationConfig*-element dat lijnen 3 tot en met 5 beslaat. Een dergelijk element wordt bijgehouden als een object van het type *Element* (of dus eigenlijk *org.jdom2.Element*).

```
Element root = document.getRootElement ();
```

Een element is een gestructureerd object dat bestaat uit de volgende onderdelen:

- Een *naam*. De naam van het wortelelement is de string “applicationConfig”. Het element op de vierde lijn van het voorbeeldbestand heeft als naam “sysAdminEmail”.
- Andere elementen — die *kinderen* van dit element worden genoemd. Het sysAdminEmail-element is een kind van het applicationConfig-element.
- Zogenaamde *attributen*. Het applicationConfig-element heeft een version-attribuut met een bijbehorende *waarde* (het versienummer, als string).
- Tekstuele inhoud. Het sysAdminEmail-element bevat het bewuste e-mail-adres als tekstuele inhoud.

Om JDOM te gebruiken, moet je een JAR-archief voor deze bibliotheek in het class path plaatsen – zowel tijdens compilatie als tijdens de uitvoering van je programma. Je kan deze JAR downloaden vanaf [jdom.org](http://jdom.org). (Opgelet! we gebruiken versie 2.) Je vindt daar ook de elektronische documentatie.

Elk element heeft een naam, maar niet elk element hoeft tegelijkertijd kindelementen, attributen en tekst te bevatten.

De waarde van een attribuut vraag je op met *getAttributeValue(naam)*. Deze methode geeft een string terug of **null** wanneer het element geen attribuut heeft met de opgegeven naam. Een kindelement bekom je met *getChild(naam)*. Het resultaat is opnieuw van het type *Element*. Je kan ook *alle* kinderen met een bepaalde naam opvragen (zie verder.)

Om de tekstuele inhoud van een element op te halen, gebruik je ofwel *getText*, *getTextTrim* of *getTextNormalize*. De eerste methode geeft de tekst letterlijk terug, in de tweede wordt witruimte vóór- en achteraan de tekst automatisch verwijderd en bij de derde wordt ook tussenliggende witruimte genormaliseerd: opvolgende spaties en *linefeeds* worden tot één enkele spatie herleid.

Passen we dit allemaal toe op ons voorbeeld, dan krijgen we het volgende:

```
version.setText("Versie " + element.getAttributeValue("version"));
email.setText(element.getChild("sysAdminEmail").getTextTrim());
```

Als tweede toepassing van JDOM bekijken we opnieuw het ‘graden’-voorbeeld. We gebruiken een XML-bestand van de volgende vorm:

```
<grades>
  <grade code="dgo">
    <name>Grootste onderscheiding</name>
    <points>17</points>
  </grade>
  <grade code="go">
    <name>Grote onderscheiding</name>
    <points>15</points>
  </grade>
  ...
</grades>
```

We zetten dit bestand om naar een lijst van objecten van de klasse *Grade*, een record-klasse die die drie eigenschappen groepeerd: de strings *name* en *code* en het geheel getal *points*.

```

Element grades = new SAXBuilder().build(...).getRootElement();
List<Grade> list = grades.getChildren("grade").stream()
    .map(element -> new Grade(
        element.getChild("name").getTextNormalize(),
        Integer.parseInt(element.getChild("points").getTextNormalize()),
        element.getAttributeValue("code")
    )).collect(Collectors.toList());

```

We maken hierbij van de gelegenheid gebruik om nog eens een Java stream toe te passen.

De klasse *Element* bezit ook een methode *getChildren()*, zonder parameter, die *alle* kinderen van het element retourneert. Bij het overlopen van die kinderen kan je dan hun naam opvragen met *getName*. Deze methode kan handig zijn in situaties waarbij je niet op voorhand weet welke kinderen er in het XML-bestand al dan niet aanwezig zijn.

## 3.9. JSON met behulp van Jackson

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.config* en *be.ugent.objprog.json*. ]

Naast XML wordt tegenwoordig ook vaak JSON<sup>13</sup> gebruikt om informatie uit te wisselen tussen verschillende programma's.

JSON is een heel eenvoudig tekstueel formaat. Het ondersteunt getallen, strings, booleans, null en arrays en 'objecten'. 'Objecten' zijn sleutel/waarde-paren genoteerd tussen accolades, arrays worden genoteerd tussen vierkante haken. Arrays en objecten kunnen getallen, strings, enz. bevatten, maar ook terug andere arrays en objecten.

Het 'about'-voorbeeld van blz. 74 kan er in JSON bijvoorbeeld zo uitzien:

```

{ "version": "2.0.5",
  "sysAdminEmail" : "admin.sys@yourcompany.com"
}

```

en het 'graden'-voorbeeld van blz. 75 zo:

<sup>13</sup>JSON, spreekt uit 'dzejisen', is de afkorting voor *Javascript Object Notation*. Het formaat is afgeleid van de notatie voor objecten in Javascript maar wordt ondertussen door de meeste programmeertalen via één of andere bibliotheek ondersteund.

```
[ {
  "name" : "Grootste onderscheiding",
  "points" : 17,
  "code" : "dgo"
}, { ...
}, {
  "name" : "Niet geslaagd",
  "points" : 0,
  "code" : "ng"
} ]
```

Java biedt geen standaardondersteuning voor JSON. We gebruiken daarom opnieuw een externe bibliotheek, *Jackson* genaamd. Hoewel Jackson ook op dezelfde manier kan gebruikt worden als JDOM<sup>14</sup> maken we van de gelegenheid gebruik om hier een techniek te illustreren die door Jackson *full data binding* wordt genoemd.

Hierbij wordt een JSON-string rechtstreeks omgezet naar een Java-object van een gegeven klasse, zonder dat de programmeur het object zelf uit verschillende onderdelen moet samenstellen.

Voor de ‘about’-toepassing gaat we bijvoorbeeld op de volgende manier te werk. We definiëren eerst een eenvoudige recordklasse die de ingelezen informatie zal bevatten

```
public record ApplicationConfig (String version, String sysAdminEmail) { }
```

en dan gebruiken we Jacksons *ObjectMapper* om de JSON-string om te zetten naar een object van deze klasse:

```
ObjectMapper mapper = new ObjectMapper();
ApplicationConfig appConfig = mapper.readValue(
    getClass().getResource("appconf.json"),
    ApplicationConfig.class
);

// doe iets met deze informatie
version.setText("Versie " + appConfig.version());
email.setText(appConfig.sysAdminEmail().trim());
```

---

<sup>14</sup>Jackson biedt trouwens niet enkel ondersteuning voor JSON, waar het oorspronkelijk voor bedoeld was, maar ook voor XML en een resem andere formaten.

Net zoals bij JDOM moeten er bijkomende JARs in het *classpath* staan om Jackson te kunnen gebruiken. Als je programmeert in een geïntegreerde omgeving (zoals IntelliJ IDEA) dan biedt die omgeving meestal hulpmiddelen aan om de correcte versies van dergelijke externe Java-bibliotheken automatisch op de juiste plaatsen te installeren.

De voorbeeldbroncode voor deze nota's gebruikt hiervoor *Maven*. Het bestand `pom.xml` bevat daar alle informatie over welke externe bibliotheken er nodig zijn — JDOM, Jackson en de databank-drivers voor hoofdstuk 6.

De methode *readValue* bestaat in verschillende versies, telkens met een ander type voor het eerste argument. In dit voorbeeld is dit een *URL*-object. Het kan ook een *InputStream* zijn, of een *Reader* of een *String*. Het tweede argument is de naam van de klasse die bij de omzetting gebruikt moet worden. Een record-klasse is hier het meest eenvoudig om mee te werken, maar het kan ook een klasse zijn met gepaste getters en setters.

Een lijst of array van objecten inlezen gebeurt op een zeer gelijkaardige manier, we hebben enkel een kleine tussenstap nodig. Hieronder lezen we de lijst van graden in van het bestand op de vorige bladzijde:

```
ObjectMapper mapper = new ObjectMapper();
List<Grade> list = mapper.readerForListOf(Grade.class)
    .readValue(getClass().getResource("grades.json"));
```

We maken dus eerst een ‘reader’ aan die lijsten kan inlezen van elementen van een gegeven klasse en roepen daarvoor *readValue* op. Er zijn ook dergelijke readers voor arrays en voor maps.

Omzetten van een (lijst van) Java-object(en) naar een JSON-string is even eenvoudig:

```
ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(writer, listOfGrades);
```

Hier is *writer* een *Writer*, maar ook een *OutputStream* is toegelaten. Omzetten naar een string zonder die meteen uit te schrijven kan met *writeValueAsString*:

```
ObjectMapper mapper = new ObjectMapper();
mapper.configure(SerializationFeature.INDENT_OUTPUT, true);
System.out.println(
    mapper.writeValueAsString(listOfGrades)
);
```

We hebben hier een tweede lijn toegevoegd waarmee we aangeven dat het resultaat netjes moet geïndenteerd zijn (en over verschillende lijnen verspreid).

De reden waarom bovenstaande voorbeelden zo eenvoudig zijn, is omdat we zelf het JSON-formaat van onze gegevens en klassen hebben vastgelegd zodat de veldnamen netjes met elkaar overeenkomen. Dit is in de praktijk niet altijd mogelijk.

In het volgende voorbeeld zullen we JSON inlezen van een publieke *web service* met URL `https://v2.jokeapi.dev/joke/Any?safe-mode&type=twopart`. Tik je deze URL in in een browser dan krijg je een ‘pagina’ met een inhoud zoals deze:

```
{
  "error": false,
  "category": "Pun",
  "type": "twopart",
  "setup": "What did the cell say when his sister stepped on his foot?",
  "delivery": "Mitosis.",
  "flags": {
    ...
  },
  ...
}
```

Heel wat van die gegevens heb je niet nodig als je enkel in de mop zelf geïnteresseerd bent, t.t.z., als je dit JSON-object wil omzetten naar een record van het volgende type

```
public record Joke (String setup, String delivery) { }
```

Jackson laat dit echter niet zomaar toe, tenzij we expliciet configureren dat overbodige velden mogen weggelaten worden:

```
URL url = new URL("https://...");
ObjectMapper mapper = new ObjectMapper ();
mapper.configure(
    DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
Joke joke = mapper.readValue(url, Joke.class);
```

Wanneer de JSON-string door een andere programmeertaal werd gegenereerd dan Java, volgen de veldnamen niet altijd de Java-conventies. De *Star Wars API*<sup>15</sup>

---

<sup>15</sup>`https://swapi.dev/`.

gebruikt bijvoorbeeld 'rotation\_period' en 'orbital\_period' als veldnamen, terwijl we die in Java zouden *rotationPeriod* en *orbitalPeriod* noemen. Je kan dit oplossen door een *annotatie* toe te voegen aan veld, getter of setter:

```
public class StarWarsPlanet {  
  
    private int rotationPeriod;  
  
    public int getRotationPeriod() {  
        return rotationPeriod;  
    }  
  
    @JsonProperty("rotation_period")  
    public void setRotationPeriod(int rotationPeriod) {  
        this.rotationPeriod = rotationPeriod;  
    }  
    ...  
}
```

Hiermee zal Jackson wat in JSON als `rotation_period` staat opgegeven, in Java instellen met *setRotationPeriod*. Dit kan ook bij de parameters van een recordklasse:

```
public record StarWarsPlanet (  
    String name,  
    @JsonProperty("rotation_period") int rotationPeriod,  
    @JsonProperty("orbital_period") int orbitalPeriod  
) {}
```



## 4. JavaFX: aanvullingen

[ Als voorkennis voor dit hoofdstuk verwachten we dat je reeds een basisnotie hebt van de belangrijkste begrippen uit JavaFX: componenten, gebeurtenissen, FXML-bestanden, partnerklasse, luisteraars, stylesheets, enz. Je kan hiertoe, als aanvulling op deze tekst, de zelfstudiecursus *JavaFX — GUIs in Java* doornemen vooraleer je aan dit hoofdstuk begint. Deze cursus kan je online terugvinden op <http://inigem.ugent.be/jvlfx/jvlfx.pdf> ]

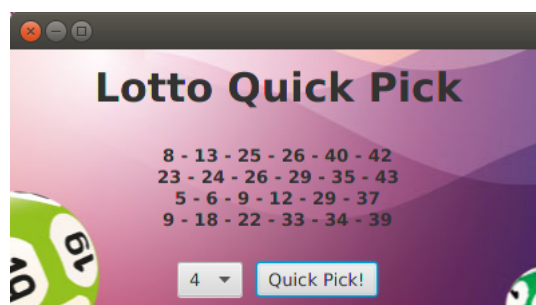
In dit hoofdstuk bespreken we enkele aspecten van JavaFX waarbij je wellicht bij een eerste kennismaking niet hebt stilgestaan. Ze zullen echter hun nut bewijzen wanneer je grotere en meer ingewikkelde JavaFX-toepassingen ontwerpt.

### 4.1. Scènes bouwen zonder FXML

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.quickpick*. ]

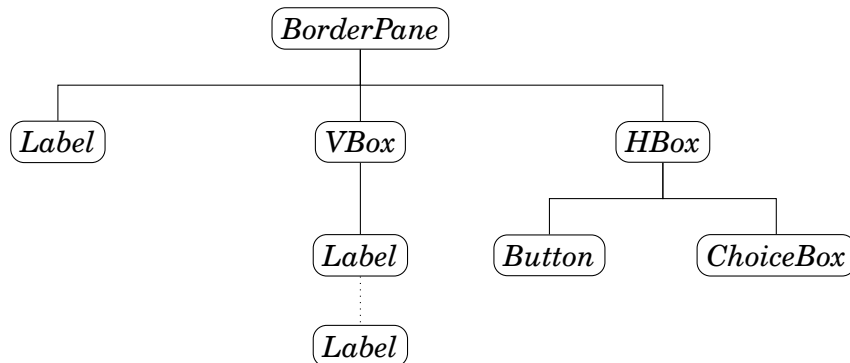
Meestal is het gebruik van FXML-bestanden en bijbehorende partnerklassen, al dan niet geholpen door de *Scene Builder*, de eenvoudigste manier om de grafische user interface van je toepassing te construeren. Het kan echter ook op een andere manier, door de scène rechtstreeks in je Java-code aan te maken.

In de praktijk heb je het niet vaak nodig om de volledige scène in Java op te bouwen (in het voorbeeld dat we hieronder bespreken, overdrijven we dus een beetje) maar het gebeurt wel vaker dat je een bepaald *onderdeel* van een scène ‘met de hand’ moet aanmaken omdat FXML hiervoor te beperkt is.



De toepassing die we hier bespreken, laat toe om met een druk op de knop één of meer willekeurige reeksen lottogetallen te genereren. Hoeveel reeksen het programma moet afdrukken (1–6), wordt aangegeven met een *choice box*.

We schetsen de structuur van deze GUI in onderstaande figuur:



Je vindt deze structuur terug in onderstaande Java-code waarmee de scène wordt gecreëerd:

```

BorderPane borderPane = new BorderPane();

Label label = new Label("Lotto Quick Pick");

VBox vbox = new VBox();

HBox hbox = new HBox();
ChoiceBox<Integer> choiceBox = new ChoiceBox<>();
Button button = new Button("Quick Pick!");
hbox.getChildren().addAll(choiceBox, button);

borderPane.setTop(label);
borderPane.setCenter(vbox);
borderPane.setBottom(hbox);

Scene scene = new Scene(borderPane);
  
```

Merk op dat de labels pas in de *VBox* zullen worden geplaatst op het moment dat er op de knop gedrukt wordt. Het aantal labels hangt immers af van de waarde van de *choice box*. (Dit deel van de scène hadden we dus niet in een FXML-bestand kunnen configureren.)

Vóór JavaFX was de *Swing*-bibliotheek de standaardmanier om GUIs te schrijven in Java. Let dus goed op wanneer je op het Internet bijkomende uitleg opzoekt. Het helpt ook niet dat vele standaardklassen van JavaFX, zoals *Font*, *Color*, *Button*, ... dezelfde naam dragen als gelijkaardige klassen in *Swing* (maar dan in een andere *package*).

Verwar ook niet met JavaFX versie 1. Dit was geen Java-bibliotheek, maar een afzonderlijke programmeertaal.

Bovenstaande code is nog niet volledig. Zo moet bijvoorbeeld de *choice box* de keuzes 1–6 meekrijgen en moet initieel de waarde 6 worden geselecteerd.

```
for (int i = 1; i <= 6; i++) {  
    choiceBox.getItems().add(i);  
}  
choiceBox.setValue(6);
```

Lettertypes, spatiëring, marges, centrering en achtergrond worden bepaald door een *style sheet* in CSS die we aan het paneel hechten:

```
borderPane.getStylesheets().add(  
    "be/ugent/objprog/quickpick/quickpick.css");
```

De *style sheet* moet onderscheid kunnen maken tussen het label dat als titel wordt gebruikt en de andere labels in het venster (in de *VBox* en als onderdeel van de *choice box*). Daarom geven we het titellabel een CSS-identificatie mee

```
label.setId("titel");
```

We moeten er nu enkel nog voor zorgen dat het programma passend reageert op het indrukken van de knop.

In FXML gaat dat min of meer vanzelf: je geeft in het FXML-bestand aan welke methode van de partnerklasse er moet worden opgeroepen. In ons voorbeeld moeten we echter iets meer werk verrichten. Het indrukken van de knop veroorzaakt een actiegebeurtenis die we opvangen met een actieluisteraar. Concreet betekent dit dat we een object nodig hebben van een klasse die de interface *EventHandler<ActionEvent>* implementeert.

De interface *EventHandler<ActionEvent>* bezit slechts één methode:

```
public void handle (ActionEvent e)
```

Deze methode wordt uitgevoerd wanneer de gebeurtenis zich voordoet. De parameter *e* kan je eventueel gebruiken om meer te weten te komen over de context waarin de gebeurtenis plaatsvond. Je kan opvragen op welk tijdstip de gebeurtenis zich voordoet, of er tegelijk met het indrukken van de knop een SHIFT-, ALT- of CTRL-toets was ingedrukt, enz. In de praktijk wordt deze parameter echter weinig gebruikt.

We schrijven dus een (binnen)klasse *ButtonEventHandler* die de interface implementeert:

```

private static class ButtonEventHandler
    implements EventHandler<ActionEvent> {
    ...
    @Override
    public void handle(ActionEvent event) {
        vbox.getChildren().clear();
        for (int i = 0; i < choiceBox.getValue(); i++) {
            Label label = new Label();
            label.setText(quickPick());
            vbox.getChildren().add(label);
        }
    }
}

```

Deze klasse heeft twee velden *vbox* en *choiceBox* die we meegeven aan de constructor

```

private static class ButtonEventHandler
    implements EventHandler<ActionEvent> {
    private final ChoiceBox<Integer> choiceBox;
    private final VBox vbox;
    private final LottoGenerator lottoGenerator = new LottoGenerator();

    public ButtonEventHandler(ChoiceBox<Integer> choiceBox, VBox vbox) {
        this.choiceBox = choiceBox;
        this.vbox = vbox;
    }
    ...
}

```

De methode *quickpick* genereert een string met willekeurige Lottogetallen.

```

private final LottoGenerator lottoGenerator;

private String quickPick() {
    int[] result = lottoGenerator.generateNumbers();
    StringBuilder b = new StringBuilder();
    b.append(result[0]);
    for (int i = 1; i < result.length; i++) {
        b.append(" - ");
        b.append(result[i]);
    }
    return b.toString();
}

```

Ze maakt gebruik van een hulpklasse *Lottogenerator*:

```
public void LottoGenerator {  
  
    public LottoGenerator() {  
        ...  
    }  
  
    public int[] generateNumbers () {  
        ...  
    }  
}
```

De methode *generateNumbers* geeft een tabel van zes willekeurige getallen terug tussen 1 en 45, in stijgende volgorde<sup>1</sup>.

Opdat JavaFX zou weten welke gebeurtenis er door welk object moet worden verwerkt, mag je niet vergeten de verwerker te *registreren* bij de component die de gebeurtenis genereert (in dit geval dus bij de knop) — het is niet voldoende om enkel de klasse te schrijven.

```
button.setOnAction(new ButtonEventHandler(choiceBox, vbox));
```

Tot slot verzamelen we nu al deze code in een klasse met de volgende *start*-methode:

```
public void start(Stage stage) throws IOException {  
    BorderPane borderPane = new BorderPane();  
    ...  
    Scene scene = new Scene(borderPane);  
    stage.setScene(scene);  
    stage.setTitle ("");  
    stage.show();  
}
```

De implementatie van deze code is zeer gelijkaardig aan wat we zouden doen als we een FXML-bestand gebruiken. Het belangrijkste verschil is nu dat we het paneel dat alle componenten bevat (het *borderPane*) zelf construeren in plaats van dit door de *FXMLLoader* te laten doen. We hebben ook geen afzonderlijke partnerklasse nodig.

---

<sup>1</sup>Je kan de implementatie van deze klasse als een oefening opvatten. Let op, het is niet zo eenvoudig om alle mogelijke zestallen te genereren met dezelfde waarschijnlijkheid!

## 4.2. Een eigen component definiëren

Het feit dat alles in één enkele klasse is geplaatst (behalve de *LottoGenerator*), is niet erg overzichtelijk. We zullen daarom de programmacode herschikken door de introductie van een nieuw klasse *QuickPickPane*.

Die klasse stelt een ‘nieuwe’ component voor die precies overeenkomt met de inhoud van het programmavenster. Voor het hoofdprogramma gebruiken we dan een afzonderlijke (kleine) klasse (zoals bij FXML) met de volgende eenvoudige *start*-methode:

```
public void start(Stage stage) throws IOException {  
    Scene scene = new Scene(new QuickPickPane());  
    stage.setScene(scene);  
    stage.setTitle("");  
    stage.show();  
}
```

De hoofdbrok van de Java-code is verplaatst naar *QuickPickPane*. Een object van die klasse is weinig meer dan een *BorderPane* waaraan alvast de juiste componenten zijn toegevoegd. De klasse *QuickPickPane* is een uitbreiding van *BorderPane* en de componenten voegen we toe als onderdeel van haar constructor.

```
public class QuickPickPane extends BorderPane {  
  
    public QuickPickPane () {  
        Label label = new Label("Lotto Quick Pick");  
        label.setId("titel"); // voor CSS  
  
        HBox hbox = new HBox();  
        Button button = new Button("Quick Pick!");  
        button.setOnAction(this::handleButton);  
  
        this.choiceBox = new ChoiceBox<>();  
        for (int i = 1; i <= 6 ; i++) {  
            choiceBox.getItems().add(i);  
        }  
        choiceBox.setValue(6);  
        hbox.getChildren().addAll(choiceBox, button);  
  
        this.vbox = new VBox();  
        vbox.getChildren().add(new Label ("- nog geen keuze -"));  
    }  
}
```

```

    setTop(label);
    setCenter(vbox);
    setBottom(hbox);

    getStylesheets().add("be/ugent/objprog/quickpick/quickpick.css");

}
}

```

Deze constructie laat ons toe om de binnenklasse *ButtonEventHandler* sterk te vereenvoudigen en uiteindelijk zelfs te vervangen door een lambda.

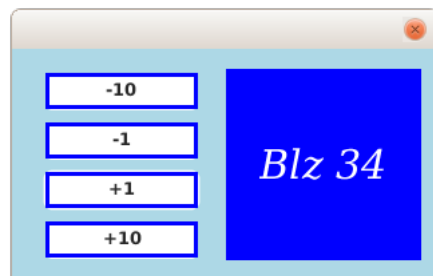
### 4.3. Gelijkaardige componenten

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.buttons.* ]

In een JavaFX-toepassing kan je vaak hetzelfde effect bekomen op verschillende manieren. In de volgende paragrafen illustreren we dit aan de hand van een eenvoudig voorbeeld<sup>2</sup> dat als prototype zou kunnen dienen voor een meer ingewikkelde documentenbrowser.

De toepassing die hiernaast is afgebeeld, bevat een aantal knoppen waarmee je vooruit of achteruit kunt bladeren in een document, hetzij bladzijde per bladzijde of in sprongen van tien bladzijden.

De vier knoppen hebben een zeer gelijkaardige werking, en we willen duplicatie van code vermijden bij hun implementatie. Bovendien zou het goed zijn als we het programma niet al te veel moeten veranderen wanneer we bijvoorbeeld ook nog een '+100'- en een '-100'-knop willen toevoegen. We gebruiken een FXML-bestand en een partnerklasse.



Er zijn een aantal opties. Gebruiken we een gebeurtenisverwerkende methode, of liever een echte klasse? In het eerste geval: gebruiken we dan vier verschillende methodes, of telkens dezelfde? En als we een klasse gebruiken: vier verschillende klassen, één klasse met vier verschillende objecten of één object dat alle knoppen voor zijn rekening neemt?

De optie die het minste denkwerk vraagt, is wellicht het gebruik van vier verschillende

<sup>2</sup>De partnerklassen in onze voorbeelden hebben publieke velden. Dit is de enige context waarin we toelaten dat velden publiek blijven. Het is trouwens mogelijk om dit soort publieke velden helemaal te vermijden door ze te annoteren met *@FXML*.

gebeurtenisverwerkende methodes: we associëren met elke knop een afzonderlijke methode uit de partnerklasse:

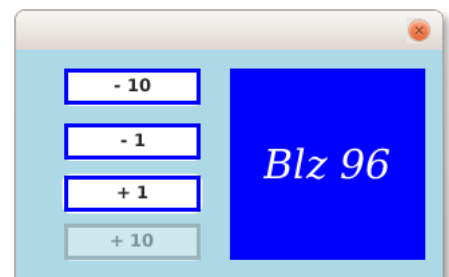
```
public class ButtonsCompanion {  
  
    public Label blzLabel;  
  
    private int blz;  
  
    public ButtonsCompanion () {  
        blz = 1;  
    }  
  
    public void doMinus1 () {  
        blz --;  
        blzLabel.setText ("Blz " + blz);  
    }  
  
    // doMinus10 en doPlus1 analoog  
  
    public void doPlus10 () {  
        blz += 10;  
        blzLabel.setText ("Blz " + blz);  
    }  
}
```

Hier hebben we heel wat duplicatie van code, wat in dit eenvoudige geval misschien nog niet zo erg is, maar zoals je goed weet: eenvoudige programma's blijven niet eenvoudig!

Inderdaad, we willen al meteen een eerste verandering aanbrengen: het document dat we simuleren heeft exact 100 bladzijden en we willen niet dat de gebruiker voorbij die grens kan doorklikken.

Daarom willen we de knoppen deactiveren (Engels: *to disable*) wanneer ze ons op een ongeldig bladzijde-nummer zouden brengen.

Je deactiveert een knop door *setDisable(true)* op te roepen. Terug activeren gebeurt op dezelfde manier, maar dan met **false** als argument.





```

public class ButtonsCompanion {

    public Button minus1;
    ...
    public Button plus10;

    public Label blzLabel;

    private int blz;

    public ButtonsCompanion() {
        blz = 1;
    }

    public void initialize() {
        adjustBlzLabel(0);
    }

    public void adjustBlzLabel(int increment) {
        blz += increment;
        blzLabel.setText("Blz " + blz);
        minus1.setDisable(blz - 1 < 1);
        minus10.setDisable(blz - 10 < 1);
        plus1.setDisable(blz + 1 > 100);
        plus10.setDisable(blz + 10 > 100);
    }

    public void doMinus1() {
        adjustBlzLabel(-1);
    }

    ...
}

```

We gebruiken nog steeds vier verschillende methodes, maar om duplicatie van code te beperken, hebben we de gemeenschappelijke code samengebracht in een methode *adjustBlzLabel*. Deze methode neemt als parameter het ‘increment’, t.t.z., het aantal waarmee we het bladzijdenummer moet verhogen.

Het zou mooi zijn als we de methode *adjustBlzLabel*, mét haar argument, rechtstreeks zouden kunnen aangeven in het FXML-bestand. Dit is helaas niet mogelijk. De enige parameter die voor een gebeurtenisverwerkende methode is toegestaan, is van het type *Event*.

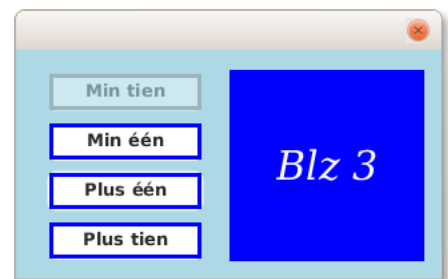
Dit biedt ons in dit geval een uitweg om het toch met één gebeurtenisverwerkende methode te doen:

```
public void doButton(ActionEvent event) {  
    Button sourceButton = (Button) event.getSource();  
    adjustBlzLabel(Integer.parseInt(sourceButton.getText()));  
}
```

Aan de parameter *event* kunnen we immers haar bron opvragen — in dit geval één van de vier knoppen. Via dit opschrift van die knop komen we dan te weten wat het increment is dat we moeten doorgeven aan *adjustBlzLabel*.

We willen het gebruik van een dergelijk achterpoortje echter niet stimuleren. In dit geval is dit enkel gelukt omdat het opschrift van de knoppen door Java als een geheel getal kan geïnterpreteerd worden.

Nog een geluk dus dat we geen spatie hebben gelaten tussen het plus- of minteken en het getal, of dat we geen opschriften hebben gebruikt zoals in de schermafdruck hiernaast!



Er bestaat een variant van bovenstaande methode die wél toelaat om andere opschriften aan de knoppen te geven. JavaFX laat toe om bij een knop, bij elke component eigenlijk, een object te registreren als zogenaamde ‘*user data*’. Op die manier kunnen we dus het juiste aantal bladzijden opslaan bij elke knop. Dit doe je met *setUserData*:

```
public void initialize() {  
    ...  
    plus1.setUserData(1);  
    plus10.setUserData(10);  
    minus1.setUserData(-1);  
    minus10.setUserData(-10);  
}
```

We vragen dit aantal dan terug op als onderdeel van *doButton*:

```
public void doButton(ActionEvent event) {  
    Button sourceButton = (Button) event.getSource();  
    int increment = (Integer)sourceButton.getUserData();  
    adjustBlzLabel(increment);  
}
```

Het retourtype van *getUserData* is *Object*. We hebben dus een *cast* nodig naar *Integer*.

Je kan de initialisatie van de *user data* ook in het FXML-bestand opnemen in plaats van in de *initialize* :

```
<Button userData="-10" fx:id="minus10"... text="Min 10" />
<Button userData="-1" fx:id="minus1" ... text="Min 1" />
<Button userData="1" fx:id="plus1" ... text="Plus 1" />
<Button userData="10" fx:id="plus10" ... text="Plus 10" />
```

JavaFX denkt nu echter dat de *user data* van het type *String* is, en dus moet je *doButton* overeenkomstig aanpassen:

```
public void doButton(ActionEvent event) {
    Button sourceButton = (Button) event.getSource();
    String userData = (String)sourceButton.getUserData();
    adjustBlzLabel(Integer.parseInt(userData));
}
```

Je kan in het FXML-bestand de *user data* ook op de volgende manier initialiseren, en dan wordt wel het juiste type gebruikt:

```
<Button fx:id="minus10"... text="Min 10">
    <userData>
        <Integer fx:value="-10"/>
    </userData>
</Button>
...
```

maar dat vraagt toch heel wat lijnen voor wat in principe slechts een eenvoudige initialisatie is.

## 4.4. Een eigen *Button*-klasse

In de volgende bladzijden gooien we het over een andere boeg: in plaats van de standaard *Button*-component te gebruiken, schrijven we een eigen uitbreiding ervan, de *IncrementButton*.

```
public class IncrementButton extends Button {  
  
    private final int increment;  
  
    public IncrementButton (int increment, String caption) {  
        super (caption);  
        this.increment = increment;  
    }  
  
    public int getIncrement () {  
        return increment;  
    }  
  
    public void disableIfNecessary (int blz) {  
        setDisable(blz + increment < 1 || blz + increment > 100);  
    }  
}
```

We hebben aan de standaardklasse een veld *increment* toegevoegd dat we kunnen instellen bij constructie en opvragen met een overeenkomstige getter-methode (*getIncrement*). Nu hoeven we geen trucjes meer te gebruiken om van een knop te weten te komen wat het bijbehorende increment is. We kunnen het hem immers rechtstreeks vragen.

De actiegebeurtenis wordt nu als volgt verwerkt:

```
public void handle(ActionEvent event) {  
    IncrementButton sourceButton = (IncrementButton) event.getSource();  
    adjustBlzLabel(sourceButton.getIncrement());  
}
```

Dit is echter nog maar een deel van het verhaal. De vier knopobjecten moeten immers ook aangemaakt worden en op de juiste plaats in de scène terechtkomen. Hiervoor bestaan er opnieuw verschillende mogelijkheden.

Een eerste optie is om de vier knoppen niet in het FXML-bestand op te nemen, maar pas aan te maken in de *initialize*-methode van de partnerklasse:

```

public VBox buttonPanel;

private IncrementButton[] buttons;

public void initialize() {

    buttons = new IncrementButton[]{
        new IncrementButton(-10, "Min tien"),
        new IncrementButton(-1, "Min één"),
        new IncrementButton(+1, "Plus één"),
        new IncrementButton(+10, "Plus tien")
    };

    for (Button button : buttons) {
        buttonPanel.getChildren().add(button);
        button.setOnAction(...);
    }

    adjustBlzLabel(0);
}

```

(Merk op dat we de knoppen voor het gemak in een array hebben geplaatst.)

Omdat de knoppen nu niet in het FXML-bestand zijn opgenomen, moeten we ook zelf de gebeurtenisverwerker bij de knop registreren, zoals in paragraaf §4.1.

Er zijn opnieuw verschillende alternatieven: de gebeurtenisverwerkende klasse kan een anonieme klasse zijn, een lambda, een binnenklasse, een externe klasse of de partnerklasse zelf. De *handle*-methode van deze klasse hebben we hierboven reeds afgedrukt. In elk van die gevallen volstaat het om slechts één object van de klasse te maken en dit zelfde object bij alle knoppen te registreren.

Ook de methode *adjustBlzLabel* van de partnerklasse hebben we enigzins gestroomlijnd.

```

public void adjustBlzLabel(int increment) {
    blz += increment;
    blzLabel.setText("Blz " + blz);
    for (IncrementButton button : buttons) {
        button.disableIfNecessary(blz);
    }
}

```

De programmalogica die bepaalt of een knop al dan niet geactiveerd moet zijn, hebben we

verplaatst naar de klasse *IncrementButton*.

Dat we nu een gedeelte van de scènegraaf opbouwen in de *initialize*-methode, is niet zo elegant — de scènegraaf moet zo veel mogelijk de verantwoordelijkheid blijven van de grafische ontwerper en niet van de programmeur.

Gelukkig mag je ook zelfgemaakte componenten opnemen in een FXML-bestand. Waar er vroeger ‘<Button...>’ stond, schrijven we nu ‘<IncrementButton...>’. Bovendien mogen velden waarvoor er zowel een getter als een setter bestaat (*increment* in ons geval), als bijkomend attribuut gebruikt worden.

We schrijven dus:

```
<IncrementButton increment="-10" fx:id="minus10" ... text="Min 10" />
<IncrementButton increment="-1"  fx:id="minus1"  ... text="Min 1"  />
<IncrementButton increment="+1"  fx:id="plus1"   ... text="Plus 1"  />
<IncrementButton increment="+10" fx:id="plus10"  ... text="Plus 10" />
```

Bovendien moet je bovenaan het FXML-bestand een ‘import’-opdracht toevoegen die aangeeft waar de nieuwe klasse *IncrementButton* kan gevonden worden.

```
<?import be.ugent.objprog.buttons.*?>
```

Zo kunnen we uiteindelijk de ganse *initialize* vermijden, en kunnen we weer een gebeurtenisverwerkende methode gebruiken in plaats van een klasse.

De *Scene Builder* laat toe om met zelfgemaakte componenten te werken<sup>3</sup> maar dit vraagt redelijk wat bijkomend werk: je moet een JAR-archief maken met daarin (de gecompileerde versies van) je componenten en dit archief bij de *Scene Builder* importeren als componentenbibliotheek. Het is vaak eenvoudiger om te werken op de volgende manier:

- Gebruik *Scene Builder* om een voorontwerp te maken van je scène. Waar uiteindelijk zelfgemaakte componenten moeten komen (*IncrementButton*), plaats je de standaardcomponenten waarvan ze zijn afgeleid (*Button*).
- Bewaar je scène als FXML-bestand en pas het met de hand aan door waar nodig de namen van componentklassen aan te passen (*Button* wordt *IncrementButton*). Voeg ook de nodige attributen toe (*increment="..."*) en vergeet de import-opdrachten niet.

Latere veranderingen in het ontwerp kan je dan eventueel rechtstreeks aanpassen in het FXML-bestand.

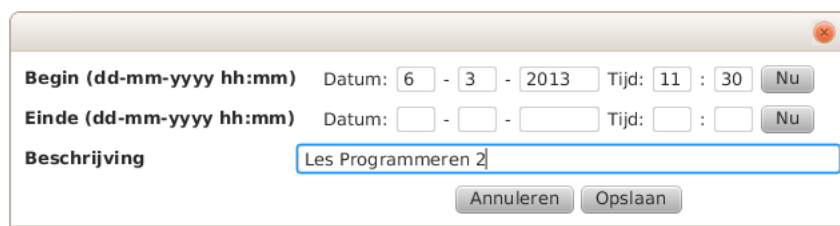
---

<sup>3</sup>Op dit moment is dit enkel mogelijk met de alleenstaande versie van *Scene Builder* maar niet met de versie die is ingebed in IntelliJ IDEA.

## 4.5. Componenten opnieuw gebruiken

[ Voorbeelden uit deze paragraaf vind je in *be.ugent.objprog.datetime* en *be.ugent.objprog.tracker*. ]

Onderstaande schermafdruck toont een dialoogvenster uit een ‘*time tracker*’, een toepassing waarmee je kan bijhouden hoeveel tijd je besteedt aan bepaalde taken.



Je merkt onmiddellijk dat de twee bovenste ‘rijen’ van dit dialoogvenster nagenoeg identiek zijn. Een goede software-ontwikkelaarspraktijk indachtig, willen we dit bereiken zonder al te veel knip- en plakwerk. We zullen daarom zelf een nieuwe component ontwikkelen, *DateTime* genaamd, waarvan we dan twee instanties in het venster kunnen opnemen. (Het venster zelf zullen we een *TimeSlotDialog* noemen.)

De *DateTime*-component bevat naast de velden waarin je datum en tijd kunt intikken ook een knop ‘Nu’ die als effect heeft dat het huidige tijdstip in de velden wordt ingevuld.

Datum: 6 - 3 - 2013 Tijd: 11 : 30 Nu

De combinatie van datum en tijd die je ingeeft, wordt intern voorgesteld als een object van het type *LocalDateTime* (uit het pakket *java.time*). Het is toegelaten om alle datumvelden leeg te laten, en in dat geval krijgt de *LocalDateTime* als waarde **null**.

We implementeren de volgende methodes:

- Een methode *setLocalDateTime(LocalDateTime dateTime)* waarmee je de afgebeelde datum en tijd instelt. We laten ook **null** als argument toe.
- Een methode *getLocalDateTime* die de ingevoerde datum en tijd teruggeeft als een object van het type *LocalDateTime* (of **null**).

Hierbij treedt een kleine complicatie op: niet alles wat je in de tekstvelden intikt, is een geldige datum: je tikt letters in in plaats van cijfers, je geeft een onbestaande datum in (een dertiende maand, een dertigste februari, ...) of je laat enkele velden leeg, maar niet allemaal. In deze gevallen zal *getLocalDateTime* ook **null** teruggeven.

Om het onderscheid te maken met een geldige **null** bieden we een methode *isValid* aan die aangeeft of de waarde die werd teruggeven door *getLocalDateTime* al dan niet correspondeerde met een geldige invoer. Het is pas bij het oproepen van *getLocalDateTime* dat de component zal controleren of de inhoud van de tekstvelden een geldige datum en tijd voorstelt. Het heeft dus geen zin om *isValid* op te roepen vóór *getLocalDateTime*.

Het is niet zo moeilijk (en een goede oefening) om een dergelijke component te programmeren, ware het niet dat we er ook nog een bijkomende functionaliteit aan hebben toegevoegd: wanneer *getLocalDateTime* merkt dat de ingegeven datum- en tijdvelden geen geldige waarde voorstellen, dan kleuren we de tekst in die velden rood. Zodra er echter opnieuw iets aan de inhoud van één van die velden verandert, krijgt de tekst van alle velden weer zijn originele kleur.

Om te reageren op veranderingen aan de inhoud van een tekstveld, luisteren we naar haar *text*-eigenschap. Deze luisteraar wordt dan automatisch uitgevoerd telkens wanneer de inhoud van het tekstveld verandert<sup>4</sup>. We gebruiken in dit geval een eenvoudige lambda als luisteraar en hechten die aan alle tekstvelden:

```
textFields = new TextField[] {  
    year, month, day, hours, minutes  
};  
  
for (TextField field : textFields) {  
    field.textProperty().addListener(o -> setValid(true));  
}
```

De methode *setValid* moet niet alleen de waarde van het Booleaans veld *valid* instellen, maar moet meteen ook de tekstkleur aanpassen van alle tekstvelden (met behulp van CSS-stijlklassen). Deze methode wordt ook aangeroepen door *getLocalDateTime*.

In het FXML-bestand van *TimeSlotDialogCompanion* willen we graag het volgende opnemen:

```
<DateTime fx:id="beginDateTime" ... />  
...  
<DateTime fx:id="endDateTime" ... />
```

zoals we ook eerder bij *IncrementButton* hebben gedaan (zie §4.4).

We kunnen inderdaad een klasse *DateTime* schrijven die de klasse *HBox* uitbreidt, maar in tegenstelling tot wat we bij *IncrementButton* hebben gedaan, willen we dit keer de

---

<sup>4</sup>Meer over luisteraars en JavaFX-eigenschappen in hoofdstuk 5.



*FXMLLoader* gebruiken om de betreffende *HBox* op te vullen met componenten. Gelukkig heeft JavaFX dit scenario voorzien en hoeven we slechts enkele aanpassingen te maken.

Ten eerste mag in het FXML-bestand van de *DateTime*-component het buitenste element (het wortel-element) niet langer *HBox* heten, en ook niet *DateTime*, maar moet je hiervoor een speciaal `fx:root`-element gebruiken:

```
<fx:root type="HBox" fx:id="hbox" ... >
    ...
    <Label text="Datum:" />
    <TextField fx:id="day"... />
    <Label text="-" />
    <TextField fx:id="month"... />
    ...
    <Button onAction="#nowPushed" ... />
    ...
</fx:root>
```

Behalve de eerste en de laatste regel, is de inhoud van het FXML-bestand precies hetzelfde als zou je een *HBox*-paneel willen aanmaken met daarin de nodige elementen. Het `type`-attribuut van het `fx:root`-element is de naam van de bovenklasse die we zullen gebruiken voor onze nieuwe component.

De constructor van de nieuwe *DateTime*-component ziet er nu zo uit:

```
private DateTimeCompanion companion;

public DateTime() {
    try {
        FXMLLoader loader = new FXMLLoader(
            DateTime.class.getResource("DateTime.fxml"));
        loader.setRoot(this);
        this.companion = new DateTimeCompanion();
        loader.setController(companion);
        loader.load();
    } catch (IOException exception) {
        throw new RuntimeException(exception);
    }
}
```

Let hier vooral op de opdracht `loader.setRoot(this)`. Hiermee geven we aan dat de loader straks bij het uitvoeren van `loader.load()` als wortelknoop van de scènegraaf het `this`-object moet gebruiken en niet zelf een dergelijk object moet aanmaken, zoals voorheen.

In JavaFX-voorbeelden van andere auteurs zal je soms zien dat men de *DateTime*-component zelf als partnerklasse gebruikt voor het FXML-bestand. Wij kiezen er echter voor om nog steeds een afzonderlijke partnerklasse te gebruiken, om ze zoveel mogelijk voor de eindgebruiker te verbergen.

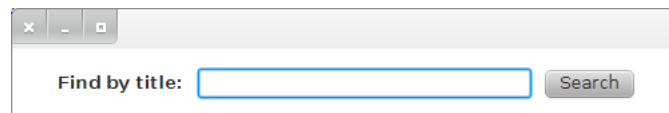
## 4.6. Internationalisatie

[ De voorbeelden uit deze paragraaf vind je in *be.ugent.objprog.i18n*. ]

Een goede (GUI-)toepassing is op een zodanige manier ontworpen dat ze gemakkelijk kan worden aangepast aan verschillende talen en regio's. Een geïnternationaliseerde toepassing kan op een eenvoudige manier worden geconfigureerd voor een nieuwe taal of een nieuw land zonder dat de gecompileerde code moet gewijzigd worden.

Dit betekent onder andere dat berichten, labels, opschriften van knoppen en titelbalken niet hardgecodeerd mogen zijn, maar moeten worden opgeslagen buiten het programma. Ook bepaalde conventies, zoals de vorm waarin een datum wordt afgedrukt, of het gebruik van een decimale punt of komma, mogen niet in de broncode van het programma op voorhand zijn vastgelegd<sup>5</sup>.

Java en JavaFX bieden hierbij heel wat ondersteuning en gebruiken hiervoor onder andere de *eigenschapsbestanden* die we al eerder hebben ontmoet (in §3.7). Als eerste voorbeeld bespreken we een programmafragment waarin een 'zoek'-component wordt opgebouwd. Deze component bestaat uit een tekstveld met bijbehorend label en een zoekknop. De tekst van het label en het opschrift van de knop willen we graag tweetalig maken (Nederlands/Engels).



Hieronder drukken we de *niet*-geïnternationaliseerde code af waarmee een dergelijke component kan worden aangemaakt:

```
textField = new TextField ();
button = new Button("Search");

panel = new HBox ();
...
```

---

<sup>5</sup>Er zijn nog meer internationale verschillen waarvoor Java ondersteuning biedt, maar die we hier niet bespreken: taalafhankelijk alfabetisch sorteren, niet-Europese schriften, schrijven van rechts naar links, of van boven naar onder, ...

De Engelse term *internationalization* wordt vaak afgekort tot 'i18n' (omdat er 18 letters staan tussen de begin-i en de eind-n). Men schrijft ook 'l10n' voor het verwante begrip *localization*.

```
panel.getChildren().addAll(  
    new Label("Find by title:"),  
    textField,  
    button);
```

De eerste internationalisatiestap bestaat erin om een eigenschapsbestand op te stellen met daarin alle taalafhankelijke elementen.

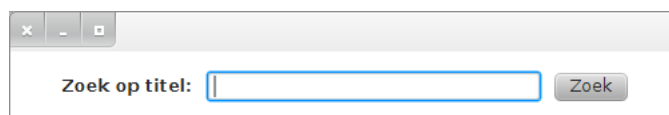
```
# Engelse versie  
fieldLabel=Find by title:  
buttonCaption=Search
```

We noemen dit bestand bijvoorbeeld *i18n.properties*.

De sleutels zijn taal-onafhankelijk maar de waarden verschillen van taal tot taal. De Nederlandse versie ziet er zo uit:

```
# Nederlandse versie  
fieldLabel=Zoek op titel:  
buttonCaption=Zoek
```

Dit bestand noemen we *i18n\_nl.properties*. De basisnaam van het bestand is dezelfde als bij de Engelse versie, maar nu hebben we er het taalachtervoegsel '*nl*' aan toegevoegd. (Straks geven we meer details over de gebruikte conventies.)



Om dergelijke eigenschapsbestanden die specifiek bedoeld zijn voor internationalisatie in ons programma aan te spreken, gebruiken we niet rechtstreeks de *Properties*-klasse zoals in paragraaf §3.7, maar de klasse *ResourceBundle* uit het pakket *java.util*.

We halen eerst een zogenaamde *bundel* op:

```
ResourceBundle bundle  
    = ResourceBundle.getBundle("be/ugent/objprog/i18n/i18n");
```

(Merk op dat we een klassenmethode *getBundle* gebruiken om de bundel op te halen en niet de constructor **new** *ResourceBundle*(...).

Als parameter geef je de naam op van het eigenschapsbestand, zonder *.properties*-extensie en zonder taalaanduiding (*\_nl*). Het bestand wordt opgezocht in het class path. Er wordt steeds gezocht vanaf de wortel van het class path, en niet vanuit de directory waarin de ‘huidige’ klasse zich bevindt!

Om later een string uit deze bundel op te halen, gebruik je de methode *getString* met de sleutel als argument. In ons voorbeeld wordt dit dus

```
ResourceBundle bundle =  
    ResourceBundle.getBundle("be/ugent/objprog/i18n/i18n");  
  
textField = new TextField ();  
button = new Button(bundle.getString("buttonCaption"));  
  
HBox panel = new HBox ();  
...  
panel.getChildren().addAll(  
    new Label (bundle.getString("fieldLabel")),  
    textField,  
    button);
```

De methode *getString* zoekt de opgegeven sleutel in een *properties*-bestand dat zo goed mogelijk met de taal en het land van de gebruiker overeenkomt. Taal en land worden door het besturingssysteem bijgehouden als twee korte codes<sup>6</sup>:

- Een internationaal overeengekomen code die de taal aangeeft: *en* voor Engels, *nl* voor Nederlands, ...
- Een internationale code die het land aangeeft: *FR* voor Frankrijk, *BE* voor België, ...

Voor een Belg die Nederlands spreekt, wordt er achtereenvolgens gezocht in de bestanden met de volgende namen:

```
be/ugent/objprog/i18n/i18n_nl_BE.properties  
be/ugent/objprog/i18n/i18n_nl.properties  
be/ugent/objprog/i18n/i18n.properties
```

Java gebruikt het eerste bestand in de lijst waarin de gevraagde sleutel voorkomt. Het bestand *i18n.properties* bevat met andere woorden de defaultwaarden (in ons geval zijn

---

<sup>6</sup>In sommige gevallen kunnen er nog bijkomende codes worden gebruikt, maar dat komt in de praktijk niet zoveel voor, zeker wanneer je je beperkt tot Europese talen.

die toevallig in het Engels opgesteld), terwijl *i18n.nl.properties* enkel gebruikt wordt wanneer we met een Nederlandstalige gebruiker te doen hebben.

Wanneer je de keuze niet aan het besturingssysteem wil overlaten, kan je ze ook zelf instellen. Dit doe je met een object van het type *Locale*. Deze klasse (uit het pakket *java.util*) heeft een aantal verschillende constructoren waarvan de parameters overeenkomen met de codes die we hierboven hebben opgesomd:

```
public Locale (String taal);  
public Locale (String taal, String land);
```

Je kan een dergelijk object dan doorgeven als tweede parameter aan *getBundle*:

```
Locale locale = new Locale ("nl", "BE");  
ResourceBundle bundle  
    = ResourceBundle.getBundle ("be/ugent/objprog/i18n/i18n", locale);
```

Er is hier een kleine complicatie: als je een *locale* opgeeft voor een taal die niet door jouw programma wordt ondersteund, bijvoorbeeld '**new** *Locale*("fr")' terwijl je geen bestand *i18n.fr.properties* hebt voorzien, dan zal Java de bundel gebruiken die overeenkomt met de standaardtaal die door het besturingssysteem is ingesteld, en niet de 'lege' locale. Er wordt dus niet noodzakelijk teruggevallen op *i18n.properties*<sup>7</sup>.

We hebben in bovenstaand voorbeeld de scènegraaf zelf opgebouwd, maar ook wanneer je dit doet met de *FXMLLoader* kan je de ingebouwde internationalisatiefaciliteiten gebruiken. Bij het inladen van een FXML-bestand geef je dan een *ResourceBundle* mee als bijkomende parameter:

```
Parent root = FXMLLoader.load(  
    I18nMain.class.getResource("I18nFX.fxml"),  
    ResourceBundle.getBundle("be/ugent/objprog/i18n/i18n")  
);
```

In dat geval zal elke attribuutwaarde die met een procent begint, automatisch in de bundel worden opgezocht. Het FXML-bestand voor ons lopend voorbeeld, ziet er nu zo uit:

```
<HBox prefHeight="-1.0" prefWidth="500.0" ... >  
  <Label text="%fieldLabel" />  
  <TextField />  
  <Button text="%buttonCaption" />  
  ...  
</HBox>
```

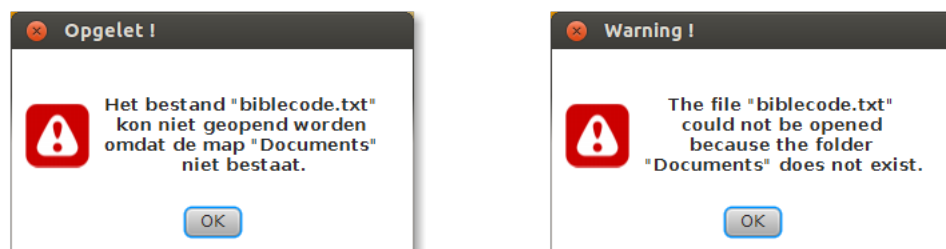
---

<sup>7</sup>Je kan dit gedrag wijzigen via *ResourceBundle.Control*, zie elektronische documentatie.

Let op de `text`-attributen van het label en de knop. Die verwijzen rechtstreeks naar de sleutels in de eigenschapsbestanden.

In bovenstaand voorbeeld was het altijd mogelijk om opschriften en teksten door één enkele sleutel voor te stellen. De praktijk is echter niet altijd zo eenvoudig. Vaak worden berichten samengesteld uit verschillende delen: vaste stukken die taalgebonden zijn en variabele onderdelen die door het programma werden gegegeneerd.

Wat denk je bijvoorbeeld van de onderstaande berichten:



De naam van het bestand en van de map zijn variabel. De rest van de tekst is vast maar moet in verschillende talen kunnen worden afgebeeld.

Je kan dit soort berichten natuurlijk construeren door de afzonderlijke stukken te concateneren zoals hieronder,

```
ResourceBundle bundle = ResourceBundle.getBundle(...);

String message = bundle.getString("open.error.part.1") +
    fileName + bundle.getString("open.error.part.2") +
    folderName + bundle.getString("open.error.part.3");
```

maar dit is nogal wijldlopijg en werkt bovendien niet altijd. Het aantal stukken vaste tekst hoeft immers niet in alle talen noodzakelijk hetzelfde te zijn en het is zelfs mogelijk dat de volgorde van de variabele gedeelten van taal tot taal verschilt.

Voor dergelijke gevallen voorziet Java de klasse *MessageFormat*. In de eigenschapsbestanden plaats je de volledige tekst met daarin de variabele gedeelten in een speciale notatie:

```
open.error = Het bestand "{0}" kon niet geopend worden omdat \
    de map "{1}" niet bestaat.
```

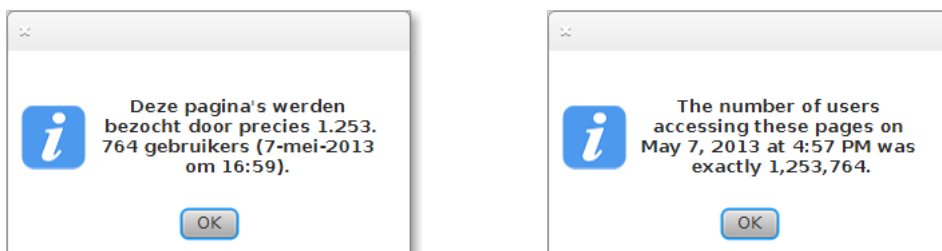
Elk variabel gedeelte krijgt een volgnummer tussen accolades, tellend vanaf 0. Deze volgnummers worden dan later gebruikt als indices in een tabel van objecten die de actuele waarden bevat van wat er moet worden ingevuld.

Het omzetten van dit ‘patroon’ naar de uiteindelijke tekst die moet worden afgebeeld, gebeurt met de (klassen)methode *format* van *MessageFormat*:

```
String fileName = ...           // hier "biblecode.txt"
String folderName = ...         // hier "Documents"
String message = MessageFormat.format (
    bundle.getString ("message"), fieldName, folderName)
);
messageLabel.setText (message);
```

De eerste parameter van *format* is het patroon, de andere parameters zijn de objecten die in het patroon worden ingevuld. In dit voorbeeld zijn dit strings, maar in het algemeen zijn alle objecttypes toegelaten. Voor numerieke types, datums en tijdstippen kan je *MessageFormat* zelfs bijkomende formattering laten doen.

In de twee dialoogvensters hieronder zie je bijvoorbeeld hoe ook het datumformaat en de manier waarop grote getallen worden voorgesteld, zijn aangepast aan de gekozen taal en regio.



Het patroon in de *ResourceBundle* ziet er nu iets ingewikkelder uit:

```
# Engels
message.two=The number of users accessing these pages on {0,date,medium} \
    at {1,time,short} was exactly {2}.

# Nederlands
message.two=Deze pagina''s werden bezocht door precies {2} gebruikers \
    ({0,date,medium} om {1,time,short}).
```

Het is niet mogelijk om *MessageFormat* rechtstreeks te gebruiken in een FXML-bestand. Bij de dialoogvenstervoorbeelden hebben we dan ook de labeltekst pas ingesteld als onderdeel van de *initialize*-methode.

JavaFX helpt ons hier een klein beetje: als de partnerklasse een veld heeft met de naam *resources* van het type *ResourceBundle*, dan zal dit automatisch worden ingevuld met de bundel die je aan de *FXMLLoader* als parameter hebt meegegeven.

Bij elke parameter geef je aan over welk type het gaat (`time`, `date` of `number`) en eventueel bijkomende informatie over hoe het object precies dient te worden opgemaakt (zie de elektronische documentatie van *MessageFormat* voor bijkomende details). We hadden parameter `{2}` hier trouwens ook kunnen schrijven als `{2,number,integer}`.

*MessageFormat* biedt trouwens nog een bijkomende functionaliteit die bijzonder handig kan zijn: een manier om woorden in het enkelvoud of meervoud te schrijven afhankelijk van de context. In plaats van onderstaande gemakkelijksoplossing

```
Er werd(en) 200 bestand(en) gewist.
```

kan je het *MessageFormat* zodanig kiezen dat het resultaat één van de volgende drie berichten is, afhankelijk van het betreffende aantal bestanden:

```
Er werden geen bestanden gewist.  
Er werd één bestand gewist.  
Er werden 200 bestanden gewist.
```

De bundel bevat nu de volgende lijnen:

```
files.deleted = {0,choice,0#Er werden geen bestanden|\n1#Er werd één bestand|\n1<Er werden {0} bestanden} gewist.
```

(De ‘\’-tekens dienen hier enkel om de lijn op te splitsen en zijn niet relevant.)

En in het Engels wordt dit

```
files.deleted = {0,choice,0#No files were|\n1#One file was|1<{0} files were} deleted.
```

In het programma zelf hoef je niets speciaals te voorzien. Er wordt slechts één argument meegegeven aan *MessageFormat.format*:

```
String message = MessageFormat.format (\n    bundle.getString("files.deleted"), nrOfDeletedFiles\n);
```

Meer details vind je in de elektronische documentatie van *MessageFormat* en van *ChoiceFormat*.



## 4.7. Drag en drop

[ Voorbeelden uit deze paragraaf vind je in *be.ugent.objprog.dnd*<sup>8</sup>. ]

Vaak laat een grafische gebruikersinterface toe om ‘elementen’ met de muis vast te pakken, elders naartoe te verslepen, de muis op een andere plaats los te laten, en op die manier een bepaalde actie te veroorzaken (denk aan je browser, of zelfs de volledige desktop van je computer). Dit ‘muisgebaar’ (Engels: *mouse gesture*) heet *drag en drop*<sup>9</sup>. Ook JavaFX biedt hier ondersteuning voor.

Een drag en drop-gebaar bestaat uit verschillende fasen:

- *DRAG DETECTED* Eerst detecteert het systeem dat de gebruiker een drag en drop wil uitvoeren: de muisknop wordt ingedrukt boven een element (de *bron* van de drag en drop) en de muis wordt ‘versleept’ (bewogen terwijl de knop blijft ingedrukt). Er is hier een bijkomend probleem: hetzelfde muisgebaar kan ook iets anders betekenen. Als je een venster groter of kleiner maakt, dan mag dit niet als een drag en drop worden beschouwd.
- *DRAG OVER* Terwijl de muis versleept, passeert ze over andere elementen die daarbij van vorm of kleur kunnen veranderen om aan te geven dat ze een drop accepteren. Ook de cursor kan aangeven of een bepaalde locatie een mogelijke bestemming is voor de drag en drop.
- *DRAG DROPPED* Uiteindelijk wordt de muisknop losgelaten boven een element (het *doel* van de drag en drop).

Bij drag en drop maakt men onderscheid tussen drie verschillende *transfer modes*: *COPY*, *MOVE*, en *LINK*. Deze geven aan welke vorm van overdracht wordt gewenst (door de bron) of is toegelaten (door het doel). De gebruiker kan tijdens de drag en drop de transfer mode aangeven door de CTRL-, SHIFT- of ALT-toets in te drukken (of equivalente toetsen bij MacOS). Niet elke transfer mode is echter bij elke drag en drop toegelaten.

Het is belangrijk te weten dat drag en drop iets is dat door het besturingssysteem wordt geregeld, en dat niet eigen is aan JavaFX of Java. Drag en drop kan gebruikt worden tussen verschillende programma’s die niet allemaal in dezelfde programmeertaal hoeven geschreven te zijn — dit is in de praktijk trouwens hun meest voorkomende toepassing.

Als eerste voorbeeld bespreken we daarom een programma dat toelaat om een afbeel-

---

<sup>8</sup>Op Linux, afhankelijk van de versie van het besturingssysteem en van de versie van JavaFX, moet je misschien bij het uitvoeren van deze programma’s de bijkomende VM-optie `-Djdk.gtk.version=2` opgeven opdat ze zouden werken.

<sup>9</sup>Er bestaat geen geijkte Nederlandse term voor dit gebaar. In Nederland noemt men dit in de volkstaal soms ‘sleur en pleur’.

dingsbestand te slepen, vanuit bijvoorbeeld de verkenner, naar een JavaFX-venster, en het daar dan als afbeelding te tonen.

Het programmavenster bevat een *BorderPane*, genaamd *parent*, met daarop een label. We configureren *parent* als *doel* van een mogelijke drag and drop, en wanneer er een bestand op *parent* wordt gedropt, zetten we dit om naar een *Image* en gebruiken we deze als *graphic* van het label.

Om de drag en drop te implementeren, moeten we twee luisteraars implementeren die overeenkomen met de twee laatste fasen uit het drag en drop-gebaar. We bespreken eerst de laatste fase:

```
parent.setOnDragDropped (event -> {  
    Dragboard db = event.getDragboard();  
    if (db.hasFiles()) {  
        fileToImage(db.GetFiles().get (0));  
    }  
    event.setDropCompleted(true);  
    event.consume();  
});
```

We overlopen dit fragment lijn per lijn.

- De gebeurtenis die we hier verwerken is van het type ‘drag dropped’. Dit komt overeen met de naam die we eerder hebben gegeven aan de corresponderende fase.
- Een *dragboard* is een object dat alle gegevens bevat die tijdens een drag en drop van bron naar doel worden overgeheveld. Dit object wordt opgevuld door de bron tijdens de ‘drag detected’-fase — in dit geval door een ander programma.
- De klasse *Dragboard* biedt standaardondersteuning voor enkel veelgebruikte gegevenstypes, waaronder bestanden. Met *hasFiles* weet je óf er bestanden worden uitgewisseld en met *getFiles* haal je die bestanden op. (Dit is een lijst, want je kan met één drag en drop-beweging tegelijk meerdere bestanden verslepen.)
- De (zelf geschreven) methode *fileToImage* neemt een *File* als argument, zet de inhoud van dit bestand om naar een *Image* en installeert die afbeelding in het label.
- Na afloop roep je best *setDropCompleted* op, om het besturingssysteem te laten weten dat je het dragboard niet meer nodig hebt.
- En ook is het best om als laatste stap *event.consume()* op te roepen, zodat de ‘ouders’ van de doelcomponent niet denken dat de drag en drop voor hen bedoeld was.

Het is niet voldoende om de ‘drag dropped’-gebeurtenis op te vangen, je moet ook op ‘drag

over' reageren, anders zal de 'drag dropped'-gebeurtenis het doel zelfs nooit bereiken.

```
parent.setOnDragOver(event -> {  
    if (event.getDragboard().hasFiles()) {  
        event.acceptTransferModes(TransferMode.COPY);  
        event.consume();  
    }  
});
```

In de tweede fase moeten we aan het besturingssysteem laten weten of we een drop accepteren en met welke transfer mode. Hier laten we enkel *COPY* toe (we willen niet dat het bestand uit de verkenner verdwijnt), en enkel wanneer er bestanden worden versleept, niet wanneer er tekst of een URL wordt versleept (vanuit een browser bijvoorbeeld).

Voor de eerste fase hoeven we hier niets te doen. Die wordt immers in een ander programma (de verkenner) opgestart.

Vaak is het nuttig om drag en drop ook te gebruiken binnenin één en dezelfde JavaFX-toepassing. Vooraleer we hiervan een voorbeeld geven, dienen we echter stil te staan bij het feit dat we niet zomaar elke object van eender welk type kunnen overbrengen van bron naar doel. Opdat dit mogelijk zou zijn, moet het object *serialiseerbaar* zijn<sup>10</sup>.

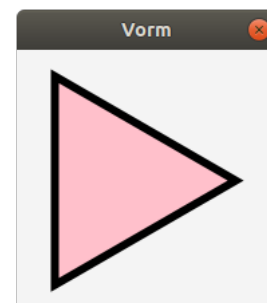
Een object is serialiseerbaar als Java het op een eenduidige manier kan omzetten naar (en van) een reeks bytes (lees: *weet* hoe ze om te zetten). In de praktijk komt het erop neer dat het object moet behoren tot een klasse

- die interface *Serializable* implementeert, en
- waarvan alle velden serialiseerbaar zijn.

Alle primitieve types (**int**, **double**, ...) en opsomtypes zijn serialiseerbaar en ook heel wat standaardtypes (*String*, *File*, *URL*, ..., maar bijvoorbeeld *niet Color*). JavaFX-componenten zijn echter *niet* serialiseerbaar.

In onderstaand voorbeeld laten we toe om 'vormen' te verslepen van en naar bepaalde plaatsen in het programma. Een vorm wordt voorgesteld als een *ShapeLabel* dat als bron dient voor een drag en drop-operatie.

Een vorm is ofwel een cirkel, een vierkant of een driehoek, heeft een bepaalde kleur en is klein of groot.



<sup>10</sup>Deze beperking heeft dan weer als gelukkig gevolg dat je op die manier ook objecten kunt slepen tussen twee *verschillende* JavaFX-programma's.

We zullen *ShapeLabel* laten reageren op de ‘drag detected’-fase van een drag en drop-beweging. M.a.w., het label moet dus iets op het dragboard plaatsen dat we er bij de ‘drag dropped’ van kunnen kopiëren.

Aangezien de ‘vorm’ van een *ShapeLabel* wordt voorgesteld als een object van het type *Shape* (ingesteld als *graphic*-eigenschap van het label), ligt het voor de hand om een dergelijk *Shape*-object op het dragboard te plaatsen. Helaas is *Shape* niet serialiseerbaar.

Om dit op te lossen introduceren we een klasse *ShapeDescription* die alle voornaamste eigenschappen van onze vormen kan beschrijven — vorm, kleur en grootte — en die wel serialiseerbaar is. We zullen met drag en drop dergelijke beschrijvingen verslepen in plaats van de vormen zelf. Bij ‘drag’ slaan we een beschrijving op in het dragboard in de plaats van de *Shape* zelf, en bij ‘drop’ gebruiken we deze beschrijving om een nieuwe *Shape* aan te maken.

```
public class ShapeDescription implements Serializable {  
  
    private ShapeType type;           // enum: SQUARE, CIRCLE of TRIANGLE  
  
    private double red;               // telkens tussen 0.0 en 1.0  
    private double green;  
    private double blue;  
  
    private double size;              // 0.5 of 1.0  
  
    // constructor, getters en setters  
    ...  
}
```

(Merk op dat we de kleur hier opslaan als afzonderlijke rood-, groen- en blauwwaarden, en niet als *Color*, want *Color* is niet serialiseerbaar.)

De klasse *ShapeLabel* bezit een veld *description* dat de overeenkomstige *ShapeDescription* bijhoudt, en een setter om die beschrijving aan te passen, en meteen ook het uitzicht van het label.

We zijn nu voorbereid om de ‘drag detected’-gebeurtenis van *ShapeLabel* op te vangen. We verwerken die als volgt:

```
Dragboard db = startDragAndDrop(TransferMode.COPY_OR_MOVE);  
ClipboardContent content = new ClipboardContent();  
content.put(CUSTOM_DESCRIPTION, description);  
db.setContent(content);  
event.consume();
```

Opnieuw overlopen we dit fragment lijn per lijn.

- De methode *startDragAndDrop* (van de bovenklasse *Node*) geeft aan dat we het indrukken van de muisknop en het verslepen van de muis verwerken als het begin van een drag en drop-gebaar, en niet als een reeks gewone muisgebeurtenissen. Zonder deze oproep wordt er later geen ‘drag over’ of ‘drag dropped’ gesignaleerd. Als resultaat krijgen we een vers dragboard waarop we onze *ShapeDescription* kunnen plaatsen.
- Je kan echter niet zomaar een object op een dragboard plaatsen, maar je moet dit wikkelen in een *ClipboardContent*-object<sup>11</sup>.
- Omdat de inhoud van het dragboard in principe ook door andere (niet-Java) programma’s kan bekeken worden, is het niet voldoende om het object dat we willen verslepen op het dragboard te plaatsen, maar moeten we ook aangeven over welk soort object het gaat — welk ‘gegevensformaat’ (*data format*) we gebruiken.

Er bestaan enkele standaardgegevensformaten (*PLAIN\_TEXT*, *HTML*, *FILES*, ...) voor uitwisseling tussen verschillende programma’s, maar hier gebruiken we een eigen formaat dat we op de volgende manier hebben gedefinieerd:

```
public static DataFormat CUSTOM_DESCRIPTION  
    = new DataFormat("custom/description");
```

De behandeling van de andere fasen van het drag en drop-gebaar lijkt goed op wat we in het eerste voorbeeld hebben gedaan. Voor ‘drag over’ krijgen we het volgende:

```
private void doDragOver(DragEvent event) {  
    Dragboard db = event.getDragboard();  
    if (event.getGestureSource() != this) {  
        if (db.hasContent(CUSTOM_DESCRIPTION)) {  
            event.acceptTransferModes(TransferMode.COPY);  
            event.consume();  
        }  
    }  
}
```

Merk op dat een *ShapeLabel* niet reageert op ‘drag over’ wanneer het zelf de bron is van de drag-gebeurtenis. In plaats van het eerdere ‘*hasFiles*’ kijken we nu of het dragboard iets bevat in het juiste gegevensformaat.

De ‘drag dropped’ wordt op de volgende manier opgevangen:

---

<sup>11</sup>Het dragboard is een speciaal soort clipboard (klembord). Knippen en plakken via het klembord kan ook in JavaFX, maar valt buiten het bestek van deze nota’s.

```

private void doDragDropped(DragEvent event) {
    Dragboard db = event.getDragboard();
    if (db.hasContent(CUSTOM_DESCRIPTION)) {
        description = (ShapeDescription) db.getContent(CUSTOM_DESCRIPTION);
        setShape(description);
        event.setDropCompleted(true);
    }

    else {
        event.setDropCompleted(false);
    }
    event.consume();
}

```

Opnieuw kijken we of het dragboard iets bevat in het juiste gegevensformaat, en in dat geval halen we het corresponderende object op met *getContent*. Dit resultaat moet dan wel naar het juiste type worden ge‘cast’.

Er blijft nog één drag en drop-gebeurtenis over die we niet hebben behandeld<sup>12</sup>, namelijk ‘drag done’. Op het moment dat de drag en drop-gebeurtenis is afgelopen, wordt dit gesignaleerd aan de *bron* (en niet aan het doel zoals bij ‘drag dropped’). Op die manier kan de bron bijvoorbeeld gepast reageren op een *MOVE*:

```

private void doDragDone(DragEvent event) {
    if (event.getTransferMode() == TransferMode.MOVE) {
        setText(" (leeg) ");
        setGraphic(null);
        this.description = null;
    }
}

```

---

<sup>12</sup>Niet helemaal waar — er zijn ook nog ‘drag entered’, ‘drag exited’, ‘drag entered target’ en ‘drag exited target’. Meer informatie vind je in de elektronische documentatie.

## 5. Model/view/controller

In dit hoofdstuk bespreken we het zogenaamde *MVC-patroon*, een bijzondere techniek die onontbeerlijk wordt bij het ontwerp van GUI-toepassingen die meer zijn dan de speelgoedvoorbeelden die we tot nog toe hebben behandeld. We zullen dit ontwerpspatroon vanaf nu regelmatig toepassen.

Ook de JavaFX-bibliotheek zelf maakt er uitvoerig gebruik van, onder andere door de introductie van JavaFX-*eigenschappen* waarvan elke GUI-component er verschillende bezit (zie §5.7).

### 5.1. Model, view en controller

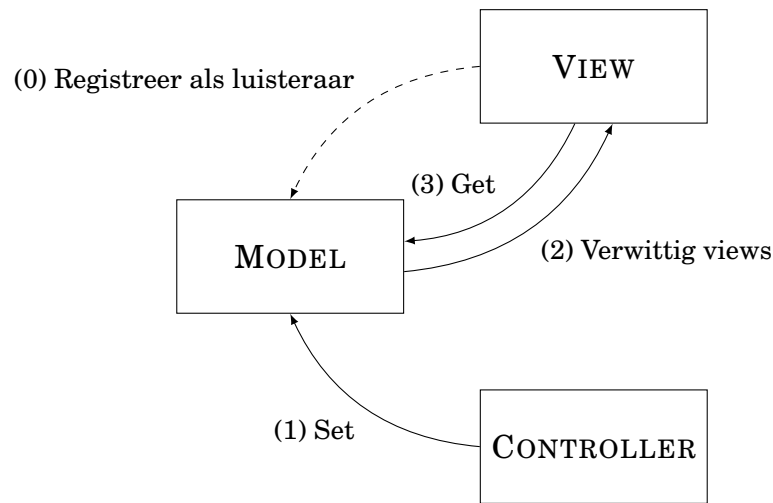
[ Onderstaande voorbeelden vind je in `be.ugent.objprog.buttons`. ]

Het MVC-patroon onderscheidt drie verschillende functionaliteiten in een component van een GUI-toepassing: het *model*, de *controller* en de *view*.

Het *model* bevat de feitelijke informatie die grafisch zal worden afgebeeld: bijvoorbeeld de meetwaarden waarvan een grafiek wordt getoond of de woorden in een keuzelijst. De *view* correspondeert met de grafische voorstelling van de gegevens uit het model: het staafdiagram met de meetwaarden of de keuzelijst zelf. De *controller* zorgt ervoor dat het model kan worden aangepast door interactie met de gebruiker: misschien kan je de staven in het diagram met de muis groter of kleiner maken of kan je elementen verwijderen uit de keuzelijst door een druk op de DELETE-toets.

Het is niet ongebruikelijk dat hetzelfde model verschillende views toelaat: een staafdiagram of een taartdiagram voor dezelfde meetwaarden bijvoorbeeld. Er hoeven ook niet altijd controllers te zijn. Soms is er geen gebruikersinteractie mogelijk met het model, maar wordt dit vanuit het programma aangepast.

Eén van de bedoelingen van het MVC-patroon is om model en view zo goed mogelijk van elkaar te scheiden. Het moet eenvoudig zijn om de interne voorstelling van het model aan te passen zonder dat de view daarom opnieuw moet worden geprogrammeerd en je moet een view kunnen aanpassen zonder dat dit gevolgen heeft voor de klasse die het model voorstelt.



Men zorgt er zelfs voor dat nieuwe, bijkomende views aan een model kunnen worden gehecht zonder dat het model daarvan echt op de hoogte hoeft te zijn. Hiervoor gebruikt men een *luisterarme*mechanisme.

De algemene strategie wordt hierboven geschetst.

- (0) Elke view registreert zichzelf eerst bij het model als luisteraar.
- (1) De controller gebruikt (setter)methodes van het model om er aanpassingen aan te brengen. (Het model kan trouwens op dezelfde manier ook rechtstreeks vanuit het programma worden gemanipuleerd.)
- (2) Telkens wanneer er gegevens in het model veranderen, signaleert het model dit aan al zijn luisteraars. (Soms gebeurt dit met behulp van gebeurtenissen, maar dit is niet noodzakelijk.)
- (3) Wanneer een view hoort dat het model is veranderd, vraagt hij aan het model de nieuwe gegevens op met behulp van (getter)methodes van het model en past zichzelf overeenkomstig aan.

Op deze manier heeft interactie met de gebruiker slechts onrechtstreeks een visueel effect: eerst wordt het model aangepast en dit heeft op zijn beurt als gevolg dat de view verandert, omdat de view een luisteraar is van het model.

Het is niet toegelaten dat een controller of een ander deel van het programma de view rechtstreeks aanpast. In feite weet de controller niet eens wat op een bepaald moment de views zijn van het model, en hoeft dit ook niet te weten.

Om dit allemaal te verduidelijken, zullen we dit MVC-patroon toepassen op het vierknop-



penvoorbeeld uit hoofdstuk 4. Hierbij gebruiken we de volgende klassen:

- Een klasse *PageModel* waarvan één object zal dienst doen als model. De enige informatie die we nodig hebben om alle knoppen en labels te kunnen afbeelden, is het huidige bladzijdenummer, een geheel getal. Dit is dan ook het enige gegeven dat door een object van het type *PageModel* wordt bijgehouden en beheerd.
- Een klasse *PageLabel* die zorgt voor een view van het model in de vorm van een label met daarop het overeenkomstige bladzijdenummer.
- De klasse *PageButton* waarvan we vier objecten zullen gebruiken. In de eerste plaats zijn dit controllers. Ze laten de gebruiker namelijk toe om de waarde van het model aan te passen. Tegelijkertijd doen ze ook dienst als minimale view: de waarde van het model bepaalt immers of de knoppen actief (*enabled*) zijn, of niet.

We gebruiken ook nog twee interfaces die voor dit doel in de JavaFX-bibliotheek zijn gedefinieerd:

- De interface *InvalidationListener* waaraan alle views moeten voldoen. Het model hoeft enkel te weten dat alle views van dit type zijn. Je zal de namen *PageLabel* en *PageButton* dus nergens terugvinden in de definitie van *PageModel*.
- De interface *Observable* waaraan het model moet voldoen. Deze interface geeft aan dat er *InvalidationListeners* bij het model kunnen worden geregistreerd.

## 5.2. Het model

De klasse *PageModel* is vrij eenvoudig van opzet. Het bladzijdenummer dat door het model wordt beheerd, bevindt zich in een private variabele *number* die met een *getter* kan worden opgevraagd.

```
public int getNumber () {  
    return number;  
}
```

Om het bladzijdenummer aan te passen, gebruik je een corresponderende *setter*.

Java voorziet nog andere interfaces en klassen die je als onderdeel van een MVC-patroon kunt gebruiken. Zo bevat het pakket *java.util* bijvoorbeeld de interface *Observer* en de klasse *Observable*.

Let op dat je deze laatste niet verwart met de interface *Observable* die we hier gebruiken.

```

public void setNumber (int number) {
    if (number != this.number) {
        this.number = number;
        fireInvalidationEvent();
    }
}

```

De methode *setNumber* is wellicht iets langer dan je had verwacht. Het MVC-patroon schrijft immers voor dat een model elke verandering moet signaleren aan al zijn luisteraars. Dat is de taak van de methode *fireInvalidationEvent* waarvan we later de implementatie zullen tonen.

Merk op dat luisteraars slechts worden ingelicht wanneer de waarde werkelijk verandert — en dit is geen onbelangrijk detail.

Voor het gemak hebben we ook nog een andere ‘setter’ geïmplementeerd. Hiermee kan je het bladzijdenummer verhogen (of verlagen).

```

public void incrementNumber (int increment) {
    if (increment != 0) {
        this.number += increment;
        fireInvalidationEvent();
    }
}

```

Merk opnieuw op dat *fireInvalidationEvent* slechts wordt opgeroepen wanneer het bladzijdenummer werkelijk verandert.

De rest van de klasse dient voor het beheer van de luisteraars van het model:

```

private final List<InvalidationListener> listenerList;

@Override
public void addListener(InvalidationListener listener) {
    listenerList .add(listener);
}

@Override
public void removeListener(InvalidationListener listener) {
    listenerList .remove(listener);
}

```

(De methode *removeListener* zullen we niet vaak nodig hebben. Ze moet er echter staan om aan het contract van de interface *Observable* te voldoen.)

De methode *fireInvalidationEvent* moet aan alle luisteraars die met *addListener* zijn geregistreerd, laten weten dat er iets aan het model is veranderd. Hij doet dit door de *listenerList* te overlopen, en voor elk element de methode *invalidated* op te roepen:

```
private void fireInvalidationEvent () {
    for (InvalidationListener listener : listenerList) {
        listener.invalidated(this);
    }
}
```

De interface *InvalidationListener* bestaat uit één methode (*invalidated*) die een *Observable* neemt als parameter (het model). Op die manier kan een bepaalde view zich eventueel bij meerdere modellen tegelijkertijd registreren en toch nog weten door welk model hij wordt gecontacteerd.

De code voor de methodes die de luisteraars beheren, is bij de meeste modellen dezelfde. Het is daarom een goed idee om deze code te bundelen in een klasse *Model* waarvan je dan alle andere modellen kan afleiden<sup>1</sup>.

## 5.3. Het bladzijdelabel

Ook de definitie van *PageLabel* is zeer kort.

```
public class PageLabel extends Label implements InvalidationListener {
    ...
    public void invalidated (Observable o) {
        setText ("Blz. " + model.getNumber ());
    }
}
```

De klasse is een extensie van de bibliotheekklasse *Label* die tegelijkertijd dient als view van het *PageModel*. Daarom moet *PageLabel* voldoen aan de interface *InvalidationListener* en bijgevolg de methode *invalidated* implementeren.

Volgens het MVC-patroon wordt de methode *invalidated* opgeroepen telkens als het model verandert, m.a.w., wanneer er een nieuw bladzijdenummer wordt geselecteerd (door de gebruiker of vanuit het programma). De view vraagt dan het nieuwe bladzijdenummer op aan het model met een gepaste getter (*getNumber*) en gebruikt die nieuwe waarde om zijn eigen grafische voorstelling aan te passen.

---

<sup>1</sup>In paragraaf §5.8 zal blijken dat je JavaFX-eigenschappen ook als model kunt gebruiken. Dit betekent dat je in de praktijk slechts zelden zelf een modelklasse zult moeten implementeren.

Opdat dit allemaal zou werken, is het nodig dat de view het model kan terugvinden waarmee het communiceert. In dit geval zouden we dit kunnen doen door de parameter *o* van *invalidated* te casten naar het type *PageModel*, maar de praktijk leert dat het beter is om het model (dat tijdens de levensduur van de view wellicht toch niet verandert) op te slaan als attribuut van het view-object. (De variabele *model* in dit geval.)

Vaak wordt het model al meegegeven (en opgeslagen) tijdens constructie van de view, zoals in onderstaande code:

```
public class PageLabel extends Label implements InvalidListener {  
  
    private final PageModel model;  
  
    public PageLabel (PageModel model) {  
        this.model = model;  
        model.addListener (this);  
    }  
    ...  
}
```

(Merk op dat de view wanneer hij wordt gecreëerd, zichzelf meteen als luisteraar bij het model registreert.)

In een JavaFX-omgeving is bovenstaand opzet niet altijd mogelijk en kan het model vaak pas worden doorgegeven nadat het view-object reeds is aangemaakt. Daarvoor is er dan een gepaste setter nodig<sup>2</sup>:

```
public class PageLabel extends Label implements InvalidListener {  
  
    private PageModel model;  
  
    public PageModel getModel() {  
        return model;  
    }  
  
    public void setModel (PageModel model) {  
        this.model = model;  
        model.addListener (this);  
    }  
    ...  
}
```

---

<sup>2</sup>Je kan er best een gewoonte van maken om dan ook meteen de gelijknamige getter in te voeren. FXML wil bijvoorbeeld niet met setters werken als de corresponderende getter niet bestaat.

En om volledig correct te zijn, moet je er ook rekening mee houden dat *setModel* meerdere keren kan worden opgeroepen, en dat het argument eventueel **null** zou kunnen zijn:

```
public void setModel (PageModel newModel) {  
    if (newModel != model) {  
        if (model != null) {  
            model.removeListener (this);  
        }  
        model = newModel;  
        if (model != null) {  
            model.addListener (this);  
        }  
    }  
}
```

## 5.4. De vier knoppen

De knoppen functioneren tegelijkertijd als controller en als view. Daarom implementeert de klasse *PageButton* zowel *InvalidationListener*, om als view op veranderingen in het model te kunnen reageren, en *EventHandler<ActionEvent>*, om interactie met de gebruiker mogelijk te maken.

```
public class PageButton extends Button  
    implements InvalidationListener, EventHandler<ActionEvent> {  
  
    public PageButton() {  
        setOnAction(this);  
    }  
  
    private PageModel model;  
    private int increment;  
    ... // setters en getters voor model en increment  
  
    public void invalidated(Observable o) {  
        int blz = model.getNumber();  
        setDisable(blz + increment < 1 || blz + increment > 100);  
    }  
  
    public void handle(ActionEvent t) {  
        model.incrementNumber(increment);  
    }  
}
```

Merk op dat *handle* niet zelf zorgt voor het al dan niet deactiveren van de knop. Dit gebeurt onrechtstreeks in *invalidated* wanneer de knop door het model van de verandering op de hoogte is gebracht.

## 5.5. De partnerklasse

Tot slot illustreert het volgende fragment hoe views en model in de partnerklasse worden geïnitialiseerd en met elkaar verbonden.

```
public void initialize() {  
    model = new PageModel();  
  
    pageLabel.setModel(model);  
    minus1.setModel(model);  
    minus1.setIncrement(-1);  
    ...  
    plus1.setModel(model);  
    plus1.setIncrement(10);  
    model.setNumber(1);  
}
```

We maken eerst het model aan en daarna de verschillende views. Bij elke view geven we een verwijzing door naar hetzelfde model. De laatste opdracht stelt een bladzijdenummer in bij het model, en dit zal meteen de nodige veranderingen teweeg brengen bij de verschillende views. We hoeven de views dus niet afzonderlijk te initialiseren.

Net zoals eerder (op blz. 94) kunnen we het *increment* opnemen als attribuut in het FXML-bestand. Er is zelfs meer: omdat er zowel een getter als een setter voor bestaan, is een *model*-attribuut ook toegestaan. Omdat we dit attribuut echter niet zomaar kunnen initialiseren met een geheel getal of een string, moeten we wat voorbereidend werk doen.

```
<fx:define>  
    <PageModel fx:id="model"/>  
</fx:define>  
<children>  
    <PageLabel model="$model" ... />  
    <PageButton model="$model" increment="-10" ... />  
    <PageButton model="$model" increment="-1" ... />  
    ...  
</children>
```

De eerste drie lijnen vertellen de *FXMLLoader* dat hij een nieuw object moet maken van het type *PageModel* en daaraan de naam *model* toewijzen. Bij de andere componenten kan je daarna dit model als attribuut gebruiken met de notatie *\$model*.

De partnerklasse is nu heel beknopt:

```
public class ButtonsCompanion {  
  
    public PageModel model;  
  
    public void initialize() {  
        model.setNumber(1);  
    }  
}
```

Honderd procent realistisch is dit voorbeeld nog niet. In de praktijk zal hetzelfde model wellicht ook nog in andere delen van de toepassing worden gebruikt. Je kan het model eventueel opvragen aan het partnerobject, maar waar vind je dit partnerobject terug?

De *FXMLLoader* kan ons een referentie naar dit object bezorgen nadat het FXML-bestand is ingeladen (hij heeft het object immers zelf aangemaakt). Die *FXMLLoader* wordt aangemaakt in de eerste lijn van de *start*-methode van het hoofdprogramma:

```
public void start(Stage stage) throws Exception {  
    FXMLLoader loader = new FXMLLoader(  
        ButtonsMain.class.getResource("Buttons.fxml")  
    );  
    Scene scene = new Scene(loader.load());  
    ...  
}
```

Aan dit *loader*-object kunnen we dan het partnerobject opvragen<sup>3</sup> (de ‘controller’), en uiteindelijk het model.

```
ButtonsCompanion companion = loader.getController();  
PageModel model = companion.getModel();
```

---

<sup>3</sup>Het omgekeerde gebeurt vaker in de praktijk: we maken zelf het partnerobject aan en bieden dat dan aan de *FXMLLoader* aan. We geven hiervan een voorbeeld in §5.6.

## 5.6. Views en controller samen in het partnerobject

Hoewel de manier van werken die we in vorige paragraaf hebben geschetst zeker onze voorkeur geniet, ook wanneer het om eenvoudige voorbeelden gaat zoals hier — ze blijven immers niet altijd eenvoudig — is het ook mogelijk een soort middenweg te bewandelen, waarbij model, view en controller minder extreem gescheiden zijn. De methode die we hieronder zullen schetsen, heeft bovendien als voordeel dat je het FXML-bestand niet met de hand hoeft te editeren.

De idee is om nog steeds een model te behouden, maar de view- en controllerfunctionaliteit te integreren in de partnerklasse: het zijn niet meer de knoppen en het label zelf die als view fungeren voor het model, maar wel het partnerobject.

Net als in onze eerste voorbeelden, wordt het indrukken van de knoppen opnieuw verwerkt met behulp van een methode, en niet met een afzonderlijk gebeurtenisverwerkend object. (Vier verschillende methodes, om exact te zijn.)

```
public class ButtonsCompanion implements InvalidationListener {

    public PageModel model;

    public Button minus1;
    ...
    public Button plus10;

    public Label blzLabel;

    public ButtonsCompanion(PageModel model) {
        this.model = model;
        model.addListener(this);
    }

    public void doMinus1() {
        model.incrementNumber(-1);
    }
    ...
    public void doPlus10() {
        model.incrementNumber(10);
    }

    public void initialize() {
        model.setNumber(1);
    }
}
```



```

@Override
public void invalidated(Observable observable) {
    int blz = model.getNumber();

    blzLabel.setText("Blz " + blz);
    minus1.setDisable(blz - 1 < 1);
    ...
    plus10.setDisable(blz + 10 > 100);
}
}

```

We hebben van de gelegenheid gebruik gemaakt om nog een andere techniek te illustreren: we laten het model dit keer niet aanmaken door de *FXMLLoader* en ook niet door de partnerklasse, maar maken het op voorhand aan en geven het mee als argument aan de constructor van de partnerklasse.

We kunnen (de naam van) de partnerklasse nu niet langer opnemen in het FXML-bestand, maar moeten het partnerobject nu expliciet instellen in het hoofdprogramma. Onderstaande code komt in de *start*-methode van de toepassing:

```

FXMLLoader loader = new FXMLLoader(
    ButtonsMain.class.getResource("Buttons.fxml")
);
PageModel model = new PageModel();
loader.setController(new ButtonsCompanion(model));
Scene scene = new Scene(loader.load());

```

## 5.7. JavaFX-eigenschappen

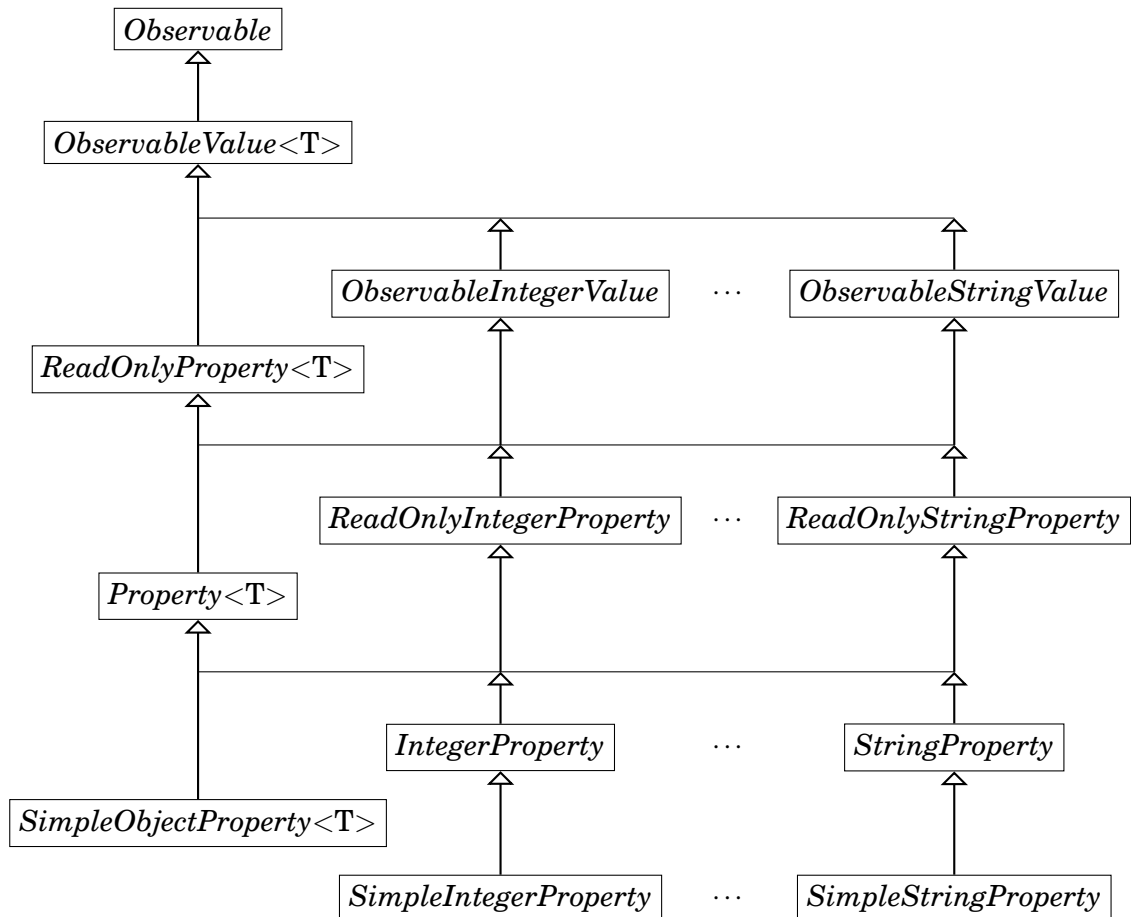
Het woord ‘eigenschap’ kan in de Java-wereld heel wat verschillende betekenissen hebben. We hebben in paragraaf §3.7 reeds de eigenschappen ontmoet die als sleutel-waardeparen zijn opgeslagen in *.properties*-bestanden. Daarnaast gebruikt men de term *eigenschap* ook voor een *getter-settercombinatie* — officieel heet dit een *booneigenschap* (Engels: *bean property*<sup>4</sup>).

JavaFX heeft het begrip nog verder uitgebreid door luisteraars te introduceren waarmee je op de hoogte wordt gebracht van veranderingen in de waarde van die eigenschap.

<sup>4</sup>De term *bean* verwijst naar ‘koffieboon’, omdat Java ook de naam is van een koffievariëteit. De term ‘bean property’ klinkt voor Engelstaligen dus even gek in de oren als ‘booneigenschap’ voor ons.

Dergelijke eigenschappen worden ook *gebonden* eigenschappen genoemd (Engels: *bound properties*). Als we het expliciet over *JavaFX* eigenschappen hebben, dan bedoelen we deze laatste.

Om goed te begrijpen wat *JavaFX*-eigenschappen zijn, bekijken we eerst onderstaande klassenhierarchie:



Dit schema is verre van volledig: er ontbreken bijvoorbeeld *WritableValue<T>*, *WritableIntegerValue*, ..., *WritableStringValue*, *IntegerExpression*, ..., *StringExpression*, *ReadOnlyJavaBeanProperty<T>*, ..., *ReadOnlyIntegerPropertyBase*, ..., *IntegerPropertyBase*, ..., en nog veel meer.

De interface *Observable* kennen we reeds: die geeft aan dat men een *InvalidationListener* kan registreren om op de hoogte gehouden te worden van wijzigingen in de waarde van het corresponderende object.

De interface *ObservableValue<T>* voegt daar nog enkele methodes aan toe:

```

public interface ObservableValue<T> extends Observable {
    T getValue();
    void addListener(ChangeListener<T> cl);
    void removeListener(ChangeListener<T> cl);
}

```

Een *ObservableValue* kan je beschouwen als een model dat slechts één waarde bijhoudt — een waarde van het type *T*. Je kan die waarde opvragen met *getValue*.

Naast de gebruikelijke *InvalidationListener* kan je nog een ander type luisteraar aan een *ObservableValue* hechten. En net zoals de *InvalidationListener* zal een dergelijke *ChangeListener* steeds op de hoogte gebracht worden van alle veranderingen in de waarde van het model. Alleen wordt er nu niet alleen gesignaleerd dát er iets is gewijzigd, maar ook meteen wát:

```

public interface ChangeListener<T> {
    void changed(ObservableValue<T> source, T oldVal, T newVal);
}

```

Dit maakt een variant op het MVC-patroon mogelijk waarbij een view niet langer het model hoeft te raadplegen om de nieuwe waarde op te halen (en dus ook geen referentie naar het model hoeft bij te houden) maar deze waarde rechtstreeks opgeleverd krijgt (als parameter van de methode *change*) telkens als het model verandert. Strikt gezien voldoet deze techniek niet aan onze definitie van MVC, maar voor eenvoudige modellen kan dit soms toch bruikbaar zijn.

Omdat generische types niet altijd even efficiënt en eenvoudig zijn, zeker niet wanneer men primitieve types als parameter wil gebruiken, heeft men voor deze primitieve types een ‘specialisatie’ van de interface *ObservableValue* voorzien: de interfaces *ObservableBooleanValue*, *ObservableIntValue*, *ObservableDoubleValue*, enz. (zoals aangegeven in het schema van bladzijde 122).

Elk van deze interfaces heeft naast de methode *getValue* ook nog een methode *get* die het primitieve type teruggeeft. Bijvoorbeeld:

```

public interface ObservableBooleanValue
    extends ObservableValue<Boolean> {
    boolean get();
}

```

Er bestaat ook nog een *ObservableStringValue* die min of meer een synoniem is van *ObservableValue*<*String*>, en een *ObservableNumberValue* die de interface *ObservableValue*<*Number*> uitbreidt met enkele methodes. En voor de volledigheid heeft men ook

nog een interface *ObservableObjectValue*<*T*> geïntroduceerd die aan de interface *ObservableValue*<*T*> een methode *get* toevoegt met dezelfde signatuur (en effect) als *getValue*. De JavaFX-documentatie adviseert om bij voorkeur deze gespecialiseerde interfaces te gebruiken in plaats van de generische interface.

De volgende interface die we bekijken is *ReadOnlyProperty*<*T*> en zijn specialisaties *ReadOnlyBooleanProperty*, ..., *ReadOnlyNumberProperty*, *ReadOnlyStringProperty* en *ReadOnlyObjectProperty*<*T*>. ‘Officieel’ is een JavaFX-eigenschap een object dat één van die interfaces implementeert. Voor onze doeleinden zijn *ReadOnlyProperty* en *ObservableValue* min of meer equivalent. (*ReadOnlyProperty* introduceert twee bijkomende methodes die wij echter nooit zullen gebruiken.)

De interface *Property* en zijn specialisaties bieden echter meer mogelijkheden. In de eerste plaats geeft *Property* ons een manier om de geobserveerde waarde ook te veranderen:

```
void setValue(T t);
```

(Voor specialisaties zoals *BooleanProperty* is er ook een methode *set* met een parameter van een primitief type — de tegenganger van *get*.) Waar *ObservableValue* (en dus ook *ReadOnlyProperty*) als model fungeert waaraan we enkel *views* kunnen koppelen, laat *Property* nu ook *controllers* toe<sup>5</sup>.

*Property* en *ReadOnlyProperty* zijn interfaces en hun specialisaties naar primitieve types zijn abstracte klassen. Wanneer we zelf JavaFX-eigenschappen willen aanmaken moeten we echter een concrete klasse kunnen gebruiken. Daarvoor dienen de klassen *SimpleIntegerProperty*, ..., *SimpleStringProperty* en *SimpleObjectProperty*<*T*>. Deze klassen bieden kant-en-klare implementaties van de overeenkomstige interfaces.

De GUI-componenten van de JavaFX-bibliotheek bezitten elk een ganse resem JavaFX-eigenschappen. Hierbij volgt JavaFX een bijzondere naamgevingsafspraken.

Heeft een klasse bijvoorbeeld een *item*-eigenschap, dan kan je een methode *itemProperty()* verwachten (zónder *get*) die de eigenschap zelf teruggeeft, een methode *getItem()* die de *waarde* van de eigenschap teruggeeft, en een methode *setItem(..)* die de waarde aanpast. De oproep *getItem()* is een afkorting van *itemProperty().get()* en *setItem(..)* is een afkorting van *itemProperty().set(..)*. Je hebt dit wellicht al opgemerkt in de elektronische documentatie.

---

<sup>5</sup>De naam *ReadOnlyProperty* is enigszins misleidend. Meestal kan een object dat aan de interface *ReadOnlyProperty* voldoet, wél van waarde veranderen — het zou anders niet veel zin hebben om er een luisteraar bij te registreren. Alleen gebeurt dat niet via deze interface maar met setters of methodes die eigen zijn aan de specifieke klasse waartoe het object behoort.

## 5.8. Een JavaFX-eigenschap als model

Wanneer het model van een MVC-toepassing zeer eenvoudig is, hoeven we dit vaak niet afzonderlijk te implementeren en kunnen we in de plaats als model een standaard-JavaFX-eigenschap nemen.

In het *PageModel* van het vierknoppenvoorbeeld houden we enkel een bladzijdenummer bij, t.t.z., een geheel getal. We kunnen hiervoor dan evengoed een *SimpleIntegerProperty* gebruiken. De partnerklasse van het voorbeeld uit paragraaf §5.6 krijgt daardoor de volgende implementatie:

```
public class ButtonsCompanion implements InvalidationListener {

    private final IntegerProperty pageNumber;

    public Button minus1;
    ...
    public Button plus10;

    public Label blzLabel;

    public ButtonsCompanion(IntegerProperty pageNumber) {
        this.pageNumber = pageNumber;
        pageNumber.addListener(this);
    }

    public void doMinus1() {
        pageNumber.set(pageNumber.get()-1);
    }
    ...

    public void doPlus10() {
        pageNumber.set(pageNumber.get()+10);
    }

    public void initialize() {
        pageNumber.set(1);
    }
}
```

```

@Override
public void invalidated(Observable observable) {
    int blz = pageNumber.get();

    blzLabel.setText("Blz " + blz);
    minus1.setDisable(blz - 1 < 1);
    ...
    plus10.setDisable(blz + 10 > 100);
}
}

```

en we geven het partnerklasseobject op de volgende manier aan de FXML-loader door:

```

IntegerProperty pageNumber = new SimpleIntegerProperty();
loader.setController(new ButtonsCompanion(pageNumber));

```

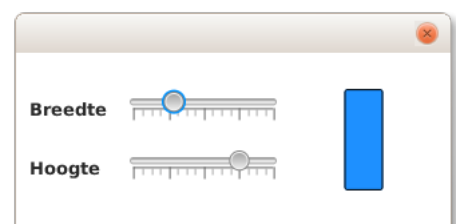
Er is weinig verschil met de vorige implementatie: waar er vroeger een *PageModel* stond, staat nu een *IntegerProperty*. De oproep van *getNumber* is vervangen door *get*, *setNumber* door *set* en er is niet langer een methode *incrementNumber* zodat we het verhogen en verlagen van het paginanummer nu voluit moeten noteren. De partnerklasse wordt nog steeds als luisteraar geregistreerd bij het model.

## 5.9. Eigenschappen verbinden

[ Voorbeelden uit deze paragraaf vind je o.a. in *be.ugent.objprog.bindings*. ]

Eén van de paradepaardjes van JavaFX is de mogelijkheid om eigenschappen met elkaar te *verbinden*.

In de toepassing hiernaast kan je de hoogte en breedte van de rechthoek rechtstreeks beïnvloeden met twee schuivers. Elk van die schuivers stelt een waarde voor tussen 0 en 100, een waarde die rechtstreeks correspondeert met de afmetingen (in pixels) van de rechthoek. De afgebeelde rechthoek is dus 28 pixels breed en 75 pixels hoog.



De waarde van een schuiver is een JavaFX-eigenschap met naam *value*. De hoogte en breedte van een rechthoek zijn eigenschappen met namen *height* en *width*. Om het geschetste effect te verwezenlijken, hebben we twee luisteraars nodig: de eerste luistert naar de *value*-eigenschap van de eerste schuiver en past de *width*-eigenschap van de

rechthoek aan de nieuwe waarde aan telkens wanneer die verandert, de tweede doet hetzelfde met de andere schuiver en de *height*-eigenschap.

Hoewel dit op zich niet zoveel werk vraagt, biedt JavaFX toch nog een kortere manier om dit te verwezenlijken. De volgende twee lijnen volstaan:

```
rectangle.heightProperty().bind (height.valueProperty());  
rectangle.widthProperty().bind (width.valueProperty());
```

De interface *Property*<T> bevat onder andere de volgende methode:

```
void bind(ObservableValue<T> observable);
```

Met deze methode kan je een eigenschap *verbinden* (Engels: *to bind*) met een *ObservableValue*. Het effect hiervan is dat de waarde van de eigenschap automatisch gesynchroniseerd wordt met die van de geobserveerde waarde. (Achter de schermen wordt er nog steeds een luisteraar gebruikt zoals hierboven geschetst.)

Een eigenschap kan op die manier slechts met één *ObservableValue* tegelijkertijd worden verbonden en de verbinding geldt slechts in één richting. De *ObservableValue* verandert dus niet automatisch van waarde wanneer de eigenschap dit doet (er is in de interface *ObservableValue* ook geen methode voorzien waarmee dit zou kunnen gebeuren). Zolang een eigenschap is gebonden, is het trouwens niet toegelaten zijn waarde rechtstreeks te veranderen (en veroorzaakt dit een uitzondering).

In de plaats van een dergelijke enkelvoudige verbinding tussen een eigenschap en een *ObservableValue*, kan je ook een *bidirectionele* verbinding opzetten tussen twee eigenschappen, met de volgende methode:

```
public void bindBidirectional(Property<T> other)
```

In het programma dat hiernaast is afgebeeld, hebben we zo'n verbinding gebruikt om ervoor te zorgen dat het tekstveld altijd exact dezelfde waarde toont als de schuiver, en omgekeerd, dat de schuiver naar de corresponderende waarde verspringt zodra de gebruiker iets in het tekstveld intikt (zelfs *tijdens* het tikken).



Dit doen we door de *value*-eigenschap van de schuiver bidirectioneel te verbinden met de *text*-eigenschap van het tekstveld.

```
field.textProperty().bindBidirectional( slider.valueProperty() );
```

Zoals de code hier is afgedrukt, zal ze echter niet werken (en zelfs niet compileren). Er is namelijk een type-incompatibiliteit — *textProperty()* retourneert een *StringProperty* en *valueProperty()* een *DoubleProperty*.

Gelukkig voorziet *StringProperty* (maar niet *Property* zelf!) nog een tweede versie van *bindBidirectional*:

```
public void bindBidirectional  
    (Property<T> other, StringConverter<T> converter)
```

Je kan een stringeigenschap bidirectioneel verbinden met een eigenschap met een basistype *T* dat verschillend is van *String* door aan te geven hoe een object van het type *T* moet worden omgezet naar een *String*, en omgekeerd. Dit doe je door een *StringConverter* mee te geven als bijkomende parameter.

*StringConverter* is een abstracte klasse met de volgende methodes:

```
public String toString(T t);  
  
public T fromString(String string);
```

Er bestaan reeds heel wat standaardstringconverters die je hiervoor kan gebruiken. In ons voorbeeld gebruiken we een *NumberStringConverter*:

```
field.textProperty().bindBidirectional(  
    slider.valueProperty(),  
    new NumberStringConverter()  
);
```

Het kan nog spectaculairder: zo is het bijvoorbeeld mogelijk om een eigenschap te verbinden met de *som* of het *product* van twee *ObservableValues*, of een Booleaanse eigenschap te synchroniseren met het feit of twee *ObservableValues* al dan niet gelijk zijn aan elkaar.

We voegen aan het vorige voorbeeld een (*read only*) checkbox toe die op elk moment moet tonen of de rechthoek een vierkant is of niet. Dit vraagt een luisteraar die tegelijk naar beide *value*-eigenschappen luistert en indien nodig de *checkbox* (de)selecteert.

In JavaFX kan je dit noteren op de volgende manier:

```
checkbox.selectedProperty().bind (  
    Bindings.equal(height.valueProperty(), width.valueProperty())  
);
```



Of een checkbox al dan niet geselecteerd is, wordt aangegeven door zijn *selected*-eigenschap. De klassenmethode *Bindings.equal(..)* retourneert een 'model' dat **true** is of **false** al naargelang de waarden van zijn twee parametereigenschappen gelijk zijn of niet. In de elektronische documentatie van *Bindings* vind je nog meer dergelijke objecten die twee of meer JavaFX-eigenschappen op één of andere manier met elkaar combineren.

Hoe interessant dit verbinden van eigenschappen ook mag ogen, in de professionele praktijk heeft het echter niet zoveel directe toepassingen.



## 6. Databanken aanspreken met JDBC

Wanneer je, zoals in de meeste professionele toepassingen, gegevens wil bewaren op schijf om die later weer in te lezen en opnieuw te veranderen, dan is de gewone bestandsinvoer/uitvoer (zoals in hoofdstuk 3) hiervoor niet altijd de meest geschikte technologie. Je gebruikt dan best een zogenaamde *relationele databank*.

Relationele databanken bestuur je met een programmeertaal die *SQL* heet en Java biedt je de mogelijkheid om dergelijke SQL-opdrachten vanuit je programma te lanceren met behulp van de *JDBC-API (Java Database Connectivity)*<sup>1</sup>.

JDBC bestaat ondertussen reeds in heel wat verschillende versies, waarbij elke versie meer mogelijkheden biedt dan de vorige. In deze tekst beperken we ons voornamelijk tot (een gedeelte van) de basisfunctionaliteiten uit versie 1.0. In de praktijk kan je hier al een heel eind mee verder.

Merk op dat JDBC je alleen maar een manier biedt om gemakkelijk vanuit Java met een bestaande databank te communiceren. Voor de databanksoftware zelf moet je afzonderlijk zorgen. Je hebt hiervoor verschillende opties:

- Bij grotere projecten maak je gebruik van een bestaand databanksysteem dat reeds op één of andere databankserver draait. Jouw programma is dan wellicht niet het enige dat deze databank gebruikt en misschien zijn de handelingen die je erop kunt uitvoeren wel beperkt ('alleen lezen', bijvoorbeeld). Je hebt een gebruikersidentificatie en wachtwoord nodig om toegang te krijgen en communicatie gebeurt over het netwerk.

(De server hoeft niet noodzakelijk een afzonderlijk toestel te zijn, maar kan ook hetzelfde toestel zijn waarop het Java-programma draait.)

Dergelijke databanksoftware hoeft niet duur te zijn. Er bestaan heel wat *open source* producten die heel degelijk zijn, waaronder *MySQL* en *PostgreSQL* twee van de meest bekende zijn. Voor onze voorbeelden gebruiken we *Apache Derby*, een databankserver die heel gemakkelijk te installeren valt.

---

<sup>1</sup>Appendix A bevat een hele korte inleiding tot SQL voor de lezer die er niet mee vertrouwd is.

JDBC is niet de enige techniek die wordt gebruikt om databanken aan te spreken vanuit Java. Een heel populair alternatief is bijvoorbeeld JPA (*Java Persistence Architecture*), een raamwerk dat databankbewerkingen zoveel mogelijk verbergt door ze te vermommen als gewone Java-bewerkingen. (Een Java-object kan bijvoorbeeld zodanig worden geconfigureerd dat een eenvoudige setter-oproep automatisch tot gevolg heeft dat ook een overeenkomstig databankelement wordt geüpdatet.)

Een woord van waarschuwing is hier op zijn plaats: JPA is een heel krachtig instrument dat echter veel ingewikkelder is dan wat uit een eerste oppervlakkige kennismaking mag blijken. Je moet heel goed weten wat er achter de schermen gebeurt om het foutloos te kunnen gebruiken. Misschien is dat de reden waarom het de laatste jaren een beetje aan populariteit moet inboeten?

- Bij kleinere projecten gebruikt men vaak een zogenaamde *ingebedde* databankserver. In dat geval heeft alleen het programma zelf toegang tot de databank. Identificatie en wachtwoorden zijn meestal niet nodig. *SQLite* is een populair voorbeeld van een dergelijk systeem, maar ook *Apache Derby* biedt de optie om ingebed te werken.
- Tot slot bestaan er ook zogenaamde *in memory* databanken. Deze systemen bewaren hun gegevens niet op harde schijf maar in het interne geheugen van de computer. Met andere woorden, het programma start met een lege databank, en wanneer het stopt zijn alle gegevens die er werden in opgeslagen weer verloren. Deze databanken worden vaak gebruikt om software te testen terwijl ze wordt ontwikkeld.

## 6.1. Drivers en verbindingen

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.zip*. ]

Om in de praktijk met JDBC te werken, heb je steeds een zogenaamde *driver* nodig. Dit is een Java-klasse die specifiek is voor het databanktype dat je bij je toepassing wenst aan te spreken. Een dergelijke driver wordt meestal ter beschikking gesteld door de producent van de databank.

Drivers komen in de vorm van JAR-archieven. Om ze te gebruiken, moet het overeenkomstige archief in het *class path* staan<sup>2</sup>. (De driver-JAR voor *Apache Derby* komt in twee versies: *derby.jar* voor de ingebedde en *derbyclient.jar* voor de alleenstaande versie.)

Merk op dat het perfect mogelijk is om op hetzelfde moment verschillende drivers actief te hebben. Je toepassing kan op die manier bijvoorbeeld tegelijkertijd gebruik maken van verschillende databanken van verschillende leveranciers.

<sup>2</sup>In eerdere versies van Java moest de driver ook nog expliciet worden *ingeladen*, en dit vraagt om bijkomende code die er nogal ontoegankelijk uitziet. Heel wat auteurs op het Internet hebben nog niet door dat dit niet langer nodig is.

Eenmaal de gepaste JDBC-driver is ingeladen, kan je een verbinding met een databank opzetten. Een dergelijke verbinding wordt voorgesteld door een object dat tot de interface *Connection* behoort. De eigenlijke implementatie van een verbinding is afhankelijk van de gebruikte databank (t.t.z., van de gebruikte driver).

Om de verbinding aan te maken, gebruik je de methode *getConnection* van de klasse *DriverManager*. Deze methode heeft drie parameters: een JDBC-URL, een gebruikersnaam en een wachtwoord. Er bestaat ook een versie van *getConnection* met slechts één parameter, voor databanken die geen wachtwoord vereisen (bijvoorbeeld bij ingebedde databankservers).

De JDBC-URL bestaat uit drie stukken: het protocol, het subprotocol en de 'subname'. Het protocol is 'jdbc:' en het subprotocol identificeert de databankdriver en wordt bepaald door de databankproducent. Gebruik je Apache Derby, dan is het subprotocol 'derby:'. Bij SQLite is dit 'sqlite:'.

Het laatste gedeelte van de URL (de subname) beschrijft de gebruikte databank op een manier die verschilt van driver tot driver. Bij Apache Derby (en bij vele andere client/server-databanken) is dit laatste stuk meestal van de vorm `//server/naamdatabank`. Hierbij is `naamdatabank` de naam van de databank en `server` de naam van de server waarop de databank draait:

```
private final static String JDBC_URL =  
    "jdbc:derby://localhost/objprog";
```

Bij SQLite is de subname de naam van een bestand (SQLite slaat de volledige databank immers op in één enkel bestand).

```
private final static String JDBC_URL =  
    "jdbc:sqlite:/public/db/objprog.sqlite";
```

SQLite biedt ook de mogelijkheid om het databankbestand in het *class path* op te nemen (wat alleen zin heeft als we enkel uit de databank zullen lezen en er niet in schrijven). Dit noteer je dan als volgt:

```
private final static String JDBC_URL =  
    "jdbc:sqlite::resource:be/ugent/objprog/zip/zipcodes.sqlite";
```

(In dit geval heet het databankbestand dus *zipcodes.sqlite* en bevindt het zich in het pakket *be.ugent.objprog.zip*.)

Het volgende codefragment vat dit allemaal samen:

```

private final static String USER_ID = "objprog";
private final static String PASSWD = "sesamopenu";

try (Connection conn =
    DriverManager.getConnection (JDBC_URL, USER_ID, PASSWD)) {
    ... // gegevens ophalen, toevoegen, veranderen, ...
} catch (SQLException e) {
    ... // databankproblemen
}

```

We gebruiken hier een try-met-bronnen (zie §3.3) om er zeker van te zijn dat de databank-verbinding wordt gesloten als we ermee klaar zijn, ook wanneer er een fout is gebeurd. De meeste methodes uit JDBC kunnen uitzonderingen opgooien van het type *SQLException* — een *gecontroleerde* uitzondering.

Databankverbindingen zijn ‘duur’ en vaak laat een databank slechts enkele connecties toe met dezelfde toepassing. Het is dus zeer belangrijk dat elke verbinding die open gaat ook telkens weer wordt afgesloten.

## 6.2. Opdrachten

Eenmaal de verbinding is gemaakt, kan je de databank vanuit een Java-programma een aantal SQL-opdrachten laten uitvoeren. Een dergelijke opdracht wordt voorgesteld als een object van het type *PreparedStatement*. *PreparedStatement* is een interface die SQL-opdrachten naar de databank stuurt en de resultaten ophaalt<sup>3</sup>. Een *PreparedStatement*-object wordt op de volgende wijze aangemaakt en (automatisch) terug afgesloten:

```

try (PreparedStatement stmt = conn.prepareStatement (
    "... SQL-opdracht ..."
)) {
    ... // opdrachten uitvoeren
} catch (SQLException e) {
    ...
}

```

Het object *conn* is van het type *Connection* en stelt een verbinding met de databank voor.

---

<sup>3</sup>Er bestaat ook een interface *Statement* die op een lichtjes andere manier wordt gebruikt. Wij kiezen er echter voor om steeds *PreparedStatement* te gebruiken, omdat *Statement* slechts in heel weinig gevallen (een klein) voordeel biedt.

We maken onderscheid tussen SQL-opdrachten die rijen of records van de databank als resultaat hebben (zoekopdrachten) en andere opdrachten. Deze laatste soort gebruik je om tabellen (of gebruikers, ed.) aan te maken<sup>4</sup> of om rijen aan een tabel toe te voegen, te wijzigen of te verwijderen. In deze gevallen roep je de methode *executeUpdate* van *PreparedStatement* op om de opdracht uit te voeren.

**int executeUpdate () throws SQLException;**

De SQL-opdracht die zal worden uitgevoerd is de string die werd opgegeven bij het aanmaken van de *PreparedStatement*. De retourwaarde van *executeUpdate* is het aantal aangepaste rijen.

In de volgende fragmenten gebruiken we een databanktabel *town* die bestaat uit twee kolommen: een kolom *zip* met het postnummer van een stad of gemeente, en een kolom *name* met de bijbehorende naam. Beide kolommen bevatten strings.

Onderstaand programmafragment voegt een nieuw record toe aan de databank, voor een gemeente met naam 'Rivendel' en postnummer 9999.

```
try (Connection conn = DriverManager.getConnection (JDBC_URL);
    PreparedStatement stmt = conn.prepareStatement(
        "INSERT INTO town(zip,name) VALUES ('9999' , 'Rivendel' )"
    )) {
    stmt.executeUpdate();
} catch (SQLException e) {
    System.err.println ("Database error: " + e);
}
```

Merk op dat we in deze try-met-bronnen *twee* bronnen tegelijk initialiseren — de verbinding en de opdracht.

Voor zoekopdrachten (SELECT-opdrachten) die een aantal rijen uit de databank als resultaat hebben, gebruik je niet de methode *executeUpdate* maar de methode *executeQuery*:

**ResultSet executeQuery (String sqlOpdracht) throws SQLException;**

Het resultaat van deze methode is een object van het type *ResultSet*. Je kan de inhoud van dit object (de waarden van de verschillende kolommen) opvragen met behulp van gepaste methodes.

---

<sup>4</sup>Voor het aanmaken van tabellen en gebruikers of andere opdrachten die maar één keer worden uitgevoerd, schrijf je in de regel geen afzonderlijk Java-programma maar gebruik je beter een SQL-script.

Elke *ResultSet*-object bevat een wijzer (ook *cursor* genoemd). Deze wijzer staat na het aanmaken van het object vóór de eerste rij. Met deze wijzer wordt het *ResultSet*-object rij na rij overlopen. Je kan enkel gegevens opvragen uit de rij waarnaar de cursor op dat moment wijst. De interface *ResultSet* biedt hiervoor verschillende methodes:

```
boolean getBoolean (String columnName);  
double getDouble (String columnName);  
int     getInt      (String columnName);  
String getString (String columnName);  
Date    getDate     (String columnName);  
Time    getTime     (String columnName);  
...
```

De methode-oproep *getXxx(columnName)* geeft de waarde van de kolom met naam *columnName* terug als een element van het type *xxx*. Let wel op dat het type dat je in het Java-programma gebruikt, overeenkomt met het type waarmee de overeenkomstige kolom in de databank is gedefinieerd<sup>5</sup>.

Er is ook een versie van *getXxx(..)* die het *volgnummer* van de kolom als argument neemt (geteld vanaf 1). Omdat de volgorde van de kolommen in een databanktabel echter in principe niet vastligt, is dit een zeer gevaarlijk alternatief, behalve misschien wanneer het resultaat slechts één kolom heeft<sup>6</sup>.

De implementatie van *ResultSet* kan verschillen van driver tot driver: in sommige gevallen wordt de inhoud van deze set meteen ingevuld wanneer *executeQuery* wordt opgeroepen, in andere gevallen wordt de databank slechts gecontacteerd op het moment dat je de gegevens met *getXxx*-methodes opvraagt. Dit verklaart waarom elk van deze methodes een *SQLException* kan veroorzaken, en waarom het belangrijk is om ook een *ResultSet* telkens mooi af te sluiten met *close*, door ook hier een try-met-bronnen te gebruiken.

Om de cursor één rij naar beneden te schuiven, gebruik je de methode *next* (zonder parameters). Deze functie geeft **true** terug als er nog rijen zijn en **false** in het andere geval. Ze wordt vaak gebruikt in de conditie van een **while**-lus van de volgende vorm:

```
try (ResultSet res = stmt.executeQuery ()) {  
    while (res.next ()) {  
        // haal de gegevens op van de huidige rij  
    }  
}
```

---

<sup>5</sup>De types *Date* en *Time* behoren tot het pakket *java.sql* en mogen dus niet verward worden met het standaardtype *Date* uit *java.util*. Er bestaan methodes om objecten van deze types om te zetten van en naar *LocalDate* en *LocalTime*.

<sup>6</sup>Wij gebruiken steeds de versie met de naam van de kolom, behalve in één specifiek geval dat we behandelen in paragraaf §6.4.



(De try-met-bronnen zorgt ervoor dat de *ResultSet* telkens wordt afgesloten.)

Hieronder gebruiken we deze techniek om de namen af te drukken van alle gemeenten met een opgegeven postnummer:

```
public void findTowns(Connection conn, String zip) throws SQLException {  
    try (PreparedStatement stmt = conn.prepareStatement(  
        "SELECT name FROM town WHERE zip=' " + zip + "' ")  
    );  
    ResultSet res = stmt.executeQuery()  
    {  
        while (res.next()) {  
            System.out.println(res.getString("name"));  
        }  
    }  
}
```

We nemen aan dat *conn* hier een verbinding met de databank voorstelt die in een ander gedeelte van het programma wordt geopend en gesloten.

Merk op dat de SQL-opdrachtstring in dit geval geen vaste waarde heeft, maar *at runtime* geconstrueerd wordt uit de parameter *zip*. Er bestaat hiervoor een handig alternatief. We kunnen *findTowns* op de volgende manier herschrijven:

```
public void findTowns(Connection conn, String zip) throws SQLException {  
    try (PreparedStatement stmt = conn.prepareStatement(  
        "SELECT name FROM town WHERE zip = ?"  
    )) {  
        stmt.setString(1, zip);  
        try (ResultSet res = stmt.executeQuery()) {  
            while (res.next()) {  
                System.out.println(res.getString("name"));  
            }  
        }  
    }  
}
```

De vraagtekens in de SQL-opdracht stellen parameters voor die later een waarde krijgen (in ons voorbeeld is er maar één vraagteken, maar meerdere parameters zijn toegelaten). Deze vraagtekens worden dan ‘ingevuld’ met methodes van de volgende vorm:

```
public void setXxx (int indexParameter, xxx waardeParameter)
```

Het eerste argument bepaalt het volgnummer van de parameter (het vraagteken) die de meegegeven waarde *waardeParameter* moet krijgen (parameters worden genummerd vanaf 1). Alle parameters moeten op deze manier een waarde krijgen vooraleer de SQL-opdracht uitgevoerd kan worden. De methodes *executeUpdate* en *executeQuery* voeren dan uiteindelijk de opdracht uit.

Er is nog een belangrijke reden om vraagtekens in een *PreparedStatement* te gebruiken en niet zelf de SQL-opdrachtstring samen te stellen. Veronderstel bijvoorbeeld dat we onze lijst met postnummers nu op de omgekeerde manier willen doorzoeken: gegeven de naam van een gemeente, wat is dan haar postnummer? We zouden dit kunnen doen op de volgende manier:

```
PreparedStatement stmt = conn.prepareStatement (
    "SELECT zip FROM town WHERE name=' " + naam + "' "
);
ResultSet rs    = stmt.executeQuery ();
...
```

Dit zal echter niet altijd werken, bijvoorbeeld wanneer de naam van een gemeente een aanhalingsteken bevat (Braine-l'Alleud, Sint-Job-in-'t-Goor, ...). Een aanhalingsteken moet je in een SQL-string namelijk op een speciale manier noteren. We zouden dit zelf kunnen doen door in *naam* op voorhand alle aanhalingstekens aan te passen, maar dit wordt niet vereenvoudigd door het feit dat dit niet bij elk type databank op dezelfde manier moet gebeuren.

De volgende Java-code, alhoewel één lijn langer, vermijdt al deze problemen<sup>7</sup>.

```
PreparedStatement stmt = conn.prepareStatement (
    "SELECT zip FROM town WHERE name=?"
);
stmt.setString(1, naam);
ResultSet rs    = stmt.executeQuery ();
...
```

(Merk trouwens op dat er geen aanhalingstekens staan rond het vraagteken.)

Een *PreparedStatement* kan (over dezelfde verbinding) meer dan één keer gebruikt worden, telkens met verschillende parameters. Je mag voor hetzelfde *PreparedStatement*-object meerdere keren *executeUpdate* of *executeQuery* oproepen nadat je telkens de vraagtekens met nieuwe waarden hebt ingevuld.

In onderstaand fragment drukken we alle namen van gemeenten af waarvan de postnummers zijn opgeslagen in de tabel *args*:

<sup>7</sup>Zelf SQL-opdrachtlijnen samenstellen is ook onveilig en kan een weg openen naar *SQL-injectie*.

```

try (Connection conn = DriverManager.getConnection(JDBC_URL);
      PreparedStatement stmt = conn.prepareStatement(
          "SELECT name FROM town WHERE zip=?"
      )) {
    for (String zip : args) {
        stmt.setString(1, zip);
        try (ResultSet res = stmt.executeQuery()) {
            while (res.next()) {
                System.out.printf("%s: %s\n", zip, res.getString("name"));
            }
        }
    }
} catch (SQLException e) {
    System.err.println("Database error: " + e);
}

```

## 6.3. Batch-bewerkingen

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.proglangs.* ]

In de toepassing die hiernaast is afgebeeld, kan de gebruiker zijn voorkeuren voor een aantal programmeertalen aan- of uitvinken.

We slaan deze voorkeuren op in een databanktabel *prefs* met twee kolommen: *name* bevat de naam van de programmeertaal, *checked* bevat een Booleaanse waarde die aangeeft of dit een voorkeursprogrammeertaal is, of niet.

De informatie die we uiteindelijk met behulp van JDBC in de databank willen opslaan, is opgeslagen in een lijst *list* van records van de klasse *Preference* (met velden *name* and *checked*.)

Het ligt voor de hand om dit te doen door één prepared statement herhaaldelijk uit te voeren, maar telkens met andere argumenten.

C	<input type="checkbox"/>
C#	<input type="checkbox"/>
C++	<input checked="" type="checkbox"/>
Java	<input checked="" type="checkbox"/>
Javascript	<input type="checkbox"/>
PHP	<input checked="" type="checkbox"/>
Python 2	<input type="checkbox"/>
Python 3	<input type="checkbox"/>
Ruby	<input type="checkbox"/>
Scala	<input type="checkbox"/>
VB.Net	<input type="checkbox"/>

```

try (PreparedStatement stat = conn.prepareStatement(
      "UPDATE prefs SET checked = ? WHERE name = ?"
  )) {

```

```

    for (Preference pref : list) {
        stat.setBoolean(1, pref.checked());
        stat.setString(2, pref.name());
        stat.executeUpdate();
    }
} catch (SQLException ex) {
    ...
}

```

Voor dit soort toepassingen biedt JDBC een alternatieve techniek — *batch update* genaamd. In de plaats van elke update-opdracht afzonderlijk door te geven aan de databankserver worden ze eerst verzameld (met *addBatch*) en dan in één enkele lading doorgestuurd (met *executeBatch*). Vaak is dit een stuk sneller.

```

try (PreparedStatement stat = conn.prepareStatement(
    "UPDATE prefs SET checked = ? WHERE name = ?"
)) {
    for (Preference pref : prefs) {
        stat.setBoolean(1, pref.checked());
        stat.setString(2, pref.name());
        stat.addBatch();
    }
    stat.executeBatch();
} catch (SQLException ex) {
    ...
}

```

Er bestaat ook een *batch insert* waarmee je meerdere rijen aan een tabel kunt toevoegen in één keer. Stel bijvoorbeeld dat we de voorkeuren niet als logische waarden opslaan, maar in de plaats een afzonderlijke tabel *chosen* bijhouden waarin enkel de voorkeurs-programmeertalen zitten. Die tabel vul je dan op de volgende manier op.

```

try (PreparedStatement stat = conn.prepareStatement(
    "INSERT INTO chosen(name) VALUES (?)"
)) {
    for (Preference pref : list) {
        if (pref.checked()) {
            stat.setString(1, pref.name());
            stat.addBatch();
        }
    }
    stat.executeBatch();
}

```

Je moet de tabel *chosen* wel vooraf volledig leegmaken met een afzonderlijke SQL-opdracht. Deze opdracht kan niet in dezelfde ‘batch’ worden opgenomen, omdat we er een aparte prepared statement voor nodig hebben.

## 6.4. Automatisch gegenereerde sleutels

Vaak is het zo dat een databanktabel een unieke sleutel bezit die bij het aanmaken van een nieuwe rij niet expliciet moet worden opgegeven maar automatisch door de databank wordt gegenereerd. We bekijken als voorbeeld hiervan een tabel *personen* met kolommen *id*, *familienaam* en *voornaam* (zie ook appendix A).

Omdat twee personen toevallig dezelfde familienaam en voornaam kunnen hebben, kan je de naam van een persoon niet als primaire sleutel gebruiken en zijn we daarom verplicht om een bijkomende kolom *id* te introduceren. Welke waarde deze sleutel heeft voor een specifieke persoon, is van geen enkel belang, zolang deze maar verschillend is van persoon tot persoon.

Wanneer we nieuwe personen toevoegen aan de databank, kunnen we enkel hun familienaam en voornaam opgeven, en meer heeft de databank trouwens ook niet nodig om een nieuwe rij in de tabel aan te maken — de SQL-opdracht

```
INSERT INTO personen(familienaam,voornaam) VALUES ('Lejeune','Louis')
```

is perfect toegelaten, ook al geven we geen *id*-waarde op. Het is echter vaak nuttig om in het programma te weten te komen wat de nieuwe sleutelwaarde is die automatisch werd gegenereerd. We zouden de sleutel eventueel op de volgende manier kunnen opvragen.

```
SELECT id FROM personen WHERE familienaam='Lejeune' AND voornaam='Louis'
```

Maar wat als er twee Louis Lejeunes zijn?

Hiervoor bestaat een elegante oplossing. De methode *prepareStatement* aanvaardt als bijkomende parameter een constante *Statement.RETURN\_GENERATED\_KEYS* om aan te geven dat we de sleutel later willen opvragen

```
stat = conn.prepareStatement(  
    "INSERT INTO personen(voornaam,familienaam) VALUES (?,?)",  
    Statement.RETURN_GENERATED_KEYS  
)  
...
```

Nadat je dit prepared statement hebben uitgevoerd, kan je met *getGeneratedKeys()* de gegenereerde waarden opvragen. Het resultaat van deze methode is een *ResultSet*.

```
stat.setString("voornaam", voornaam);
stat.setString("familienaam", familienaam);
stat.executeUpdate();
try (ResultSet keys = stat.getGeneratedKeys()) {
    if (keys.next()) {
        nieuweId = keys.getInt(1);
    } else {
        throw new Exception(...); // zou niet mogen gebeuren
    }
}
```

Merk op dat we hier een *getInt* gebruiken die het getal 1 als argument neemt en niet de naam van een kolom zoals het eigenlijk hoort. Afhankelijk van de databanksoftware blijkt die kolomnaam immers niet altijd op een betrouwbare manier te voorspellen.

Een uitzondering hierop is PostgreSQL. In dat geval geeft de *result set* alle kolommen terug van de nieuwe rij en kan je beter de kolomnaam van de primaire sleutel te gebruiken, ook al staat die kolom wellicht vooraan.

## 7. Draden in JavaFX

Java is één van de weinige programmeertalen waarin je op een eenvoudige manier twee verschillende delen van je programma *tegelijkertijd* kan laten draaien. Hieronder overlopen we kort welke faciliteiten de programmeertaal hiervoor biedt en bespreken we een aantal extra voorzorgen die je moet nemen om er onder JavaFX gebruik van te maken.

### 7.1. Draden in Java

Om een nieuw gedeelte van een programma op te starten naast het reeds lopende programma, moet je een zogenaamde *draad* (*thread*) creëren en die lanceren. Een draad wordt voorgesteld als een object van het type *Thread*.

Het aangeven welk gedeelte van het programma er door een nieuwe draad moet worden uitgevoerd, gebeurt op het moment dat je die draad creëert. Er zijn verschillende manieren om dit te doen, waarvan we er hier slechts één bespreken. Het programmagedeelte dat als draad dient, moet steeds in één of andere klasse worden geplaatst als methode met de volgende signatuur:

```
public void run();
```

Bovendien moet deze klasse de interface *Runnable* implementeren, een interface die precies deze ene methode definieert.

Om dan een nieuwe draad aan te maken, roept men de constructor van *Thread* op met een object van die klasse als enige parameter. De draad wordt dan later opgestart met behulp van de methode *start*.

```
Thread thread = new Thread (someRunnableObject);  
thread.start ();
```

Het opstarten van de draad betekent dat de corresponderende methode *run* van het object wordt uitgevoerd, tegelijkertijd met de andere draden uit de toepassing. De draad stopt

De klasse *Thread* bevat een methode *stop* waarmee je de ene draad vanuit een andere kan laten stoppen. Men heeft echter ingezien dat deze methode een fout bevat — erger nog, dat het in principe niet mogelijk is om een *stop*-methode te schrijven die deze fout niet bevat.

Vandaar dat alle Java-documentatie nu zegt dat je deze methode niet meer mag gebruiken. Hetzelfde geldt voor de methodes *suspend* en *resume*.

vanzelf wanneer de uitvoering van *run* is afgelopen, of wanneer tijdens *run* een uitzondering wordt gegenereerd die niet wordt opgevangen. Dezelfde draad kan slechts één keer worden opgestart.

Het is niet altijd nodig om het draadobject zelf bij te houden. Het vorige fragment wordt dan ook vaak afgekort tot één enkele lijn.

```
new Thread (someRunnableObject).start ();
```

Vaak bestaat de implementatie van *run* uit slechts één of twee opdrachten. In dat geval kan je een anonieme klasse gebruiken:

```
new Thread (new Runnable () {  
    public void run () {  
        sendMailInBackground (message, receiver);  
    }  
}).start ();
```

of een lambda:

```
new Thread ( () -> sendMailInBackground (message, receiver) ).start();
```

Eenmaal een draad loopt, kan je hem in principe niet meer stoppen vanuit een andere draad. Er zit dus niets anders op dan de draad zichzelf te laten stoppen, bijvoorbeeld door hem regelmatig een bepaalde logische variabele te laten controleren die vanuit een andere draad op **false** wordt gezet.

In §3.6 programmeerden we een eerste versie van een *echo server*. Deze server kon echter slechts één client tegelijk aan. Met behulp van draden kunnen we dit gemakkelijk oplossen: na de *accept* van de server-socket verwerken we de connectie met de client in een afzonderlijke draad, zodat de server meteen klaarstaat met een nieuwe *accept*.

```
try (ServerSocket serverSocket = new ServerSocket(port)) {  
    for (; ; ) { // oneindige lus  
        new Thread(new EchoService(serverSocket.accept())).start();  
    }  
}
```



```

} catch (IOException ex) {
    System.err.println("Er gebeurde een fout in de server");
    ex.printStackTrace(System.err);
}

```

De (binnen)klasse *EchoService* verwerkt de communicatie met de client:

```

private record EchoService(Socket socket) implements Runnable {

    public void run() {
        try {
            try (OutputStream out = socket.getOutputStream();
                InputStream in = socket.getInputStream()) {
                int ch = in.read();
                while (ch >= 0) {
                    ou.write(ch);
                    ch = inputStream.read();
                }
            } finally {
                socket.close ();
            }
        } catch (IOException ex) {
            System.err.println("Er gebeurde een fout in de service");
        }
    }
}

```

Merk op dat de socket die *EchoService* heeft meegekregen bij constructie wordt gesloten in het **finally**-gedeelte van de try-met-bronnen. Helaas kan het sluiten van een socket opnieuw een uitzondering veroorzaken, die we dus opvangen in een afzonderlijke **try**.

## 7.2. Draden en JavaFX

[ Onderstaande voorbeelden vind je in *be.ugent.objprog.threads*. ]

Zelfs al ben je niet van plan toepassingen te schrijven die gebruik maken van afzonderlijke draden, toch dien je op de hoogte te zijn van een aantal details over de interactie tussen draden en JavaFX. Elke JavaFX-toepassing bevat immers reeds een aantal draden, waarvan er één een bijzondere rol speelt: de *JavaFX-draad*. Deze verwerkt alle gebeurtenissen en het hertekenen van de vensters.

Het is steeds gevaarlijk om twee verschillende draden dezelfde gegevens te laten wijzigen zonder hiervoor speciale voorzieningen te treffen. Om redenen van efficiëntie neemt JavaFX echter geen dergelijke voorzorgen — men zegt dat JavaFX niet *thread safe* is — dus moet je zelf bijzonder goed oppassen. De belangrijkste regel die je hierbij in acht moet nemen is de volgende:

*Alle programmacode die een GUI-component op één of andere manier wijzigt, moet worden uitgevoerd vanuit de JavaFX-draad.*

JavaFX doet zijn best om je je aan die regel te houden: in vele gevallen zal hij een uitzondering opgooien wanneer je toch een GUI-bewerking doet vanuit een andere draad.

Over het algemeen is het niet moeilijk om aan deze regel te voldoen: de meeste Java-code die je schrijft wordt uiteindelijk opgeroepen als direct gevolg van het indrukken van een knop, het selecteren van een element, het bewegen van de muis, . . . , allemaal acties die gebeurtenissen verwerken en dus worden uitgevoerd in de JavaFX-draad.

Er is echter een belangrijk bijkomend probleem. Omdat de JavaFX-draad ook zorgt voor het afbeelden van de vensters en de componenten en voor het verwerken van standaard muisbewerkingen, mogen gebeurtenisroutines geen langdurige bewerkingen uitvoeren. Anders zal je toepassing niet vlot op de gebruiker blijven reageren. (In het slechtste geval kunnen vensters zelfs ‘bevrozen’: de vensterinhoud verduistert en het besturingsstelsel meldt dat het programma niet langer reageert.)

Er zijn verschillende courante situaties waarbij dit een rol speelt. Bijvoorbeeld:

- Een toepassing heeft een langdurige initialisatie nodig — afbeeldingen of andere gegevensbestanden moeten eerst worden ingeladen over het Internet. In dat geval toont de toepassing meestal een voorlopige GUI terwijl de nodige bewerkingen in een andere draad op de achtergrond worden uitgevoerd. Na afloop van de draad moet dan de GUI worden aangepast.

De initialisatie moet in dit geval buiten de gebeurtenisverwerkingsdraad plaatsvinden, terwijl het aanpassen van de GUI erbinnen moet gebeuren.

- Een toepassing moet wachten op een externe gebeurtenis — informatie die via een netwerkverbinding wordt ontvangen, of gegevens die uit een externe databank worden ingelezen.

Ook dergelijke bewerkingen kan je beter niet in de gebeurtenisverwerkingsdraad uitvoeren.

We hebben hierboven al aangegeven hoe je een bewerking opstart in een andere draad. Om dan vanuit die draad iets aan de GUI te wijzigen, moet je op één of andere manier ‘terugkeren’ naar de JavaFX-draad. Hiervoor gebruik je de methode *runLater* uit de

klasse *Platform*. Deze methode keert niet echt terug naar de JavaFX-draad, maar neemt een stukje code en zorgt ervoor dat dit later zal worden uitgevoerd als onderdeel van de JavaFX-draad.

```
public static void runLater (Runnable runnable);
```

Het codefragment dat je wil uitvoeren, geef je door als een object van een klasse die de interface *Runnable* implementeert. Vaak zal dit een anonieme klasse zijn of een lambda.

We zullen dit illustreren aan de hand van het volgende voorbeeld: een paneel toont enkele afbeeldingen in een rooster van 4 kolommen. Bij de start van het programma is het rooster leeg, maar je kan de afbeeldingen inladen door op een knop te drukken. Het resultaat worden pas getoond nadat alles is opgehaald. Ondertussen wordt enkel de tekst 'Even geduld ...' getoond.

Als basisstructuur voor het programma gebruiken we een *StackPane* met daarop enerzijds een label *placeholder* (met de tekst 'Even geduld...') en anderzijds een roosterpaneel (*GridPane*) van vier kolommen.

Initieel is het label niet zichtbaar (*setVisible(false)*) en het roosterpaneel wel. Tijdens het laden maken we het label terug zichtbaar (en het roosterpaneel onzichtbaar) en na afloop keren we de situatie terug om. Dit alles zouden we kunnen implementeren op de volgende manier (de methode *load* wordt opgeroepen bij het indrukken van de knop):

```
private static final String[] IMAGE_NAMES = { "aardbei", "ajuin", ...};
```

```
public void load() {  
    placeholder.setVisible(true);  
    gridPane.setVisible(false);  
    Image[] images = new Image[IMAGE_NAMES.length];  
    for (int i = 0; i < images.length; i++) {  
        images[i] = new Image(IMAGE_NAMES[i] + ".png");  
    }  
    int nrOfColumns = gridPane.getColumnConstraints().size();  
    for (int i = 0; i < images.length; i++) {  
        gridPane.add(new ImageView(images[i]),  
                     i % nrOfColumns, i / nrOfColumns);  
    }  
    placeholder.setVisible(false);  
    gridPane.setVisible(true);  
}
```

Deze implementatie is echter verre van ideaal. Terwijl de afbeeldingen worden ingeladen, reageert de GUI niet op de gebruiker, wat heel vervelend is als dit een tijdje duurt.

Om dit probleem te verhelpen, doen we het volgende: we starten een nieuwe draad op die alle afbeeldingen inlaadt en pas wanneer de draad is afgelopen, tonen we alle afbeeldingen in het venster. Dit laatste moet echter terug in de JavaFX-draad gebeuren. Daarom splitsen we het inladen en het tonen van de afbeeldingen op in twee afzonderlijke methodes: een methode *loadImages* die ze inlaadt, en een methode *imagesLoaded* die ze in het rooster plaatst. De implementatie van *imagesLoaded* is eenvoudig:

```
private void imagesLoaded() {
    int nrOfColumns = gridPane.getColumnConstraints().size();
    for (int i = 0; i < images.length; i++) {
        gridPane.add(new ImageView(images[i]),
                    i % nrOfColumns, i / nrOfColumns);
    }
    placeHolder.setVisible(false);
    gridPane.setVisible(true);
}
```

De tabel *images* is een veld van de klasse waarin deze methodes zich bevinden. Ze wordt aangemaakt en opgevuld door *loadImages*:

```
private void loadImages() {
    images = new Image[IMAGE_NAMES.length];
    for (int i = 0; i < images.length; i++) {
        images[i] = new Image(IMAGE_NAMES[i] + ".png");
    }
    Platform.runLater(this::imagesLoaded);
}
```

Het laatste wat *loadImages* doet, is *imagesLoaded* opstarten, maar dan wel in de JavaFX-draad. We kunnen *imagesLoaded* niet direct oproepen omdat we *loadImages* in een andere dan de JavaFX-draad willen uitvoeren. Deze draad wordt gelanceerd zodra er op de knop wordt gedrukt:

```
public void load() {
    placeHolder.setVisible(true);
    gridPane.setVisible(false);
    new Thread(this::loadImages).start();
}
```

Merk op dat het werk nu verdeeld is over *drie* verschillende plaatsen, afhankelijk van wanneer en in welke draad het wordt uitgevoerd.

## 7.3. De klasse *Task*

JavaFX biedt een alternatieve manier om bovenstaande te implementeren door gebruik te maken van de abstracte klasse *Task<T>*. Je moet deze klasse uitbreiden en de taak die je wil laten uitvoeren, implementeren als corpus van de methode *call* die je moet overschrijven.

Er zijn enkele essentiële verschillen tussen een *Task* en een *Runnable*.

- De methode *call* heeft een retourwaarde en *Runnable* niet. De bedoeling van een taak is m.a.w., dat ze een bepaalde *waarde* genereert. Het type van die waarde is de parameter *T* van de klasse *Task<T>*.
- *Task* bezit enkele eigenschappen waar je *vanuit de JavaFX-draad* kunt naar luisteren. Op die manier kan je bijvoorbeeld op de hoogte worden gebracht dat de taak is beëindigd, of dat er een bepaald percentage van de taak is volbracht.
- *Task* heeft enkele methodes waarmee je deze eigenschappen kunt instellen *vanuit de draad waarin de taak wordt uitgevoerd*.

Willen we deze techniek gebruiken voor het voorbeeld uit de vorige paragraaf, dan implementeren we een klasse *ImageLoaderTask* die een tabel van afbeeldingen inleest en teruggeeft als waarde. (Het type van onze taak is dus *Task<Image[]>*.)

In principe verschilt de inhoud van *call* weinig van de (impliciete) *run* die we eerder hebben geschreven, alleen wordt het plaatsen van de afbeeldingen in het rooster nu nog niet in gang gezet:

```
public class ImageLoaderTask extends Task<Image[]> {  
  
    private static String[] IMAGE_NAMES = ...  
  
    @Override  
    protected Image[] call() throws Exception {  
        Image[] images = new Image[IMAGE_NAMES.length];  
        for (int i = 0; i < images.length; i++) {  
            images[i] = new Image(IMAGE_NAMES[i] + ".png");  
        }  
        return images;  
    }  
}
```

Om de afbeeldingen uiteindelijk in het venster te krijgen, gebruiken we dezelfde *imagesLoaded* als voorheen, alleen wordt die nu niet opgestart door een *Platform.runLater*.

In de plaats daarvan luisteren we naar de *status* van de taak en reageren we als die verandert.

Een taak kan de volgende statuswaarden aannemen

- *READY* of *SCHEDULED*, wanneer de taak nog niet is opgestart. (Het verschil tussen beide statussen is voor ons niet belangrijk.)
- *RUNNING*, terwijl de taak loopt.
- *SUCCEEDED*, als de taak met goed gevolg is afgelopen. In dit geval kan je de waarde die de taak heeft ‘berekend’, opvragen met *getValue*.
- *FAILED*, als er onderweg iets is misgegaan. M.a.w., wanneer er een uitzondering werd opgegooid die niet werd opgevangen — bijvoorbeeld als de taak iets aan de GUI probeert te veranderen.
- *CANCELLED*, wanneer de taak werd geannuleerd.

In beide laatste gevallen gaat de eventueel reeds gedeeltelijk berekende waarde van de taak onherroepelijk verloren.

Vooraleer we de taak opstarten, registreren we daarom een luisteraar bij de *state*-eigenschap van die taak:

```
task = new ImageLoaderTask();
task.stateProperty().addListener(...);
```

De luisteraar bekijkt de status en reageert op gepaste wijze.

```
public void invalidated (Observable o) {
    if (task.getState() == Worker.State.SUCCEEDED) {
        int nrOfColumns = gridPane.getColumnConstraints().size();
        Image[] images = task.getValue();
        for (int i = 0; i < images.length; i++) {
            gridPane.add(new ImageView(images[i]),
                           i % nrOfColumns, i / nrOfColumns);
        }
        placeHolder.setVisible(false);
        gridPane.setVisible(true);
    } else if (task.getState() == Worker.State.FAILED) {
        placeHolder.setText("Er ging iets fout :-(");
    }
}
```

De luisteraar wordt ook opgeroepen met andere statuswaarden, bijvoorbeeld wanneer de taak opstart, maar in die gevallen hoeft er niets te gebeuren. Merk op hoe we *getValue* gebruiken om de tabel van afbeeldingen op te vragen die door de taak is aangemaakt.

De draad waarin je de taak lanceert, start je op zoals bij elke andere *Runnable*. Dit is onze nieuwe versie van *load*:

```
public void load() {  
    placeHolder.setVisible(true);  
    gridPane.setVisible(false);  
    task = new ImageLoaderTask();  
    task.stateProperty().addListener(...);  
    new Thread(task).start();  
}
```

In plaats van naar de *state*-eigenschap van een taak te luisteren, kan je ook een gebeurtenisluisteraar rechtstreeks bij de taak registreren. Dit doe je met één van de methodes *setOnSucceeded*, *setOnFailed*, ... Een dergelijke luisteraar is van het type *EventHandler* *<WorkerStateEvent>*. Ook hier gebruik je meestal een anonieme klasse of een lambda:

```
task.setOnFailed(ev -> placeHolder.setText("Er ging iets fout :-("));
```

De (abstracte) klasse *Task* luistert trouwens zelf reeds naar elke van die gebeurtenissen door een overeenkomstige methode *failed()*, *succeeded()*, ... op te roepen. De implementatie van deze methodes is leeg — het is de bedoeling dat je ze in je eigen taakklasse overschrijft. Merk op dat deze methodes gegarandeerd in de JavaFX-draad worden opgeroepen, ook al maken ze deel uit van de *Task*-klasse.

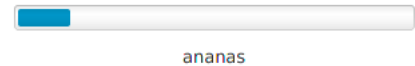
Als derde alternatief, kunnen we dus ook de volgende methode opnemen in de implementatie van *ImageLoaderTask*:

```
@Override  
protected void failed() {  
    placeHolder.setText("Er ging iets fout :-(");  
}
```

Opdat dit zou werken, moet de component *placeHolder* toegankelijk zijn voor de taakklasse, en in de praktijk is dit niet altijd zo handig — behalve misschien wanneer de taakklasse een binnenklasse is van de klasse die de GUI beheert.

## 7.4. Het tonen van voortgang

Een label met de tekst ‘Even geduld’ is niet zo veelzeggend. Het is dan ook gebruikelijk om de voortgang van het programma op een meer dynamische manier aan te geven, bijvoorbeeld met behulp van een zogenaamde *voortgangsbalk* (klasse *ProgressBar*). In het voorbeeld dat we hieronder uitwerken, plaatsen we onder deze balk nog een label dat telkens de naam toont van de afbeelding die op dat moment wordt ingeladen (zie afbeelding).



Een *ProgressBar*-component heeft een *progress*-eigenschap die een reële waarde bevat tussen 0 en 1 die aangeeft welk percentage van de balk dient gekleurd te worden. Met *setProgress* kan je die waarde instellen. We zouden dit kunnen doen als onderdeel van de **for**-lus in onze taak, maar dan mogen we niet vergeten om dit in te pakken in een *runLater*.

De klasse *Task* biedt ons echter een andere manier om dit te doen. Elk object van die klasse heeft immers zelf een *progress*-eigenschap die we vanuit de JavaFX-draad kunnen beluisteren. Om de waarde van deze eigenschap in te stellen vanuit de taak zelf, werd de volgende methode voorzien<sup>1</sup>:

```
protected void updateProgress(long workDone, long max)
```

Deze methode heeft twee argumenten: het aantal stappen dat reeds voltooid is en het totaal aantal stappen dat wordt verwacht. We hoeven dus zelf de deling niet te maken. (Er is ook een variant met twee reële waarden als parameter.)

In de JavaFX-draad zullen we luisteren naar de *progress*-eigenschap van de taak, en telkens wanneer die verandert, passen we de waarde van de *progress*-eigenschap van de balk aan zodat hij exact dezelfde waarde krijgt. We zouden hiervoor zelf een luisteraar kunnen schrijven, maar dit is een ideale gelegenheid om nog eens gebruik te maken van ‘bindings’ (zie §5.9) — we hoeven slechts beide eigenschappen met elkaar te verbinden.

```
progressBar.progressProperty().bind(task.progressProperty());
```

De klasse *Task* heeft nog meer eigenschappen die we op deze manier kunnen gebruiken. Er zijn de *message*- en *title*-eigenschappen met corresponderende *update*-methodes, en een *running*-eigenschap die aangeeft of de taak zich in de *SCHEDULED*- of *RUNNING*-toestand bevindt.

---

<sup>1</sup>Er is geen (publieke) methode *setProgress*. Een dergelijke methode zou niet veel zin hebben, omdat ze vanuit de JavaFX-draad moet worden opgeroepen.



Wij gebruiken de *message*-eigenschap om telkens te tonen welke afbeelding er wordt ingeladen,

```
progressMessage.textProperty().bind(task.messageProperty());
```

en de *running*-eigenschap om de voortgangsbalk al dan niet zichtbaar te maken.

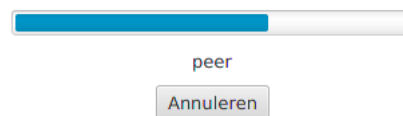
```
progressBar.visibleProperty().bind(task.runningProperty());
```

Na al deze aanpassingen ziet de *call*-methode van de taak er als volgt uit:

```
@Override
protected Image[] call() throws Exception {
    Image[] images = new Image[IMAGE_NAMES.length];
    updateProgress(0, images.length);
    for (int i = 0; i < images.length; i++) {
        images[i] = new Image(IMAGE_NAMES[i] + ".png");
        updateProgress(i+1, images.length);
        updateMessage(IMAGE_NAMES[i]);
    }
    return images;
}
```

## 7.5. Een taak annuleren

Wanneer het inladen van de afbeeldingen te lang duurt, wil de gebruiker de taak misschien annuleren. Hiervoor kan je de methode *cancel* gebruiken uit *Task*. Dit klinkt rechttoe rechtaan, maar er zit een addertje onder het gras.



Zoals we al eerder hebben aangegeven, is er geen betrouwbare methode om een draad van buitenaf te stoppen en moet een draad in de praktijk steeds zichzelf beëindigen. Dit betekent dat *cancel* eigenlijk niet veel meer doet dan ergens een logische waarde instellen die aangeeft dat de taak geannuleerd werd. Het is echter nog altijd aan de programmeur om hierop te reageren.

In de lus (in *call*) die de verschillende afbeeldingen inlaadt, dienen we bij elke stap te kijken of de taak niet werd geannuleerd (met *isCancelled*) en wanneer dit het geval is, de lus voortijdig te beëindigen. We doen dit met een bijkomende conditie.

```

protected Image[] call() throws Exception {
    Image[] images = new Image[IMAGE_NAMES.length];
    updateProgress(0, images.length);
    int i = 0;
    while (i < images.length && !isCancelled()) {
        images[i] = new Image(IMAGE_NAMES[i] + ".png");
        updateProgress(i+1, images.length);
        updateMessage(IMAGE_NAMES[i]);
        i++;
    }
    return images;
}

```

Er is nog een bijkomend ongemak: wanneer een taak met *cancel* wordt onderbroken, dan kunnen we met *getValue* de waarde niet meer opvragen — het systeem denkt immers dat die waarde niet geldig is. In ons voorbeeld kunnen we dus niet zomaar de afbeeldingen die reeds zijn ingeladen toch nog tonen.

Om dit op te lossen, moeten we de structuur van het programma aanpassen. Dit kan op verschillende manieren — wij kiezen ervoor om de prenten één voor één af te beelden zodra ze ter beschikking zijn en de verantwoordelijkheid hiervoor bij de taak te leggen.

```

public class ImageLoaderTask extends Task<Void> {

    private static String[] IMAGE_NAMES = ...
    private GridPane gridPane;

    public ImageLoaderTask(GridPane gridPane) {
        this.gridPane = gridPane;
    }

    protected Void call() throws Exception {
        int nrOfColumns = gridPane.getColumnConstraints().size();
        for (int i = 0; i < IMAGE_NAMES.length; i++) {
            Image image = new Image(IMAGE_NAMES[i] + ".png");
            int row = i % nrOfColumns;
            int col = i / nrOfColumns;
            Platform.runLater(
                () -> gridPane.add(new ImageView(image), row, col)
            );
        }
        return null;
    }
}

```

Omdat de taak nu geen echte waarde meer heeft, declareren we haar als van het type *Task<Void>*.

Merk op dat we het roosterpaneel waarop de afbeeldingen terechtkomen als argument moeten meegeven bij de constructor. We gebruiken nu ook geen afzonderlijke voortgangs-indicatie en er bestaat geen mogelijkheid meer tot annuleren.

De *Platform.runLater* op het einde van de **for**-lus is verplicht, want de taak wordt niet in de JavaFX-draad uitgevoerd. Ook de twee variabelen *row* en *col* moeten er zijn, omdat Java de lambda niet zal aanvaarden als we daarbinnen *i* rechtstreeks gebruiken<sup>2</sup>.

---

<sup>2</sup>De variabelen die je in een lambda gebruikt moeten ‘*effectively final*’ zijn — hun waarde mag niet meer kunnen veranderen nadat het object is aangemaakt dat met de lambda correspondeert. Als de variabele *i* hier zou toegelaten zijn, dan zou het voor Java niet duidelijk zijn of we de waarde van *i* wensen te gebruiken zoals ze was op het moment dat de lambda werd aangemaakt, of zoals ze is op het moment dat hij uiteindelijk wordt uitgevoerd.



## 8. Complexe componenten

De meeste van de standaardcomponenten die door de JavaFX-bibliotheek worden geleverd, zijn vrij rechtlijnig in gebruik. Meestal volstaat het om de elektronische documentatie eens diagonaal door te lezen om een dergelijke component in je programma te kunnen gebruiken. Er zijn echter enkele belangrijke componenten — *combo boxes*, lijsten, tabellen en boomdiagrammen — die iets meer achtergrondkennis vergen. De corresponderende componentklassen heten *ComboBox*, *ListView*, *TableView* en *TreeView*.

Het is belangrijk te weten dat elk van deze componenten het MVC-patroon gebruikt (zie hoofdstuk 5) om zijn gegevens bij te houden en voor te stellen.

### 8.1. De combo box en de cell factory

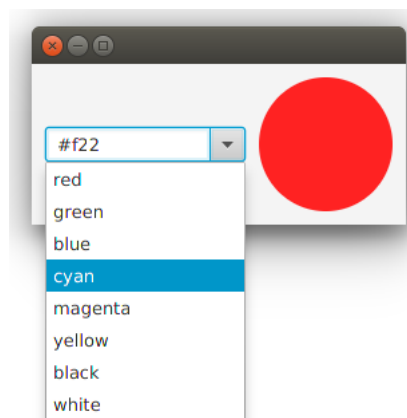
[ Onderstaande voorbeelden vind je in het pakket *be.ugent.objprog.combo*. ]

Een *ComboBox*-component is de meer ingewikkelde versie van de *ChoiceBox* die we al eerder hebben ontmoet (zie bijvoorbeeld §4.1).

De *choice box* is bedoeld om de gebruiker te laten kiezen tussen een *klein* aantal voorafbepaalde mogelijkheden en heeft een gebruikersinterface die lijkt op die van een menu.

Bij de *combo box* kan je een ruimer scala aan keuzes aanbieden — standaard verschijnt er een *scroll bar* bij meer dan 10 keuzes — en de gebruikersinterface is gebaseerd op die van een lijst — de combo box bevat trouwens een *ListView* als deelcomponent. Bovendien kan je een combo box ook *editeerbaar* maken: je kan dan ook een keuze intikken die niet tot de voorgestelde lijst behoort.

Het voorbeeld dat we hier zullen behandelen, gebruikt een combo box om kleuren te kiezen (die dan de kleur bepalen van een cirkel). Er worden acht standaardkleuren aangeboden maar je kan ook zelf een kleur intikken, op verschillende manieren ('red', '#ff0000', 'rgb(255,0,0)', ... — zie *javafx.scene.paint.Color*).



Het FXML-bestand voor het venster dat we hierboven hebben afgebeeld, heeft de volgende vorm:

```
<HBox ... fx:controller="be.ugent.objprog.combo.ComboCompanion">
  <ComboBox fx:id="comboBox" prefWidth="150.0"
    editable="true" value="green">
    <items>
      <FXCollections fx:factory="observableArrayList">
        <String fx:value="red"/>
        ...
        <String fx:value="white"/>
      </FXCollections>
    </items>
  </ComboBox>
  <Circle fx:id="circle" fill="GREEN" radius="50.0"/>
</HBox>
```

De combo box heeft drie eigenschappen die we hier invullen:

- *items* Een lijst van strings met alle keuzes van het type *ObservableList*, meer bepaald *ObservableList<String>*, een bijzondere versie van *List<String>* — meer details over dit type volgen later (zie §8.4).
- *value* De waarde die door de gebruiker werd geselecteerd of ingetikt.
- *editable* Geeft aan of je in het combo box-veld een waarde mag intikken of niet.

Merk op dat *editable* en *value* in het FXML-bestand als *attributen* van het `ComboBox`-element zijn geconfigureerd. Het is bovendien belangrijk dat *editable* vermeld wordt vóór *value*. De documentatie vermeldt immers dat de waarde van een combo box wordt teruggezet wanneer *editable* verandert. We mogen dus de waarde pas instellen nadat we de component hebben editeerbaar gemaakt<sup>1</sup>.

De partnerklasse heeft een eenvoudige implementatie:

```
public class ComboCompanion {

    public ComboBox<String> comboBox;

    public Circle circle;
```

---

<sup>1</sup>De XML-standaard zegt dat de volgorde van attributen niet van belang mag zijn. Dat betekent dus bijvoorbeeld dat de Scene Builder in principe de volgorde van de attributen mag omkeren wanneer we dit FXML-bestand inladen en terug opslaan. Het zou in dit geval dus veiliger zijn om de waarde van de combo box pas in de *initialize* van de partnerklasse een waarde te geven.

```

public void initialize() {
    comboBox.setOnAction( ev ->
        circle.setFill(Color.web(comboBox.getValue()))
    );
}

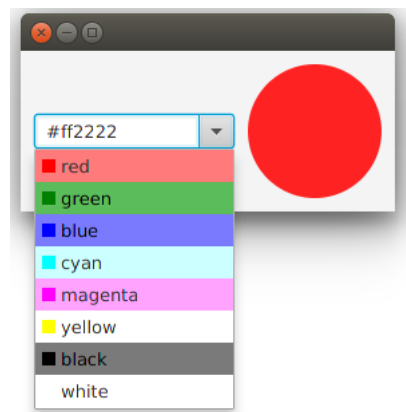
```

(De methode *Color.web* zet een string om naar een kleur volgens de naamgeving die gebruikelijk is op webpagina's.)

Het type van de combo box-component is *ComboBox<String>* omdat de onderliggende lijst een lijst van strings is. Zoals we later zullen zien, kunnen we hier in de plaats ook een lijst van *Color*-objecten gebruiken — en dan wordt het type *ComboBox<Color>*.

Tot zover biedt deze implementatie niet al te veel verrassingen. Het wordt interessanter wanneer we het programma een beetje opfleuren zoals in de afbeelding hiernaast: bij elke keuze tonen we een rechtehoekje van de correcte kleur en we passen ook de achtergrondkleur overeenkomstig aan.

Dit is mogelijk omdat de lijstelementen die worden getoond zich als labels gedragen, en een label kan behalve tekst ook een 'afbeelding' (*graphic*) bevatten.



Een *ListView* wordt opgebouwd uit cellen van het type *ListCell*. De gewone *ListCell* toont zijn inhoud enkel als tekst, maar dit kunnen we aanpassen door zelf een eigen type cel te implementeren en bij de combo box te registreren als zogenaamde *cell factory*. Deze cell factory heeft als taak nieuwe cellen aan te maken voor de combo box-lijst.

Een cell factory voor lijsten moet voldoen aan de (ingewikkelde) interface *Callback<ListView<String>, ListCell<String>>*. De interface *Callback* heeft de volgende signatuur:

```

public interface Callback<P,R> {
    R call (P param);
}

```

Ze wordt in JavaFX op verschillende plaatsen gebruikt om een klasse aan te duiden met een methode die één parameter neemt van het type *P* en een retourwaarde heeft van het type *R*.

Een cell factory (voor een lijst) is dus een dergelijke *callback* die een *ListView<T>* als

parameter neemt en als resultaat een cel teruggeeft van het type *ListCell<T>*. *T* geeft het type aan van de elementen in de lijst — in ons voorbeeld is dit *String*.

Het JavaFX-team heeft voor callbacks gekozen omdat ze wilden vooruitlopen op de introductie van lambda's in versie 8 van Java. En inderdaad, als lambda valt een dergelijke cell factory wel mee:

```
comboBox.setCellFactory(list -> new ComboCell());
```

Merk op dat onze cell factory geen gebruik maakt van het argument *lijst* en dat zal in de meeste gevallen zo zijn.

Het is de klasse *ComboCell*, een klasse die we zelf moeten implementeren, die het uitzicht bepaalt van de keuzelijst van de combo box. Deze klasse moet een uitbreiding zijn van *ListCell<String>* — anders zou de signatuur van de cell factory niet kloppen — die op haar beurt een uitbreiding is van *Labeled*, en zich m.a.w. als een label gedraagt. We voegen een rechthoekje van 10 bij 10 pixels toe aan dit label, als *graphic*, en plaatsen dit rechthoekje links van de tekst.

```
public class ComboCell extends ListCell<String> {  
  
    private final Rectangle rectangle;  
  
    public ComboCell() {  
        this.rectangle = new Rectangle(10,10);  
        setContentDisplay(ContentDisplay.LEFT);  
    }  
  
    ...  
}
```

In bovenstaande code ontbreekt er duidelijk nog het één en ander: we moeten bijvoorbeeld ergens in het programma de specifieke kleur van het rechthoekje instellen, en deze is niet voor alle cellen dezelfde. Om dit op een correcte manier te kunnen doen, moeten we begrijpen op welke manier JavaFX zijn cellen beheert en 'recycleert'.

Cellen zijn zelf componenten en nemen daarom behoorlijk wat geheugen in beslag. Om die reden vermijdt JavaFX dan ook om meer cellen aan te maken dan nodig om de component op het scherm te kunnen afbeelden. Met andere woorden, zelfs wanneer bijvoorbeeld een lijst duizenden objecten bevat, zullen er toch maar enkele tientallen cellen worden aangemaakt, genoeg om het zichtbare gedeelte van deze tabel te kunnen afbeelden.

Een bepaalde cel wordt daarom tijdens zijn levensduur gebruikt voor verschillende elementen van dezelfde component. Wanneer we bijvoorbeeld door een lijst *scrollen*, dan



blijven we steeds dezelfde cellen zien, maar telkens met andere inhoud. Wanneer de inhoud van een cel verandert, signaleert JavaFX dit aan de cel door de volgende methode op te roepen:

```
protected void updateItem(T item, boolean empty)
```

De parameter *item* bevat de nieuwe waarde die door de cel moet worden afgebeeld. Omdat **null** in sommige gevallen een perfect toelaatbare waarde kan zijn, is er een tweede parameter om aan te geven of een cel al dan niet *leeg* is. Wanneer een lijst bijvoorbeeld slechts één element bevat, maar groot genoeg is afgebeeld om er tien te tonen, dan zijn de laatste negen cellen leeg. (Bij een combo box zal dit zich wellicht niet voordoen, maar we kunnen beter zeker spelen). Heeft de parameter *empty* de waarde **true**, dan is de cel leeg, en is de waarde van *item* niet relevant.

In ons voorbeeld is de implementatie van *updateItem* vrij eenvoudig:

```
protected void updateItem(String item, boolean empty) {  
    super.updateItem(item, empty);  
    if (empty) {  
        setGraphic(null);  
        setText(null);  
    } else {  
        rectangle.setFill(Color.web(item));  
        setGraphic(rectangle);  
        setText(item);  
        setStyle("-fx-base: " + item);  
    }  
}
```

Bij een lege cel zorgen we ervoor dat het label noch tekst noch afbeelding toont en anders tonen we als tekst de waarde van *item* en zetten die om met *Color.web* om de vulkleur van de rechthoek in te stellen. De *setStyle*-lijn is een trucje om met behulp van CSS de achtergrondkleur in te stellen als een lichtere versie van de opgegeven kleur.

Merk op dat we als eerste opdracht de *updateItem* van *ListCell* oproepen. Vergeet dit niet, anders kan dit eigenaardige fouten veroorzaken die moeilijk zijn op te sporen!

In het bovenstaande ontwerp beschouwen we de combo box als een lijst van strings. Ergens lijkt het echter logischer om die als een lijst van kleuren te bekijken. Hiertoe moeten we enkele veranderingen aanbrengen in het programma.

Ten eerste moet het type van de combo box gewijzigd worden in *ComboBox<Color>* en moeten de *items*- en de *value*-eigenschappen in het FXML-bestand met kleuren worden geïnitialiseerd en niet met strings.

```

<ComboBox fx:id="comboBox" prefWidth="150.0">
  <items>
    <FXCollections fx:factory="observableArrayList">
      <Color fx:value="RED"/>
      ...
      <Color fx:value="WHITE"/>
    </FXCollections>
  </items>
  <editable>
    <Boolean fx:value="true"/>
  </editable>
  <value>
    <Color fx:value="GREEN"/>
  </value>
</ComboBox>

```

Merk op dat *value* niet meer als attribuut van het *ComboBox*-element kan opgenomen worden, maar als een volwaardig XML-element moet worden ingepast<sup>2</sup>.

Als tweede belangrijke stap moeten we het tekstveld configureren waarmee de gebruiker in de combo box zelf een kleur kan intikken. Wat er in dit tekstveld wordt ingetikt, moet worden omgezet naar een kleur om als *value* te kunnen dienen, en JavaFX weet niet vanzelf hoe dit moet gebeuren.

Daarom moeten we een gepaste *string converter* (zie §5.9) bij de combo box registreren die kleuren omzet naar strings en omgekeerd<sup>3</sup>.

```

private static final StringConverter<Color> CONVERTER
    = new ColorConverter();

public void initialize () {
    combobox.setConverter (CONVERTER);
}

```

(We gebruiken hier een constante *CONVERTER* omdat we die zelfde converter ook nog op andere plaatsen in het programma zullen nodig hebben.)

Omdat het type van de combobox veranderd is naar *ComboBox<Color>*, wijzigt ook het type van de cell factory en dus het type van de cel. We hebben nu een *ColorCell* van het

<sup>2</sup>De eigenschap *editable* is wel nog als attribuut toegelaten, maar dat heeft een ongewenst effect. Kan je raden welk? Tip: De FXML-loader stelt de attributen pas in na de deelelementen.

<sup>3</sup>We laten de implementatie van deze converter over ‘als oefening’. Omzetten van kleur naar string kan je doen met *Color.web*, het omgekeerde is minder eenvoudig, tenminste wanneer je wil dat de computer ook *kleurnamen* kan teruggeven. Tip: je kan met behulp van *reflectie* de namen en de waarden van de kleurconstanten uit de klasse *Color* ophalen.

type *ListCell<Color>*. De methode *updateItem* neemt daarom een *Color* als parameter, en niet langer een string. De conversies die we moeten toepassen, gebeuren nu in de omgekeerde richting: van kleur naar string:

```
public static class ComboCell extends ListCell<Color> {  
    ...  
    protected void updateItem(Color item, boolean empty) {  
        super.updateItem(item, empty);  
        if (empty) {  
            setGraphic(null);  
            setText(null);  
        } else {  
            rectangle.setFill(item);  
            setGraphic(rectangle);  
            String itemString = CONVERTER.toString(item);  
            setText(itemString);  
            setStyle("-fx-base: " + itemString);  
        }  
    }  
}
```

en gelukkig hebben we al een converter die dit voor ons kan doen.

Merk op dat de keuzelijst in ons voorbeeld niet verandert in de loop van het programma. Bij combo boxen komt dit weinig voor, maar bij andere lijsten gebeurt het vaak dat er elementen *at run time* worden toegevoegd of verwijderd.

JavaFX behandelt dergelijke aanpassingen door het MVC-patroon toe te passen: voegen we tijdens de loop van het programma een nieuw element toe aan de *items*-lijst dan wordt de *ListView* hiervan op de hoogte gebracht. Deze initialiseert dan automatisch een nieuwe cel met de nieuwe waarde (met behulp van *updateItem*). We hoeven als programmeur geen verdere actie te ondernemen.

Veel van wat we in deze paragraaf hebben besproken, kan worden overgedragen op boomdiagrammen en tabellen. Ook zij hebben bijvoorbeeld een cell factory, met cellen van het type *TreeCell* en *TableCell*. In paragraaf 8.5 komen we hierop terug.

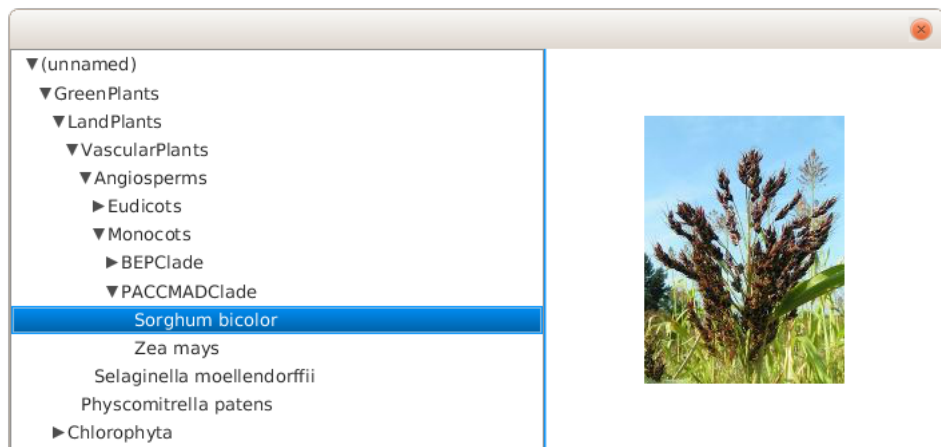
## 8.2. Een boomdiagram

[ Dit voorbeeld vind je in *be.ugent.objprog.phylotree.* ]

Vooraleer we ons verdiepen in de complexiteiten van tabelcomponenten bekijken we kort de zogenaamde *boomdiagrammen* — componenten van de klasse *TreeView*.

In het eenvoudige voorbeeld dat we hier bespreken, is de inhoud van het diagram onveranderlijk. Je kan boomdiagrammen echter ook zodanig configureren dat ze ter plekke kunnen worden geëditeerd.

Werken met een boomdiagram is in zekere zin eenvoudiger dan met een tabel. De *TreeView* heeft geen deelcomponenten behalve cellen (van het type *TreeCell*) en van deze laatste hoeven we ons niets aan te trekken bij een boom waarvan de inhoud niet verandert.



Het boomdiagram in het linkergedeelte van bovenstaande schermafbeelding stelt een zogenaamde *fylogenetische boom* voor<sup>4</sup>. Deze boom is opgebouwd vanuit de inhoud van een XML-bestand<sup>5</sup>:

```
<phyloxml ...>
  <phylogeny rooted="false">
    <clade>
      <clade>
        <name>GreenPlants</name>
        <branch_length>0.09</branch_length>
```

<sup>4</sup>Een fylogenetische boom is in de biologie een schema dat de evolutionaire geschiedenis van een bepaalde groep verwante biologische soorten (of andere taxa) weergeeft — zie Wikipedia.

<sup>5</sup>In phyloXML-formaat — zie <http://www.phyloxml.org>.

```

    <clade>
      <name>LandPlants</name>
      <branch_length>0.09</branch_length>
    </clade>
    ...
    <clade>
      <name>Sorghum bicolor</name>
      <branch_length>0.28</branch_length>
      <uri type="image">...</uri>
    </clade>
  </clade>
</clade>
</phylogeny>
</phyloxml>

```

De `<uri>`-elementen bevatten een URL van een afbeelding voor het corresponderend organisme. Deze afbeelding wordt dan getoond wanneer het overeenkomstige element in het boomdiagram is geselecteerd<sup>6</sup>. (De `<branch_length>`-elementen gebruiken we niet in onze toepassing.)

Met behulp van JDOM (§3.8) zetten we dit XML-bestand om naar een lijst van objecten van een record-klasse *Phylogeny*. Een dergelijk record bevat dan een verwijzing naar een object van een record-klasse *Clade*. Een *Clade*-record bevat een *name* en een *uri* (beiden *Strings*) en een lijst (*cladeList*) van andere *Clade*-records (die eventueel leeg kan zijn).

Als model gebruikt een boomdiagram een object van het type *TreeItem* dat de wortel van de boom voorstelt. *TreeItems* kunnen kinderen hebben die op hun beurt *TreeItems* zijn. Elk *TreeItem* heeft ook een *waarde*. Het is deze waarde die telkens wordt afgebeeld in de knopen en blaadjes van de boom.

De waarde van een *TreeItem* hoeft niet een *String* te zijn, je kan hiervoor een eigen type kiezen — dat dan wel hetzelfde moet zijn voor de ganse boom. Kiezen we voor het waardetype *T* dan heeft het boomdiagram als type *TreeView<T>* en is zijn model opgebouwd uit objecten van het type *TreeItem<T>*.

Bij dit voorbeeld kiezen we voor een boom van *NameURIPair*-objecten. (Voor het boomdiagram zelf zou *String* hebben volstaan, maar we hebben de URI uiteindelijk nodig om de geselecteerde afbeelding te kunnen tonen.)

De implementatie van *NameURIPair* is rechttoe rechtaan:

---

<sup>6</sup>In de schermafdruck gaat het om een foto van een ‘kafferkoren’plant. Informatica kan ook leerzaam zijn.

```

public class NameURIPair {

    private String name;

    private String uri;

    ...

    public NameURIPair(String name, String uri) {
        this.name = name;
        this.uri = uri;
    }

    public String toString() {
        if (name == null) {
            return " (unnamed) ";
        } else {
            return name;
        }
    }
}

```

Bij een standaardboomdiagram wordt de methode *toString* gebruikt om te bepalen welke tekst er wordt geplaatst in elke knoop of blad.

Om het boomdiagram op te bouwen, construeren we dus eerst de boom van *TreeItems* uit de lijst van *Phylogeny*-objecten (en hun kinderen). Dit doen we op een recursieve manier:

```

public TreeItem<NameURIPair> convertPhyloXML(List<Phylogeny> list) {
    TreeItem<NameURIPair> ti = new TreeItem<>();
    for (Phylogeny phylogeny : list) {
        ti.getChildren().add(convertPhylogeny(phylogeny));
    }
    return ti;
}

private TreeItem<NameURIPair> convertPhylogeny(Phylogeny p) {
    TreeItem<NameURIPair> ti = convertClade(p.clade());
    ti.setExpanded(true);
    return ti;
}

```

```

private TreeItem<NameURIPair> convertClade(Clade clade) {
    NameURIPair pair = new NameURIPair(clade.name(),clade.uri());
    TreeItem<NameURIPair> ti = new TreeItem<>(pair);
    Iterable<Clade> list = clade.cladeList();
    if (list != null) {
        for (Clade child : list) {
            ti.getChildren().add(convertClade(child));
        }
    }
    return ti;
}

```

Na al deze voorbereiding kunnen we nu het boomdiagram initialiseren in drie lijnen:

```

TreeItem<NameURIPair> treeItem = convertPhyloXML (...);
treeView.setRoot(treeItem);
treeView.setShowRoot(false);

```

De laatste lijn duidt aan dat de wortel van de boom (het *TreeItem* dat correspondeert met het *PhyloXML*-object) niet hoeft getoond te worden. Het bovenste element (“unnamed”) in de schermafdruck correspondeert inderdaad met het onderliggende *Phylogeny*-object.

Hoe zorgen we ervoor dat we in het rechterpaneel van het venster steeds de afbeelding tonen die overeenkomt met het geselecteerde boomdiagramelement? We behandelen dit in de volgende paragraaf.

## 8.3. Het selectiemodel

Elk boomdiagram, elke lijst en elke tabel bezit een zogenaamd *selectiemodel*. Je kan dit bij de *TreeView* opvragen met *getSelectionModel()*. Dit model houdt bij welke elementen van de component er door de gebruiker werden geselecteerd<sup>7</sup>.

Je kan (bij het model) instellen of er hoogstens één element tegelijk kan worden geselecteerd — de standaardinstelling — of meerdere tegelijk. Dit heet de *selectiemodus*:

```

selectionModel.setSelectionMode (SelectionMode.MULTIPLE); // of
selectionModel.setSelectionMode (SelectionMode.SINGLE);

```

<sup>7</sup>Bij lijsten en boomdiagrammen worden afzonderlijke cellen geselecteerd. Bij tabellen zijn de ‘elementen’ volledige rijen. Het is ook mogelijk om toe te laten enkelvoudige cellen te selecteren, maar dit wordt in de praktijk slechts zelden toegepast.

Omdat er verschillende manieren zijn om aan te geven wat er geselecteerd is — je kan het element rechtstreeks aanduiden, of je kan aangeven wat de *index* is van het element — kiest JavaFX ervoor om luisteraars niet rechtstreeks bij het model te laten registreren, maar voorziet het voor dit doel in de plaats afzonderlijke *eigenschappen* die je aan het model kunt opvragen. Elke eigenschap biedt een andere kijk op het selectiemodel:

- *selectionModel.selectedItemProperty()* geeft aan wat het element is dat op dat moment is geselecteerd.
- *selectionModel.selectedIndexProperty()* geeft de index (rijnummer) aan van het geselecteerd element. (Bij boomdiagrammen is dit niet echt nuttig.)
- *selectionModel.getSelectedItems()* geeft een ('alleen lezen')-lijst terug die *alle* geselecteerde elementen bevat. Deze is van het type *ObservableList*.
- *selectionModel.getSelectedIndices()* geeft een ('alleen lezen')-lijst terug met de indices van alle geselecteerde elementen.

(De eerste twee zijn bedoeld voor de enkelvoudige selectiemodus, de laatste twee voor de meervoudige selectiemodus.)

Bij de fylogenetische boom kiezen we voor de eerste optie:

```
treeView.getSelectionModel().selectedItemProperty()
    .addListener(new SelectionListener());
```

```
public class SelectionListener implements InvalidationListener {
```

```
    public void invalidated(Observable o) {
        TreeItem<NameURIPair> treeItem =
            treeView.getSelectionModel().getSelectedItem();
        if (treeItem == null) {
            label.setGraphic(null);
            label.setText ("<leeg>");
        } else {
            NameURIPair nup = treeItem.getValue();
            if (nup.getUri() == null) {
                label.setGraphic(null);
                label.setText ("<geen foto>");
            } else {
                label.setText (null);
                label.setGraphic(new ImageView(nup.getUri()));
            }
        }
    }
}
```



## 8.4. Een eenvoudige tabel

[ De voorbeelden uit deze paragraaf vind je in *be.ugent.objprog.zip.search.* ]

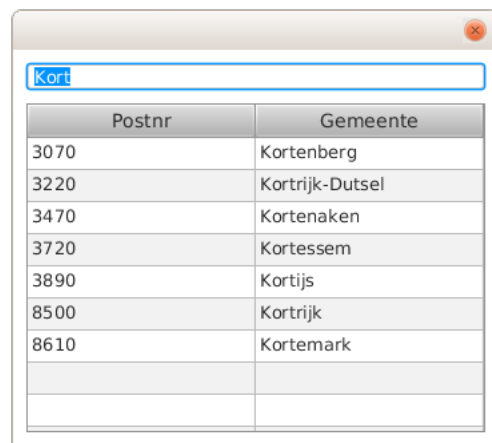
Eén van de redenen waarom de klasse *TableView* er in JavaFX zo ingewikkeld uitzielt, is omdat ze heel veel variatie toelaat. Niet alleen is het mogelijk om de gegevens in de tabel in allerlei vormen te presenteren (teksten, afbeeldingen, al dan niet aangevinkte checkboxes, ...), een tabelcomponent kan ook zodanig ingesteld worden dat de gegevens door de gebruiker ‘ter plaatse’ kunnen worden geëditeerd.

Om het voorlopig nog eenvoudig te houden, hebben we als eerste voorbeeld een tabel gekozen waarvan de gegevens in principe onveranderlijk zijn: wanneer er ook maar iets aan de inhoud van de tabel moet veranderen, zal het programma de tabel helemaal leegmaken en opnieuw opvullen.

Zoals hiernaast afgebeeld, toont onze voorbeeldtoepassing een tekstveld waar een zoekterm kan worden ingevuld, en daaronder een tabel met daarin de resultaten van de zoekopdracht. (De zoekopdracht wordt automatisch uitgevoerd wanneer men de ENTER-toets indrukt in het zoekveld.)

Je kan met het programma gemeenten opzoeken aan de hand van het begin van hun naam, of het begin van hun postnummer.

We gebruiken dezelfde databank als in paragraaf §6.1 (en volgende). De JDBC-opdrachten zijn weggelaten in een eenvoudige klasse *ZipSearcher* met daarin de volgende methode:



Postnr	Gemeente
3070	Kortenber
3220	Kortrijk-Dutse
3470	Kortenaen
3720	Kortesse
3890	Kortijs
8500	Kortrijk
8610	Kortemark

```
public Iterable<ZipInfo> find (String prefix);
```

Als argument geef je een zoekterm op, als resultaat krijg je een lijst van *ZipInfo*-objecten. *ZipInfo* is een heel eenvoudige (zelfgemaakte) klasse met slechts twee velden (en overeenkomstige getters en setters) — *zip* en *name*. De inhoud van een dergelijk object komt precies overeen met wat we in één rij van onze tabel wensen te tonen. (Dit blijkt over het algemeen de meest eenvoudige manier van werken<sup>8</sup>.)

De JavaFX-component waarmee je een tabel voorstelt, heet *TableView* — of in dit geval voluit *TableView<ZipInfo>*. Zoals bij een *ListView*, *ComboBox* en *TreeView*, moet je

<sup>8</sup>In dit voorbeeld is *ZipInfo* een klasse met getters en setters en geen record-klasse. Tabellen in JavaFX kunnen (nog) niet goed overweg met records — zie §8.6.

altijd een typeparameter meegeven die aangeeft met welke klasse de getoonde inhoud overeenkomt. Bij een tabel stelt een object van die klasse steeds een volledige rij voor.

Omdat JavaFX bij tabellen het MVC-patroon aanhoudt, waarvan *TableView* een view is, zal er ook ergens een *model* te pas komen. Dit model is van het type *ObservableList*, of in ons geval, meer bepaald *ObservableList<ZipInfo>*. Een *TableView* registreert zichzelf als waarnemer bij de *ObservableList* en wordt op die manier op de hoogte gehouden van veranderingen die aan de lijst worden aangebracht: wanneer er elementen aan de lijst worden toegevoegd, wanneer er worden verwijderd, of wanneer het ene element door een ander wordt vervangen.

Let echter op: de *ObservableList* kan niet weten wanneer er iets verandert aan de *inhoud* van de objecten die het bevat! Wanneer we bijvoorbeeld *setName* zouden oproepen voor een *ZipInfo*-object dat zich al in de lijst bevindt, dan zal het model dit niet doorhebben en zal je ook geen veranderingen zien aan de tabel. Er bestaan manieren om dit op te lossen maar voor dit eenvoudig voorbeeld vormt dit geen probleem.

Hoewel *ObservableList* een interface is, is het in de praktijk bijna nooit nodig om deze interface zelf te implementeren. We gebruiken doorgaans een standaardimplementatie die door JavaFX wordt aangeboden. Dergelijke lijsten kan je aanmaken met behulp van bepaalde klassenmethoden uit de klasse *FXCollections*. In ons voorbeeld doen we dit op de volgende manier:

```
ObservableList<ZipInfo> model = FXCollections.observableArrayList();
```

Daarna vullen we de lijst op met de gegevens die we hebben opgehaald uit de databank en tenslotte stellen we de lijst in als model bij de *TableView*, met behulp van de methode *setItems* van die klasse. Hieronder drukken we de volledige broncode af van de methode *doSearch* die de actiegebeurtenis afhandelt als gevolg van het indrukken van de ENTER-toets in het zoekveld.

```
public TableView<ZipInfo> tableView; // aangemaakt door de FXMLLoader

public void doSearch() {
    ObservableList<ZipInfo> model = FXCollections.observableArrayList();
    for (ZipInfo zipInfo : zipSearcher.find(searchField.getText())) {
        model.add(zipInfo);
    }
    tableView.setItems(model);
}
```

Tot zover het modelgedeelte van onze toepassing.

Ook het viewgedeelte verdient een bijkomend woordje uitleg. Er zijn naast *TableView* namelijk ook nog andere klassen die een rol spelen bij het produceren van de uiteindelijke

tabel. Als je met de *scene builder* een tabel toevoegt aan de scène, dan heeft die tabel geen inhoud. Je moet daartoe immers eerst *kolommen* toevoegen aan de tabel.

Kolommen zijn volwaardige GUI-componenten van het type *TableColumn* en worden zelf weer onderverdeeld in cellen — van het type *TableCell*. (Beide types zijn ‘generisch’ en hebben typeparameters — zie verder.)

Een kolom maakt zelf zijn cellen aan, dus in eerste instantie hoeven we ons van de klasse *TableCell* niet zoveel aan te trekken. Bij meer ingewikkelde tabellen kan je zelf een cell factory schrijven, zoals we eerder gedaan hebben bij de combobox van paragraaf §8.1.

De inhoud die in een bepaalde cel wordt getoond, is van een bepaald type. Meestal is dit *String*, maar ook andere types (*Number*, *Date*, ...) zijn toegelaten. Het type is echter steeds hetzelfde voor alle cellen van een vaste kolom.

In het lopend voorbeeld gebruiken we het type *String* voor al onze cellen, en daarom is het volledige type van onze kolommen *TableColumn<ZipInfo, String>*. (En de cellen zelf zijn van het type *TableCell<ZipInfo, String>*.) De kolommen hebben een `fx:id` gekregen in het FXML-bestand en worden opgeslagen in de volgende variabelen:

```
public TableColumn<ZipInfo, String> zipColumn;
```

```
public TableColumn<ZipInfo, String> nameColumn;
```

Nu we het model en de view hebben gebouwd, moeten we enkel nog beide met elkaar verbinden: JavaFX moet weten hoe een *ZipInfo*-object wordt opgedeeld in inhoud voor de verschillende cellen. Hiervoor heb je een zogenaamde cell *value* factory nodig — niet te verwarren met een cell factory.

Je moet bij elke kolom een dergelijke cell value factory registreren. Deze factory bepaalt hoe je vanuit een object uit het datamodel (een *ZipInfo*) de corresponderende inhoud (een *String*) bekomt voor de overeenkomstige tabelcel in de betreffende kolom.

Bij ons voorbeeld hebben we twee kolommen en zijn er dus twee cell value factories nodig:

- Voor de eerste kolom, een cell value factory die van een gegeven *ZipInfo*-object het *zip*-veld teruggeeft — de waarde van de *getZip*-methode.
- Voor de tweede kolom, een cell value factory die van een gegeven *ZipInfo*-object het *name*-veld teruggeeft — de waarde van de *getName*-methode.

Het zal je niet verwonderen dat er voor dit eenvoudige geval waarin er gewoon een gepaste *getter* moet worden opgeroepen, een standaardfactory bestaat — een *PropertyValueFactory*. De constructor van deze factory heeft één argument: de *naam* van de eigenschap

waarvoor de getter wordt opgeroepen<sup>9</sup>.

Dit is hoe we die cell value factories uiteindelijk registreren:

```
zipColumn.setCellValueFactory( new PropertyValueFactory<>("zip"));
nameColumn.setCellValueFactory( new PropertyValueFactory<>("name"));
```

*PropertyValueFactory* heeft twee typeparameters, één voor het type van de rij en één voor het type van de cel, net zoals *TableView*.

## 8.5. Cell factories voor tabellen

[ Voorbeelden uit deze paragraaf vind je o.a. in *be.ugent.objprog.students*, *be.ugent.objprog.zip.search*, *be.ugent.objprog.tracker.table* en *be.ugent.objprog.proglangs*. ]

We hebben al aangegeven dat een tabel-, lijst- of boomcomponent is samengesteld uit een aantal *cellen*. Bij standaardcomponenten gedraagt een dergelijke cel zich als een tekstlabel. Bij een *editeerbare* cel verandert het label in een tekstveld zodra je erop dubbelklikt. Er kunnen verschillende redenen zijn om van dit gedrag te willen afwijken. Enkele voorbeelden:

- Je wil een kolom met getallen rechts uitlijnen in plaats van links.
- Je wil negatieve getalwaarden in het rood afbeelden.
- Je wenst niet alleen tekst af te beelden, maar ook een kleine afbeelding. Je ziet dit vaak in een boomdiagram en ook bij het combo box-voorbeeld hebben we iets gelijkaardigs gedaan.
- Je wil een contextmenu hechten aan een cel.
- Je wil reageren wanneer er op een cel wordt gedubbelklikt (bij een niet-editeerbare cel).
- Je wil een editeerbare logische waarde voorstellen als een checkbox in plaats van een label of tekstveld.
- Bij een (editeerbare) cel die maar enkele mogelijke waarden toelaat, wil je liever een *choice box* gebruiken.

---

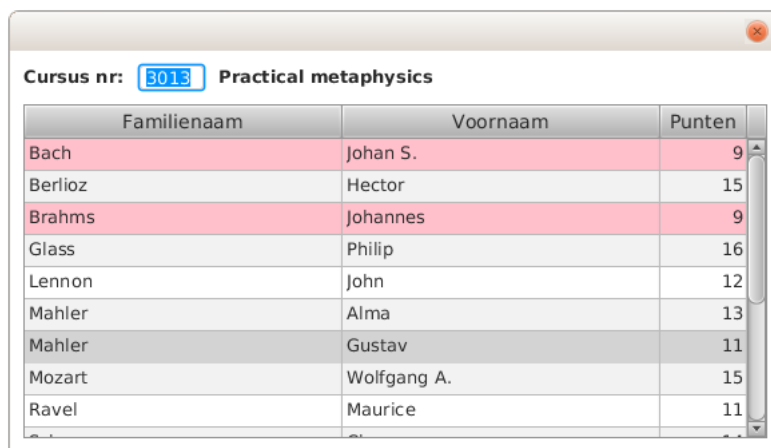
<sup>9</sup>Java maakt hierbij gebruik van *reflectie*: het is pas tijdens het uitvoeren van het programma dat Java kijkt of er een getter bestaat voor de opgegeven naam. Als je dus bijvoorbeeld een tikfout maakt in het argument van de constructor, dan zal de compiler die niet opmerken.

Een cell factory biedt hier een oplossing. (Bij tabellen hecht je een cell factory aan de betreffende *kolom*.)

Stel bijvoorbeeld dat we de postnummerkolommen in ons voorbeeld rechts willen uitlijnen. Hiervoor hoeven we enkel maar cellen aan te maken waarvoor de *alignment*-eigenschap correct is ingesteld. Het standaardceltype voor een kolom in een tabel heet *TextFieldTableCell*<sup>10</sup>. We schrijven dus:

```
zipColumn.setCellFactory(  
    column -> {  
        TableCell<ZipInfo, String> cell = new TextFieldTableCell<>();  
        cell.setAlignment(Pos.CENTER_RIGHT);  
        return cell;  
    }  
);
```

Een cell factory voor een kolom moet een nieuwe cel aanmaken van het type *TableCell* en krijgt als parameter *column*. De toepassing hieronder gebruikt een cell factory die iets ingewikkelder is.



Familiennaam	Voornaam	Punten
Bach	Johan S.	9
Berlioz	Hector	15
Brahms	Johannes	9
Glass	Philip	16
Lennon	John	12
Mahler	Alma	13
Mahler	Gustav	11
Mozart	Wolfgang A.	15
Ravel	Maurice	11

Deze tabel toont een lijst van punten van studenten. Wanneer een student niet geslaagd is (minder dan 10 punten heeft behaald) dan wordt de corresponderende rij afgebeeld in een andere kleur.

Hoewel het effect hier telkens een ganse rij betreft, hebben we dit toch bekomen met een aangepaste cell factory voor één enkele kolom. Elke cel van een tabel bevat immers een verwijzing naar de *TableRow* waarvan hij deel uitmaakt.

<sup>10</sup>De *TextField* in de naam verwijst naar het gedrag dat een dergelijke cel vertoont wanneer de cel wordt geëditeerd. Maar ook voor niet-editeerbare cellen moet je *TextFieldTableCell* gebruiken en niet gewoon *TableCell*, tenzij je zelf *updateItem* implementeert.

Dit keer definiëren we een nieuwe klasse *MarksCell* met een aangepaste *updateItem*-methode die de puntenwaarde controleert die als parameter wordt doorgegeven. Als die kleiner is dan 10 dan hechten we de CSS-klasse *failed* aan de corresponderende rij, anders verwijderen we ze terug.

```
public class MarksCell extends TableCell<MarksInfo, Integer> {

    private static final String CSS_CLASS_FAILED = "failed";

    public MarksCell() {
        setAlignment(Pos.BASELINE_RIGHT);
    }

    @Override
    protected void updateItem(Integer value, boolean empty) {
        super.updateItem(value, empty);
        ObservableList<String> rowStyles = getTableRow().getStyleClass();

        rowStyles.removeAll(CSS_CLASS_FAILED);

        if (empty) {
            setText(null);
        } else {
            setText(value.toString());
            if (value < 10) {
                rowStyles.add(CSS_CLASS_FAILED);
            }
        }
    }
}
```

Merk op dat door een samenloop van omstandigheden dezelfde cel meerdere keren na elkaar een waarde kan krijgen die kleiner is dan 10. We moeten dus opletten dat de niet per ongeluk meerdere keren de CSS-klasse *failed* wordt toegewezen. Vandaar de *removeAll* in het midden van *updateItem*.

Het corresponderende blok in het CSS-bestand ziet er dan zo uit<sup>11</sup>:

```
.table-row-cell.failed {
    -fx-control-inner-background: pink
}
```

---

<sup>11</sup>Intern wordt een *TableRow* voorgesteld als een bijzondere cel. Vandaar ‘.table-row-cell’.

In een derde toepassing wensen we dat er een dialoogvenster verschijnt wanneer we op een rij dubbelklikken. (Wat het dialoogvenster precies doet, is hier niet zo relevant — het kan bijvoorbeeld toelaten om het corresponderende item te editeren.)

We kunnen dit oplossen door aan *elke* kolom van de tabel een cell factory te hechten waarvan de cellen op muisgebeurtenissen reageren. Hieronder geven we de broncode van een dergelijke cel:

```
public class DoubleClickableTableCell<S, T>
    extends TextFieldTableCell<S, T>
    implements EventHandler<MouseEvent> {

    private EventHandler<ActionEvent> actionEventHandler;

    public DoubleClickableTableCell(EventHandler<ActionEvent> handler) {
        this.actionEventHandler = handler;
        setOnMouseClicked(this);
    }

    @Override
    public void handle(MouseEvent t) {
        if (t.getClickCount() > 1) {
            ActionEvent ae = new ActionEvent(this, null);
            actionEventHandler.handle(ae);
        }
    }
}
```

De parameter *handler* die wordt meegegeven aan de constructor bevat het object dat de dubbelklik moet verwerken. Deze gebeurtenisverwerker krijgt de cel mee als bron van de gebeurtenis en kan op die manier te weten komen op welke rij er werd gedubbelklikt:

```
public void handle(ActionEvent t) {
    TableCell tc = (TableCell) t.getSource();
    int rijIndex = tc.getIndex();
    // doe iets met de rij op index rijIndex
    ...
}
```

Een andere manier om een gelijkaardig effect te bekomen, is door het dubbelklikgedrag te hechten aan de tabelrij in plaats van aan elke cel afzonderlijk. Hiertoe registreren we een nieuwe *table row factory* bij de tabel.

```
tableView.setRowFactory(table -> new DoubleClickableTableRow<>(
    handler )
);
```

Een row factory is een callback die voor een gegeven *TableView* een gepaste *TableRow* retourneert. Hier genereren we een tabelrij van het volgende type:

```
public class DoubleClickableTableRow<S>
    extends TableRow<S>
    implements EventHandler<MouseEvent> {

    private EventHandler<ActionEvent> actionEventHandler;

    public DoubleClickableTableRow(EventHandler<ActionEvent> handler){
        this.actionEventHandler = handler;
        setOnMouseClicked(this);
    }

    @Override
    public void handle(MouseEvent t) {
        if (t.getClickCount() > 1) {
            ActionEvent ae = new ActionEvent(this, null);
            actionEventHandler.handle(ae);
        }
    }
}
```

Deze implementatie is bijna identiek aan die van *DoubleClickableTableCell*. Merk wel op dat de actiegebeurtenissen nu als bron de volledige rij hebben, en niet één cel. Maar aan een rij kan je evengoed een index opvragen, dus in de praktijk maakt dit niet veel uit.

Door lambda's te gebruiken, hoef je geen nieuwe *TableRow*-klasse te schrijven:

```
tableView.setRowFactory(table -> {
    TableRow<TimeSlot> row = new TableRow<>();
    row.setOnMouseClicked(
        ev -> {
            if (ev.getClickCount() > 1) {
                handle(row.getIndex());
            }
        }
    );
    return row;
});
```



Tenslotte illustreren we nog hoe je een cell factory kunt gebruiken om de inhoud van een tekstveld iets anders te laten zijn dan wat er door *toString* wordt geproduceerd. Onderstaande cel beeldt een datum en tijd (een *LocalDateTime*-object) af in een vooraf vastgelegd formaat:

```
public class LocalDateTimeTableCell<S>
    extends TableCell<S,LocalDateTime>
{
    private static final DateTimeFormatter DATE_TIME_FORMATTER
        = DateTimeFormatter.ofPattern("dd-MM-yyyy hh:mm");

    public void updateItem(LocalDateTime date, boolean empty) {
        super.updateItem(date, empty);
        if (empty) {
            setText(null);
        } else {
            setText(DATE_TIME_FORMATTER.format(date));
        }
    }
}
```

Als alternatief kan je, net zoals bij de laatste versie van de combo box (§8.1), een aangepaste *string converter* gebruiken. De klasse *TextFieldTableCell* heeft namelijk ook een constructor die een dergelijke converter als argument neemt:

```
cellFactory =
    table -> new TextFieldTableCell<>(
        new LocalDateTimeStringConverter(
            DATE_TIME_FORMATTER, DATE_TIME_FORMATTER)
    );
beginColumn.setCellFactory(cellFactory);
endColumn.setCellFactory(cellFactory);
```

## 8.6. Cell value factories

In veel gevallen zal je voor de kolommen een *PropertyValueFactory* als cell value factory kunnen gebruiken. Zoals de naam reeds aangeeft, lukt dit echter enkel maar wanneer de waarde overeenkomt met een *eigenschap* van het rij-object, en dat moet op zijn minst een booneigenschap zijn.

Vaak gebruik je echter records voor de rijen, bijvoorbeeld de klasse *Preference* uit het programmeertalenvoorbeeld van paragraaf §6.3:

```
public record Preference (String name, boolean checked) {
}
```

In dit geval kan je geen *PropertyValueFactory* gebruiken maar moet je zelf een cell value factory schrijven. In principe is dit niet zo moeilijk. We hebben eerder al aangegeven dat een dergelijke cell value factory uit een rij-object (van het model van de tabel) een waarde moet distilleren die door een cel in de betreffende kolom kan worden afgebeeld. Je verwacht dus iets te kunnen schrijven als

```
nameColumn.setCellValueFactory( row -> row.name() )
```

Dit werkt echter niet, want de lambda bezit niet het juiste type.

Om precies te zijn, maakt een cell value factory niet de waarde zelf aan, maar verpakt die in een *ObservableValue*. De tabel registreert zich dan als luisteraar bij deze *ObservableValue* zodat ook toekomstige veranderingen in de waarde onmiddellijk naar de corresponderende cel kunnen worden doorgegeven.

Bovendien vertrekt een cell value factory niet van het rij-object, maar van een object van het type *CellDataFeatures* dat een referentie naar het rij-object bevat die je kan opvragen met *getValue*.

Voor een kolom *nameColumn* van het type *TableColumn<Preference,Boolean>* moeten we dus schrijven

```
nameColumn.setCellValueFactory(
    features -> new SimpleStringProperty(features.getValue().name())
);
```

In plaats van *SimpleStringProperty* kan je ook *SimpleObjectProperty<String>* gebruiken. Voor kolommen met gehele getallen (en analoog, voor kommagetallen) is er een bijkomende complicatie: hier kan je enkel *SimpleObjectProperty<Integer>* gebruiken, het alternatief *SimpleIntegerProperty* is van het type *ObservableValue<Number>* en niet *ObservableValue<Integer>* en werkt dus enkel voor kolommen van het type *Number*.

Merk op dat deze implementatie van een cell factory wel aan de correcte interface voldoet, maar niet aan het corresponderende *contract*: wanneer een *Preference*-object een nieuwe naam krijgt, zal deze verandering niet zichtbaar worden in de tabel. De tabel observeert wel degelijk de *SimpleStringProperty* die we hebben aangemaakt, maar deze eigenschap is niet met het *Preference*-object ‘gesynchroniseerd’ — ze werd geïnitialiseerd met de toenmalige waarde van *name()* maar is er niet rechtstreeks mee verbonden.

Voor een toepassing waarin de tabelwaarden niet veranderen terwijl ze worden getoond, is dergelijk gedrag wellicht aanvaardbaar. Willen we echter alles volgens de regels implementeren, dan kunnen we niet zomaar *SimpleStringProperty* gebruiken, maar moeten

we zelf een nieuwe klasse maken die *ObservableValue* implementeert. Dit kan echter alleen maar als we ook een setter introduceren die signaleert dat er iets aan die naam is veranderd.

In de praktijk zal men dit vaak oplossen door bij de rij-objecten — m.a.w., de elementen van de *ObservableList* die het model van de tabel vormt — geen gewone velden te definiëren, maar velden die JavaFX-eigenschappen zijn.

Dus, in plaats van een tabel te maken die een lijst van *Preference*-objecten bevat, gebruiken we een lijst van *Row*-objecten met de volgende definitie:

```
public class Row {  
  
    private StringProperty name = new SimpleStringProperty();  
  
    public StringProperty nameProperty() {  
        return name;  
    }  
  
    public String getName() {  
        return name.get();  
    }  
  
    public void setName(String name) {  
        this.name.set(name);  
    }  
  
}
```

Let op dat je de naamgevingsafspraken volgt voor JavaFX-eigenschappen (§5.7).

## 8.7. Editeerbare tabellen

[ Voorbeelden uit deze paragraaf vind je in *be.ugent.objprog.students*. ]

We zijn nu eindelijk zover dat we ons kunnen wagen aan het implementeren van een *editeerbare tabel*: een tabel waarvan de inhoud ‘ter plaatse’ kan worden aangepast.

Als voorbeeld breiden we de puntentoepassing van bladzijde 173 uit zodat je nu ook punten van studenten kunt wijzigen (zie afbeelding op de volgende bladzijde).



Het programma werkt als volgt: wanneer je op een puntencel dubbelklikt, dan verandert deze in een tekstveld waarin je de punten kan aanpassen. Druk je op de ENTER-toets, dan verandert de cel terug in een label en kan je verder gaan met een andere kolom.

Heb je alle wijzigingen in de tabel aangebracht, dan druk je op de 'Wijzigingen opslaan'-knop onderaan. Hierdoor komen de nieuwe punten ook in de databank terecht.

### 8.7.1. Cell factories voor editeerbare tabellen

Om een *TableView* editeerbaar te maken, moet je eerst haar *editable*-eigenschap instellen.

```
tableView.setEditable(true);
```

Ook de afzonderlijke kolommen hebben een *editable*-eigenschap, maar die is standaard op **true** ingesteld.

```
lastNameColumn.setEditable(false);
firstNameColumn.setEditable(false);
marksColumn.setEditable(true); // hoeft niet
```

Het grote werk gebeurt echter door de tabelcellen. Een standaardtabelcel is *niet* editeerbaar, en daarom moeten we een ander type kiezen: de *TextFieldTableCell* is hiervoor een uitstekende kandidaat.

Een cell factory zou er dus zo kunnen uitzien:

```
column -> new TextFieldTableCell<>();
```

In de praktijk zal deze implementatie echter niet werken. De tabelcel werkt immers met strings (als tekst voor zowel het label als het tekstveld) terwijl de gegevens die we aan de cel doorgegeven van het type *Integer* zijn. We hebben hier dus een string converter nodig.

```
column -> new TextFieldTableCell<>(new IntegerStringConverter());
```

Meer nog, ook wanneer de kolom strings bevat, moet je nog altijd een string converter meegeven als argument van de constructor van *TextFieldTableCell*, namelijk een *DefaultStringConverter*. Zonder converter kan een *TextFieldTableCell* alleen gebruikt worden in een niet-editeerbare kolom.

## 8.7.2. JavaFX-eigenschappen in het rij-object

Jammer genoeg is dit nog maar het begin van het verhaal. Als je bovenstaande code uitprobeert, merk je dat dubbelklikken op een puntencel inderdaad toelaat om die cel te editeren, en hoewel de inhoud van die cel verandert nadat je op ENTER hebt gedrukt, zijn de wijzigingen niet permanent. Probeer maar eens de tabel naar boven en beneden te scrollen zodat de betreffende cel eerst verdwijnt en dan terug verschijnt. Je zal zien dat ze opnieuw haar oude waarde heeft gekregen.

Eigenlijk valt dit niet te verwonderen. Een edit-actie moet uiteindelijk als effect hebben dat er iets aan het corresponderende rij-object verandert, en we mogen van JavaFX niet verwachten dat hij dit zonder enige bijkomende instructie zomaar voor ons zal klaarspelen.

We zullen twee manieren bespreken om dit te bewerkstelligen. De eerste is het gemakkelijkste om te programmeren, maar is minder flexibel en leidt vaak tot code die ingewikkelder is dan nodig. Deze methode maakt gebruik van JavaFX-eigenschappen en is heel populair.

Bij een standaardtabel vraagt JavaFX na een edit-actie op een cel de celwaarde van die cel op aan de cell value factory. Blijkt die waarde de interface *WriteableValue* te implementeren (en niet alleen *ObservableValue* zoals het contract van een cell value factory voorschrijft) dan wordt de corresponderende *setValue* opgeroepen met de nieuwe waarde van de cel als argument.

Een oplossing voor ons probleem, zoals reeds aangehaald op het einde van paragraaf §8.6, is om de punten van de studenten in een rij-object niet gewoon als *int* voor te stellen, maar als *IntegerProperty*. Als de cell value factory deze JavaFX-eigenschap als waarde teruggeeft, dan zal ze vanzelf worden aangepast wanneer de corresponderende cel wordt geëditeerd.

In ons voorbeeld betekent dit dat we voor de rij-objecten een klasse *MarksRow* zullen gebruiken met de volgende velden:

- Een string *lastName* met overeenkomstige getters en setters.
- Een string *firstName* met overeenkomstige getters en setters.
- Een JavaFX-eigenschap *marks* van het type *IntegerProperty* met bijbehorende getter, setter en *property*-methode.
- Wellicht nog één of meer andere velden die we nodig hebben om de inhoud van het rij-object van en naar de databank over te brengen. (Een primaire sleutel, bijvoorbeeld.)

De kolommen van de tabel worden nu als volgt ingesteld:

```
lastNameColumn.setCellValueFactory(  
    new PropertyValueFactory<>("lastName"));  
lastNameColumn.setEditable(false);  
firstNameColumn.setCellValueFactory(  
    new PropertyValueFactory<>("firstName"));  
firstNameColumn.setEditable(false);  
  
marksColumn.setCellValueFactory(  
    new PropertyValueFactory<>("marks"));  
marksColumn.setCellFactory(  
    column -> new TextFieldTableCell<>(new IntegerStringConverter())  
);  
marksColumn.setEditable(true);
```

Merk op dat we voor de puntenkolom nog steeds dezelfde *PropertyValueFactory* gebruiken als voorheen. In plaats van het veld *marks* in een *ReadOnlyProperty* te pakken, zoals bij de andere kolommen, geeft de factory het *marks*-object nu rechtstreeks terug. Dit is waarvoor de klasse *PropertyValueFactory* in de eerste plaats eigenlijk is bedoeld.

Met deze aanpassingen hebben we eindelijk een werkend programma.

### 8.7.3. De ‘edit commit’-gebeurtenis

Je kan ook een editeerbare tabel maken zonder JavaFX-eigenschappen in het rij-object te hoeven introduceren. Dit doe je door zogenaamde *cell edit*-gebeurtenissen op te vangen.

(Ook al is deze methode blijkbaar minder populair, toch blijkt ze vaak een betere keuze te zijn dan die uit de vorige paragraaf<sup>12</sup>.)

Een tabelkolom genereert drie soorten gebeurtenissen:

- Een *'edit start'*-gebeurtenis zodra de gebruiker een cel *begint* te editeren, m.a.w., nadat hij op de cel heeft dubbelgeklikt.
- Een *'edit cancel'*-gebeurtenis wanneer de gebruiker het editeren *afbreekt* — door op ESC-toets te drukken.
- Een *'edit commit'*-gebeurtenis wanneer de gebruiker het editeren *afrontt* door op ENTER te drukken.

Het is deze laatste gebeurtenis die we zullen opvangen, op de volgende manier:

```
private class EditCommitHandler
    implements EventHandler<CellEditEvent<MarksInfo,Integer>> {

    public void handle(CellEditEvent<MarksInfo, Integer> t) {
        MarksInfo row = t.getRowValue();
        row.setMarks(t.getNewValue());
    }

}

...

marksColumn.setOnEditCommit(new EditCommitHandler ());
```

Het *CellEditEvent*-object bevat informatie over de cel die op dat moment geëditeerd wordt — het corresponderend rij-object, de index van de rij, de oude waarde en de nieuwe waarde van de cel, enz. In onze implementatie vragen we de nieuwe waarde van de cel op en passen daarmee rechtstreeks de puntenveld aan van het rij-object.

We kunnen dit nog behoorlijk afkorten met behulp van lambda's:

```
marksColumn.setOnEditCommit(
    t -> t.getRowValue().setMarks(t.getNewValue());
```

Voor elk van de drie edit-gebeurtenissen bestaat er een overeenkomstige methode in de klasse *Cell*:

---

<sup>12</sup>Helaas werkt deze techniek niet voor check box-cellen.

```

public void startEdit ();

public void cancelEdit ();

public void commitEdit (T newValue);

```

In plaats van de ‘edit commit’-gebeurtenis te overschrijven, kan je dus ook de *commitEdit*-methode overschrijven van de cel die door de cell factory wordt gegenereerd:

```

public class MarksCell extends TextFieldTableCell<MarksInfo,Integer> {
    ...
    @Override
    public void commitEdit(Integer newValue) {
        super.commitEdit(newValue); // niet vergeten!
        getTableView().getItems().get ( getIndex() ).setMarks (newValue);
    }
}

```

(Vanuit het oogpunt van goed software-ontwerp is dit een minder gelukkige keuze. De cel grijpt nu te veel rechtstreeks in in het model — wat in tegenstrijd is met haar rol als *view*. Bovendien is de herbruikbaarheid van een dergelijke cel heel klein.)

## 8.7.4. Verdere uitbreidingen

[ De uiteindelijke versie van het programma staat in *be.ugent.objprog.students.edit*. ]

Familiennaam	Voornaam	Punten
<i>Bach</i>	<i>Johan S.</i>	<i>6</i>
<i>Berlioz</i>	<i>Hector</i>	<i>9</i>
Brahms	Johannes	13
<i>Glass</i>	<i>Philip</i>	<i>12</i>
Lennon	John	12
Mahler	Alma	13
Mahler	Gustav	9
Mozart	Wolfgang A.	20

Opslaan Annuleren

Tot slot doen we nog enkele aanpassingen om het programma gebruiksvriendelijker te maken:



- Wanneer je op de ‘Opslaan’-knop drukt, hoeven alleen die rijen naar de databank te worden doorgestuurd waarvan de punten effectief veranderd zijn.
- Beter nog: we kunnen de gebruiker ook laten zien welke rijen er reeds zijn aangepast en welke niet, bijvoorbeeld door de betreffende rijen in een andere kleur af te beelden of een ander lettertype ervoor te gebruiken.
- De ‘Opslaan’-knop mag gedeactiveerd blijven zolang er nog geen wijzigingen zijn gemaakt.
- Zodra er echter wijzigingen zijn gemaakt, kan de gebruiker niet langer punten opvragen van een andere cursus. We voeren enkele bijkomende knoppen in: een ‘Zoek’-knop om een nieuwe opzoeking te doen en een ‘Annuleren’-knop om de wijzigingen die reeds gemaakt zijn ongedaan te maken. De ‘Zoek’-knop wordt gedeactiveerd zolang er wijzigingen zijn aan de huidige tabel die nog niet naar de databank werden doorgegeven — of geannuleerd.

Om dit allemaal mogelijk te maken, moeten we enkele aanpassingen doen aan het achterliggende model voor de tabel: elk rij-object in dit model moet niet alleen weten wat de punten zijn die in de tabel staan afgebeeld, maar ook wat de originele punten waren waarmee de rij werd geïntialiseerd. (Deze laatste zullen we opslaan in een bijkomend veld van het rij-object met de naam *originalMarks*.)

Daarnaast houden we ook een afzonderlijk model bij dat aangeeft *hoeveel* rijen een waarde hebben die verschilt van de oorspronkelijke. De drie knoppen (of de partnerklasse) kunnen naar dit model luisteren om te weten of ze al dan niet geactiveerd moeten zijn. Hieronder gebruiken we als model een JavaFX-eigenschap met de naam *numberOfChanges*.

Hoe de wijzigingen in de rij-objecten naar het *numberOfChanges*-model worden gecommuniceerd, hangt af van de techniek die we gebruiken om de editeerbare tabel te realiseren. We bespreken hier alleen het geval waarin we gebruik maken van JavaFX-eigenschappen in de rij-objecten — zoals in paragraaf §8.7.2.

Als eerste stap voegen we een tweede JavaFX-eigenschap toe aan elke rij, een Booleaanse eigenschap *changed*. Deze eigenschap geeft aan of de punten die in het rij-object zijn opgeslagen al dan niet overeenkomen met hun originele waarde. Je kan dit bijvoorbeeld bekomen door elke rij te laten luisteren naar veranderingen in zijn *marks*-eigenschap.

Een alternatief bestaat erin om ook van *originalMarks* een JavaFX-eigenschap te maken, en dan *changed* te *binden* aan de andere twee eigenschappen (zie paragraaf §5.9).

Dit kan met één enkele opdracht:

```
changed.bind( Bindings.notEqual(marks, originalMarks) );
```

Een bijkomend voordeel van deze strategie is dat we de *changed*-eigenschap nu gemakkelijk kunnen koppelen aan een vierde kolom — zonder inhoud en zeer smal zoals je op de schermafdruck kunt zien. Aan deze kolom hechten we dan een cell factory van het zelfde genre als deze die we hebben gebruikt om een rij al dan niet rood te kleuren (maar nu met een andere CSS-klasse).

Om *numberOfChanges* up-to-date te houden, moet je bij de *changed*-eigenschap van elke rij een luisteraar registreren die één optelt of aftrekt van *numberOfChanges* afhankelijk van of *changed* gelijk is aan **true** of **false**. Dit doen we op het moment dat het tabelmodel wordt ingevuld.

```
private InvalidationListener changedPropertyListener =
    property -> {
        if (((BooleanProperty) property).get()) {
            numberOfChanges.set(numberOfChanges.get() + 1);
        } else {
            numberOfChanges.set(numberOfChanges.get() - 1);
        }
    }
};

...
for (Row row : tableModel) {
    row.changedProperty().addListener(changedPropertyListener);
}
```

Dat we een blanco kolom hebben toegevoegd aan onze tabel om de CSS-stijl van de tabelrij aan te passen, was een gemakkelijksoplossing — het kan ook zonder. We kunnen het aanpassen van de CSS-stijl ook overlaten aan de tabelrij zelf, door een gepaste table row factory te voorzien.

De methode *updateItem* van *TableRow* wordt echter alleen maar opgeroepen wanneer het rij-object zelf verandert, en niet wanneer de waarden van zijn velden worden aangepast. We dienen de tabelrij dus naar veranderingen in het rij-object te laten luisteren.

Herinner je echter dat dezelfde *TableRow* niet steeds met hetzelfde rij-object blijft geassocieerd. We moeten dus bijkomende code voorzien om luisteraars van de oude rij-objecten los te koppelen vooraleer ze te registreren bij de nieuwe:

```
public class MarksTableRow extends TableRow<Row>
    implements InvalidationListener {

    private Row lastRow;
```

```

public void invalidated(Observable o) {
    updateStyles();
}

protected void updateItem(Row row, boolean empty) {
    super.updateItem(row, empty);

    if (lastRow != row) {
        if (lastRow != null) {
            lastRow.changedProperty().removeListener(this);
            lastRow.marksProperty().removeListener(this);
        }
        if (empty) {
            lastRow = null;
        } else {
            lastRow = row;
            row.changedProperty().addListener(this);
            row.marksProperty().addListener(this);
        }
        updateStyles();
    }
}
...
}

```

De CSS-stijlen worden aangepast met de volgende methode:

```

private static final String CSS_CLASS_CHANGED = "changed";
private static final String CSS_CLASS_FAILED = "failed";

private void updateStyles () {
    ObservableList<String> rowStyles = getStyleClass();

    rowStyles.removeAll(CSS_CLASS_CHANGED, CSS_CLASS_FAILED);

    if (lastRow != null) {
        if (lastRow.isChanged()) {
            rowStyles.add (CSS_CLASS_CHANGED);
        }
        if (lastRow.getMarks() < 10) {
            rowStyles.add (CSS_CLASS_FAILED);
        }
    }
}

```



## 9. Databanktoegang abstraheren

Bij een softwaretoepassing van enige grootte zal men ervoor kiezen het JDBC-gedeelte zoveel mogelijk te scheiden van de rest van het programma. In de praktijk is het zelfs vaak zo dat dit gedeelte door een afzonderlijke programmeur wordt ontworpen en geïmplementeerd — niet elke Java-programmeur is immers ook een SQL-expert.

In dit hoofdstuk illustreren we hoe je een dergelijke scheiding kunt verwezenlijken met behulp van zogenaamde *data access objects* (DAO's). Deze techniek introduceert een bijkomende abstractie. Die zorgt er enerzijds voor dat de Java-programmeur niet alle details van de databanktoegang hoeft te kennen, en laat anderzijds toe om het databankgedeelte onafhankelijk van de rest van de toepassing te testen, te optimaliseren en zelfs te veranderen.

Je vindt op het Internet heel wat verschillende definities terug voor de term *data access object*, en er bestaan inderdaad heel wat varianten. Combineer onze voorbeelden dus niet zomaar met wat je elders aantreft zonder goed te begrijpen wat je doet. Wij hebben bepaalde keuzes gemaakt wat betreft de structuur van bepaalde klassen, de signatuur van bepaalde methodes, enz., maar ons ontwerp is slechts één van de vele gangbare mogelijkheden.

### 9.1. Data access objects

Als voorbeeld baseren we ons op de 'contacten'databank die geschetst wordt in appendix A. Deze bevat contactgegevens van personen: e-mailadressen, telefoon- en faxnummers.

Een eerste stap bestaat er in om voor elk van deze basisgegevens een afzonderlijke (record-)klasse te ontwerpen: de klassen *Person*, *Contact* en *ContactType*. Objecten van deze klassen komen rechtstreeks overeen met de rijen uit de drie tabellen uit de databank: *personen*, *contactgegevens* en *contactcodes*<sup>1</sup>.

De klasse *Person* heeft drie velden:

---

<sup>1</sup>We hebben met opzet de namen van deze klassen niet letterlijk dezelfde gekozen als die van de overeenkomstige tabellen. Dit is al een eerste vorm van abstractie: wie onze klassen gebruikt, hoeft de details van de onderliggende databank niet te kennen.

```
public record Person(int id, String name, String firstName) {  
}
```

Deze drie velden representeren de drie kolommen *id*, *familienaam* en *voornaam* van de tabel *personen*. Merk op dat deze klasse bijvoorbeeld geen methoden bevat om contactgegevens aan een persoon toe te voegen. We zullen dit in een andere klasse implementeren.

De klasse *Contact* heeft een gelijkaardige structuur:

```
public record Contact(int id, int personId, char type, String address) {  
}
```

Merk op dat *type* als type **char** heeft, een type dat niet rechtstreeks door JDBC kan gebruikt worden. We zullen dit achter de schermen moeten omzetten van en naar *String*.

Tenslotte is er ook nog een record-klasse *ContactType* met velden *type* en *name*<sup>2</sup>.

Als tweede stap in het ontwerp moeten we vastleggen welke databankbewerkingen we zullen toelaten op de basisobjecten die we zojuist hebben gemaakt. Dit is waar de data access objects (DAO's) opduiken: DAO's zijn objecten (klassen) die toelaten om bepaalde bewerkingen op de databank uit te voeren, zowel zoekbewerkingen als bewerkingen die de databankgegevens wijzigen.

In ons ontwerp kiezen we voor drie verschillende DAO's: een personen-DAO, een contacten-DAO en een contacttype-DAO. DAO's worden doorgaans opgesplitst naar functionaliteit. Vaak komt die opsplitsing overeen met de verdeling van de databank in tabellen (zoals hier), maar dit is niet noodzakelijk zo. Sommige DAO's hebben tegelijkertijd verschillende databanktabellen nodig terwijl er in andere gevallen teveel bewerkingen nodig zijn op één bepaalde tabel om allemaal in één enkele DAO te passen.

Met elke DAO laten we een bepaalde *interface* overeenkomen. Voor de personen-DAO is dit bijvoorbeeld de volgende:

```
public interface PersonDAO {  
  
    int createPerson (String name, String firstName)  
                                   throws DataAccessException;  
  
    void updatePerson(int id, String name, String firstName)  
                                   throws DataAccessException;  
}
```

---

<sup>2</sup>Voor de verandering hebben we die als binnenklasse geïmplementeerd van de interface *ContactTypeDAO* — zie verder.

```

void deletePerson (int id) throws DataAccessException;

Iterable<Person> findPersons (String namePrefix)
                                throws DataAccessException;

}

```

(In de praktijk zal een dergelijke DAO wellicht nog heel wat meer methodes bevatten.)

Met de methode *createPerson* kan je een nieuwe persoon toevoegen aan de databank. Je geeft de familienaam en de voornaam mee als argument, als resultaat krijg je de unieke sleutel voor deze persoon zoals die automatisch gegenereerd werd door de databankserver.

```

PersonDAO dao = ...
int id = dao.createPerson ("Vanpoucke", "Jan");

```

De methode *updatePerson* laat toe om gegevens van een persoon in de databank aan te passen. Je geeft de identificatie van de persoon mee als argument en de nieuwe familienaam en voornaam voor die persoon.

```

dao.updatePerson (jan.id(), "Van Poucke", jan.firstName());

```

De methode *deletePerson* verwijdert de opgegeven persoon uit de databank. De methode *findPersons* geeft alle personen terug wiens familienaam begint met een zekere prefix:

```

for (Person person: dao.findPersons ("Jans")) {
    System.out.println (person.name() + ", " + person.firstName());
}

```

Merk op dat elke methode uit *PersonDAO* een *DataAccessException* kan opgooien. Dit is een (gecontroleerde) uitzonderingsklasse die we zelf hebben voorzien. Op deze manier verbergen we voor de programmeur dat er hier een *SQLException* achter schuilt.

De contacten-DAO heeft een gelijkaardige vorm:

```

public interface ContactDAO {

    Contact createContact (int personId, char code, String address)
                                throws DataAccessException;
    void updateContact (int id, String address) throws DataAccessException;
}

```

```

void deleteContact (int id) throws DataAccessException;

Iterable<Contact> findContacts (int personId) throws DataAccessException;
Iterable<Contact> findContactsByType (int personId, char type)
throws DataAccessException;
}

```

De methode *findContacts* geeft *alle* contactgegevens terug van een bepaalde persoon, de methode *findContactsByType* enkel die van een bepaald type (bijvoorbeeld, enkel e-mailadressen).

Tenslotte is er ook nog de contacttype-DAO:

```

public interface ContactTypeDAO {

    String findName(char type) throws DataAccessException;

    Iterable<ContactType> allContactTypes() throws DataAccessException;
}

```

## 9.2. Data access context en data access provider

Als we in een programma een DAO willen gebruiken, moeten we op één of andere manier een DAO-object zien te bekommen. Aangezien DAO's enkel als interface zijn gespecificeerd, kunnen we niet zomaar **new** gebruiken om een DAO-object aan te maken maar moeten we dit op een andere manier oplossen.

Hiervoor hebben we een zogenaamde *data access context*<sup>3</sup> (DAC) ingevoerd:

```

public interface DataAccessContext extends AutoCloseable {

    PersonDAO getPersonDAO ();

    ContactDAO getContactDAO ();

    ContactTypeDAO getContactTypeDAO ();

    void close () throws DataAccessException;

}

```

---

<sup>3</sup>De termen *data access context* en *data access provider* zijn niet standaard.



Wil je een DAO-object, dan vraag je dit dus op aan een DAC:

```
try (DataAccessContext dac = ...) {  
    PersonDAO pdao = dac.getPersonDAO();  
    int personId = pdao.createPerson("Janssens", "Siegfried");  
  
    ContactDAO cdao = dac.getContactDAO();  
    int contactId = cdao.createContact(personId, 'E', "sjanssens@gmail.com");  
    ...  
} catch (DataAccessException dae) {  
    ...  
}
```

Een data access *context* is een abstractie van een databank*connectie*. Alles wat je doet met één of meer DAO's die afkomstig zijn van dezelfde DAC zal worden uitgevoerd op dezelfde databankconnectie. Daarom moet een DAC ook altijd worden afgesloten nadat hij wordt gebruikt. Omdat *DataAccessContext* de interface *AutoCloseable* implementeert, kan je hiervoor een try-met-bronnen gebruiken, zoals in het bovenstaande voorbeeld<sup>4</sup>.

Je hebt al opgemerkt dat *DataAccessContext* opnieuw een interface is. Een DAC kan dus opnieuw niet zomaar worden aangemaakt met **new**. In de plaats daarvan vraag je een DAC op aan een data access *provider* (DAP):

```
public interface DataAccessProvider {  
    public DataAccessContext getDataAccessContext() throws DataAccessException;  
}
```

Een DAP vormt de abstractie van een volledige databank.

*DataAccessProvider* is opnieuw een interface, maar hier houdt het dan ook op — we hebben niet ook nog een *DataAccessProviderFactory* of zo, waaraan je providers kunt opvragen. Vooraleer we uitleggen hoe je dan uiteindelijk aan een concrete DAP komt, staan we even stil bij de reden waarom al deze data access klassen door interfaces worden voorgesteld en niet door concrete klassen.

De hoofdreden is *abstractie*. De gebruiker van onze 'data access'-bibliotheek hoeft niet te weten hoe alles concreet wordt geïmplementeerd. Dit biedt een zekere mate van veiligheid (hij kan geen misbruik maken van implementatiespecifieke functionaliteiten) maar zorgt vooral voor *flexibiliteit*. We kunnen onze implementaties bijschaven en optimaliseren zonder dat er iets hoeft te veranderen aan het programma dat onze bibliotheek gebruikt — en dit kan ver gaan.

---

<sup>4</sup>Het zou ons te ver leiden om hier dieper op in te gaan, maar een DAC zal in de praktijk meestal ook methodes bevatten om een zogenaamde *transactie* te beginnen, te bevestigen en terug te rollen.

Enkele voorbeelden:

- We kunnen beslissen om enkele databankgegevens in het geheugen bij te houden om snelheid te winnen.
- We kunnen de namen van de databanktabellen of van de kolommen in die tabellen wijzigen zonder dat er ook maar iets aan de data access interfaces hoeft te veranderen.
- We kunnen onze databank anders structureren, bijvoorbeeld tabellen samenvoegen of opsplitsen
- We kunnen na verloop van tijd voor andere databanksoftware kiezen. Misschien moeten we dan behalve driver en de JDBC-URL ook hier en daar nog enkele SQL-opdrachten aanpassen aan het nieuwe SQL-dialect.
- We kunnen beslissen om bepaalde gegevens uit de databank te halen en in de plaats op te slaan in een XML-bestand<sup>5</sup>. (De tabel *contactcodes* is hiervoor een mogelijke kandidaat.)

Bovendien kunnen we al het bovenstaande combineren, en het overlaten aan de bibliotheekgebruiker welke opties hij verkiest. We kunnen tegelijk een implementatie voorzien voor Apache Derby én één voor PostgreSQL. Of één voor een gewone databank, en één die helemaal geen databank gebruikt, maar alles in het geheugen bijhoudt in maps en lijsten — heel nuttig wanneer we de rest van het programma willen testen in een omgeving waar er geen databank voorhanden is, of waar die niet snel genoeg werkt.

## 9.3. Implementatie van de data access-bibliotheek

[ De implementatie die we hier bespreken, vind je in *be.ugent.objprog.contacts.jdbc*. ]

Hier zullen we ons beperken tot één enkele implementatie, gebaseerd op JDBC, met een kleine aanpassing: we veronderstellen dat de contacttypes tijdens het gebruik van onze toepassing niet zullen veranderen. Daarom halen we die op voorhand uit de databank en stoppen ze in een (hash)map. Maar dit blijft onzichtbaar voor de programmeur die onze bibliotheek gebruikt.

De klassen die de verschillende interfaces implementeren, zijn de volgende:

---

<sup>5</sup>In de praktijk is dit echter niet aan te raden. Als je toch een databank gebruikt, doe dit dan voor *alle* gegevens.

Interface	Implementatie
<i>DataProvider</i>	<i>JDBCDataProvider</i>
<i>DataAccessContext</i>	<i>JDBCDataAccessContext</i>
<i>PersonDAO</i>	<i>JDBCPersonDAO</i>
<i>ContactDAO</i>	<i>JDBCContactDAO</i>
<i>ContactTypeDAO</i>	<i>StaticContactTypeDAO</i> (!)

De meeste JDBC-code bevindt zich in *JDBCPersonDAO* en *JDBCContactDAO*. Om die te kunnen uitvoeren, hebben deze DAO's een *Connection*-object nodig. Dit krijgen ze mee als parameter van hun constructor. De DAO-objecten worden aangemaakt door de *JDBCDataAccessContext*. Dit object gebruikt dezelfde connectie voor elke DAO die het aanmaakt. Deze connectie is dan op haar beurt afkomstig van de *JDBCDataProvider*.

De contacttype-DAO is anders van opzet. Omdat deze helemaal geen JDBC-code gebruikt, heeft hij ook geen databankconnectie en kan hij gedeeld worden door de verschillende DAC's. Daarom wordt hij door de *JDBCDataProvider* aangemaakt en opgeslagen, die hem dan doorgeeft naar alle DAC's.

De methode *getDataAccessContext* van *JDBCDataProvider* ziet er zo uit:

```
public DataAccessContext getDataAccessContext() throws DataAccessException {
    try {
        return new JDBCDataAccessContext(contactTypeDAO, getConnection());
    } catch (SQLException ex) {
        throw new DataAccessException("Could not create DAC", ex);
    }
}
```

Het veld *contactTypeDAO* bevat de (unieke) contacttype-DAO. De methode *getConnection* creëert een nieuwe verbinding met de databank door *DriverManager.getConnection* op te roepen.

Omdat we ons niet willen vastpinnen op één bepaald databanktype en zeker al niet op één specifieke databankserver, zullen we de gegevens die *getConnection* nodig heeft (JDBC-URL, gebruikersidentificatie en wachtwoord), opslaan in een eigenschapsbestand (zoals in §3.7). De methode *getConnection* van *JDBCDataProvider* haalt zijn informatie dus uit een *Properties*-object.

```
private Connection getConnection() throws SQLException {
    String user = databaseProperties.getProperty("user");
    if (user != null) {
        return DriverManager.getConnection(
            databaseProperties.getProperty("url"),
```

```

        user, databaseProperties.getProperty("password"));
    } else {
        return DriverManager.getConnection(
            databaseProperties.getProperty("url"));
    }
}

```

(We laten ook databanken toe die geen gebruikersidentificatie vereisen.)

Het eigenschapsbestand dat we nodig hebben, zal doorgaans als bron aan de software worden toegevoegd. Om een *JDBCDataAccessProvider* aan te maken, geven we dan de naam van die bron door als parameter van zijn constructor. Een programma dat onze bibliotheek gebruikt, zal daarom ergens in het begin een opdracht uitvoeren zoals deze:

```

DataAccessProvider dap =
    new JDBCDataAccessProvider ("/config/databank.properties");

```

Verder in het programma wordt dan steeds dezelfde variabele *dap* gebruikt om databankbewerkingen uit te voeren. Merk trouwens op wat het type is van *dap*. In principe is bovenstaande lijn de enige waaruit blijkt welke implementatie van *DataAccessProvider* we precies gebruiken. En dit is dan ook de enige lijn die we moeten aanpassen wanneer we een andere implementatie wensen uit te proberen.

Het is niet zo moeilijk om een DAC te implementeren. Deze doet immers niet veel meer dan de gepaste DAO's aanmaken en uitdelen wanneer ernaar wordt gevraagd.

```

public class JDBCDataAccessContext implements DataAccessContext {

    public PersonDAO getPersonDAO() {
        return new JDBCPersonDAO(connection);
    }

    public ContactDAO getContactDAO() {
        return new JBCCContactDAO(connection);
    }

    public ContactTypeDAO getContactTypeDAO() {
        return contactTypeDAO; // altijd dezelfde!
    }

    ...
}

```

Het echte JDBC-werk gebeurt in de DAO's zelf. De implementatie van de verschillende methodes ligt in de lijn van wat we reeds eerder hebben gedaan.

Bekijk als voorbeeld de *deletePerson*-methode uit *JDBCPersonDAO*:

```
public void deletePerson(int id) throws DataAccessException {  
    try (PreparedStatement ps = prepare(  
        "DELETE FROM personen WHERE id = ?"  
    )) {  
        ps.setInt(1, id);  
        ps.executeUpdate();  
    } catch (SQLException ex) {  
        throw new DataAccessException("Could not delete person.", ex);  
    }  
}
```

We hebben een abstracte bovenklasse *JDBCAbstractDAO* ingevoerd die methodes bevat die gemeenschappelijk zijn aan meerdere DAO's — zoals de methode *prepare* die we hierboven gebruiken in de tweede lijn van *deletePerson*.

```
public class JDBCAbstractDAO {  
  
    private final Connection connection;  
  
    public JDBCAbstractDAO(Connection connection) {  
        this.connection = connection;  
    }  
  
    protected PreparedStatement prepare (String sql) throws SQLException {  
        return connection.prepareStatement(sql);  
    }  
  
    ...  
}
```

De andere DAO-implementatieklassen zijn dan extensies van deze klasse.

```
class JDBCPersonDAO extends JDBCAbstractDAO implements PersonDAO {  
  
    public JDBCPersonDAO(Connection connection) {  
        super(connection);  
    }  
  
    ...  
}
```

Merk op dat de JDBC-implementaties van de DAO's (en van de DAC) niet **public** zijn, maar slechts toegankelijk zijn binnen het pakket met alle JDBC-implementaties. Hierdoor zorgen we er nog beter voor dat een gebruiker van onze DAO-bibliotheek de implementatiedetails niet kan 'misbruiken'.

## 9.4. Filteren met een fluent interface

In een professionele databanktoepassing met tabellen die vele tientallen duizenden rijen bevatten, zal met vaak toelaten om slechts een gedeelte van de inhoud van die tabellen op te vragen gebaseerd op enkele criteria: we zoeken personen waarvan de familienaam begint met een gegeven prefix, of die na een bepaalde datum in het bedrijf zijn aangeworven, of die tot een bepaald departement behoren, enz.

Voor elk van die gevallen kunnen we gemakkelijk in de betreffende DAO afzonderlijke methodes voorzien:

```
Iterable<Person> findPersonsWithNamePrefix (String prefix)  
Iterable<Person> findPersonsAppointedAfterDate (LocalDate date)  
Iterable<Person> findPersonsInDepartment (int departmentId)  
...
```

maar elk van die methodes laat slechts toe om op één criterium te 'filteren'. Wensen we meerdere criteria tegelijkertijd toe te passen dan hebben we methodes nodig met meer dan één parameter, in alle mogelijke combinaties. Heb je  $n$  verschillende zoekcriteria, dan moet je eigenlijk  $2^n$  verschillende methodes in de DAO opnemen. Dit is in de praktijk niet haalbaar.

Om dit op te lossen, voegen we aan het ontwerp van onze bibliotheek klassen en methodes toe die de programmeur toelaten om de data access context bijvoorbeeld op de volgende manier te gebruiken:

```
Iterable<Person> persons  
    = context.findPersons()  
              .whereNameStartsWith("C")  
              .whereFirstNameStartsWith("K")  
              .getList ();
```

De structuur van methode-oproepen die we hier gebruiken wordt een '*fluent interface*' genoemd. Het Engelse woord *fluent* verwijst naar het feit dat de Java-code 'vloeiend' leesbaar is.

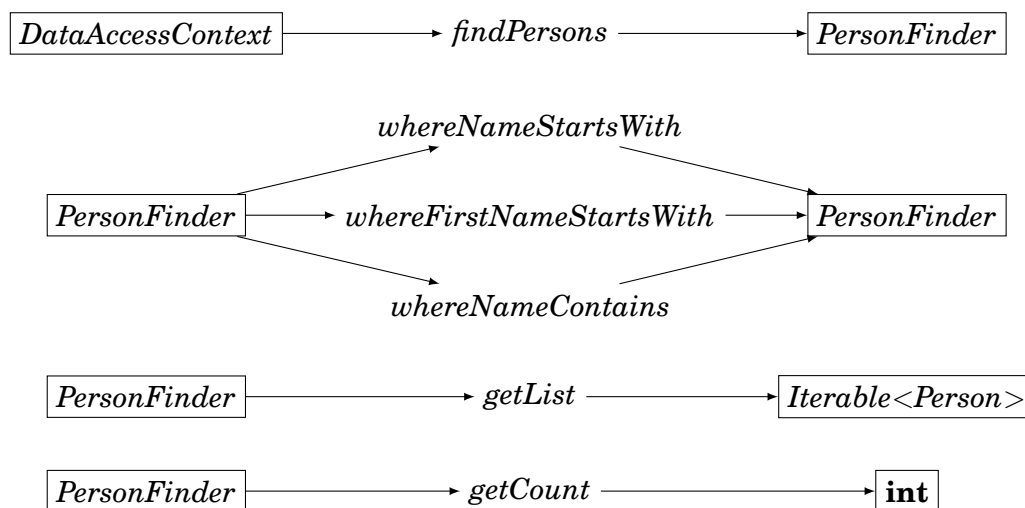
We laten enkele varianten toe: als je op het einde *getCount* gebruikt in de plaats van *getList*, krijg je enkel het aantal personen terug die aan het criterium voldoen. Laat je de *where*-methodes weg, dan wordt er niet gefilterd. Dus

```
int aantal = context.findPersons().getCount();
```

geeft je bijvoorbeeld het aantal personen terug dat is opgeslagen in de databank.

Een fluent interface is heel gebruiksvriendelijk, maar helaas niet zo eenvoudig om te implementeren. Omdat de methodes niet zomaar in een willekeurige volgorde kunnen uitgevoerd worden (niet eerst *whereNameStartsWith* en dan pas *findPersons*, bijvoorbeeld), moet je vaak heel wat nieuwe klassen (of interfaces) invoeren waarin je de verschillende methodes groepeert.

In dit eenvoudige voorbeeld valt dit nog mee: we hoeven slechts één nieuwe interface te introduceren — we dopen hem *PersonFinder*. Onderstaand schema toont de relatie tussen de verschillende interfaces en methodes:



De interface *PersonFinder* heeft dus de volgende methodes:

```
public interface PersonFinder {
    PersonFinder whereNameStartsWith(String prefix);
    PersonFinder whereFirstNameStartsWith(String prefix);
    PersonFinder whereNameContains(String fragment);
    Iterable<Person> getList() throws DataAccessException;
    int getCount() throws DataAccessException;
}
```

en *DataContext* krijgt er de volgende methode bij:

```
public interface DataContext extends AutoCloseable {  
    PersonFinder findPersons();  
    ...  
}
```

Zoals bij de DAO's, voorzien we een JDBC-implementatie van *PersonFinder* — met de naam *JDBCPersonFinder*. Het is deze implementatie die zal teruggegeven worden als waarde van de methode *findPersons* uit *JDBCDataContext*.

De implementatie van *JDBCPersonFinder* is niet zo eenvoudig en rechtlijnig als deze van *JDBCPersonDAO*. De klasse moet bijhouden op welke criteria er zal moeten gefilterd worden op het moment dat *getCount* of *getList* worden opgeroepen. We doen dit door intern de *WHERE*-clausule van de SQL-opdracht bij te houden en de parameters die moeten ingevuld worden voor de vraagtekens in het overeenkomstige *PreparedStatement*. We gebruiken deze informatie dan in *getList* (en analoog voor *getCount*):

```
private final String whereClause;  
private final List<String> parameters;  
  
public Iterable<Person> getList() throws DataContextException {  
    try (PreparedStatement ps = connection.prepareStatement(  
        "SELECT id,voornaam,familienaam " +  
        "FROM personen " + whereClause)) {  
        int index = 1;  
        for (String parameter : parameters) {  
            ps.setString(index, parameter);  
            index++;  
        }  
        try (ResultSet rs = ps.executeQuery()) {  
            List<Person> result = new ArrayList<>();  
            while (rs.next()) {  
                result.add(new Person(  
                    rs.getInt("id"),  
                    rs.getString("familienaam"),  
                    rs.getString("voornaam")));  
            }  
            return result;  
        }  
    } catch (SQLException ex) {  
        throw new DataContextException(" . . . ", ex);  
    }  
}
```



De *where*-methodes vragen wat meer denkwerk: elk van deze methodes moet, vanuit een bestaande *PersonFinder*, een nieuwe *PersonFinder* aanmaken met een `WHERE`-clause die gevormd wordt uit de originele clause met daaraan een nieuwe criterium toegevoegd. Dit wordt extra gecompliceerd doordat de vorm van een `WHERE`-clause afhangt van hoeveel criteria er zijn opgegeven.

- Zijn er nog geen criteria, dan is de clause leeg, anders begint ze met `WHERE`.
- Vanaf het tweede criterium, komt er een `AND` voor de voorwaarde.

We gebruiken onderstaande methode *extend* om een *PersonFinder* met één clause (en overeenkomstige parameter) uit te breiden:

```
private JDBCPersonFinder extend(String condition, String parameter) {
    if (parameters.isEmpty()) {
        return new JDBCPersonFinder(
            connection, " WHERE " + condition,
            List.of(parameter)
        );
    } else {
        List<String> copy = new ArrayList<>(parameters);
        copy.add(parameter);
        return new JDBCPersonFinder(
            connection, whereClause + " AND " + condition, copy
        );
    }
}
```

Hierbij gebruiken we de volgende constructor van *JDBCPersonFinder*:

```
private JDBCPersonFinder(Connection connection, String whereClause,
    List<String> parameters) {
    this.connection = connection;
    this.whereClause = whereClause;
    this.parameters = parameters;
}
```

De publieke constructor (die o.a. door *findPersons* in *JDBCDataAccessContext* gebruikt wordt) is de volgende:

```
JDBCPersonFinder(Connection connection) {
    this(connection, "", List.of());
}
```

Met de hulpmethode *extend* wordt het nu relatief eenvoudig om *where*-methodes te schrijven:

```
public PersonFinder whereNameStartsWith(String prefix) {  
    return extend("familienaam LIKE (? || ' %' )", prefix);  
}  
...
```

We hebben hier slechts een tipje van de sluier opgelicht. De implementatie van *JDBC-PersonFinder* wordt heel wat moeilijker wanneer we ook *where*-methodes toelaten waarvan de parameters geen strings zijn, maar gehele getallen, datums, enz. In dat geval zal de lijst van criteria niet meer kunnen voorgesteld worden als een simpele clausestring met bijbehorende parameterstrings.

In de praktijk zal je ook methodes zoals *orderBy* en *restrictToPage* aan de fluent interface willen toevoegen om het resultaat in een bepaalde volgorde terug te krijgen en te pagineren. Wellicht wil je dat deze methodes enkel na een reeks *where*-methodes kunnen opgeroepen worden, dat *restrictToPage* hoogstens één keer kan voorkomen en dat *getCount* alleen geldig is als er geen *orderBy* of *restrictToPage* staat, .... Het zal dan niet voldoende zijn om slechts één nieuwe interface *PersonFinder* te introduceren en te implementeren.

## A. Databanken en SQL

Een relationele databank groepeerd zijn gegevens in zogenaamde *tabellen*. Elke *rij* uit die tabel bevat een lijst van waarden die bij elkaar horen. De *kolommen* van de tabel geven betekenis aan die elementen.

De tabel *personen* die we hieronder afbeelden, bevat bijvoorbeeld een aantal gegevens uit een personendatabank:

<i>id</i>	<i>familienaam</i>	<i>voornaam</i>
101	Janssens	Griet
105	Peters	Bram
104	Vander Ven	Hendrik
107	Janssens	Jan
111	Huybrechts	Bram

In dit voorbeeld is het '*id*' van een persoon uniek voor die persoon en helpt het onder andere om twee personen met dezelfde naam van elkaar te onderscheiden. De meeste tabellen in een databank hebben een dergelijke kolom die dient om één rij van die tabel uniek te kunnen identificeren. Dit noemt men een *primaire sleutel* (Engels: *primary key*) voor die tabel.

Wanneer hij een tabel in een databank creëert, legt de databankbeheerder vast wat de naam is van die tabel, hoeveel kolommen de tabel bevat, wat de namen zijn van die kolommen (in dit voorbeeld, *id*, *familienaam* en *voornaam*) en wat de *types* zijn van die kolommen (getal, tekst, tekst). De gegevens in de rijen zelf worden tijdens de toepassing ingevuld, veranderd of verwijderd.

Merk op dat 'tabel' een abstracte notie is. Ook al drukken we een tabel meestal in die vorm af, betekent dit daarom niet noodzakelijk dat ze ook als 2-dimensionale structuur in de databank is opgeslagen. We hoeven ons echter niet druk te maken over hoe een databank er 'vanbinnen' uitziet.

Moderne databanksoftware laat toe om gegevens op te zoeken, in te voegen, te verwijderen en te veranderen met behulp van de computertaal *SQL*.

De volgende SQL-opdracht vraagt bijvoorbeeld de identificatienummers en voornamen op van alle personen die Janssens heten:

```
SELECT id, voornaam FROM personen WHERE familienaam='Janssens';
```

Het resultaat van deze zoekopdracht is de volgende lijst van gegevens (opnieuw voorgesteld als een tabel):

<i>id</i>	<i>voornaam</i>
107	Jan
101	Griet

Merk op dat de volgorde van de rijen in het resultaat niet a priori vastligt, en wellicht afhangt van de interne structuur van de databank op dat moment. Je kan zelf een volgorde vastleggen met behulp van de `ORDER`-clausule:

```
SELECT id, voornaam FROM personen
WHERE familienaam='Janssens'
ORDER BY voornaam;
```

In dit geval zal de uitvoer alfabetisch gerangschikt zijn volgens voornaam.

Om de volledige tabel in één keer op te vragen, gebruik je de volgende eenvoudige opdracht:

```
SELECT * FROM personen;
```

Een `SELECT`-opdracht kan ook gegevens opvragen die verspreid zijn over verschillende tabellen. Bekijk bijvoorbeeld onderstaande tabellen *contactgegevens* (links) en *contactcodes* (rechts):

<i>id</i>	<i>p_id</i>	<i>code</i>	<i>adres</i>	<i>id</i>	<i>naam</i>
305	101	M	0470/12 34 56	M	Mobiele telefoon
307	101	E	griet.janssens@ugent.be	T	Vaste telefoon
309	104	M	0487/21 43 33	E	E-mail
311	105	M	0491/99 77 52	F	Fax
314	105	T	053/22 87 16		
304	105	E	bram.peeters@ugent.be		
330	111	E	bram.huybrechts@ugent.be		
317	111	E	brhuyb@gmail.com		

De tabel *contactgegevens* houdt telefoonnummers en e-mailadressen bij van de personen in de databank. In plaats van de volledige naam van de betreffende persoon, bevat deze

tabel een verwijzing naar de tabel *personen*: de inhoud van de kolom *p\_id* verwijst naar de persoon met de gegeven primaire sleutel.

Onderstaande opdracht drukt alle e-mailadressen af van iedereen die 'Bram' heet:

```
SELECT adres FROM personen, contactgegevens
WHERE voornaam='Bram' AND p_id = personen.id AND code = 'E';
```

(Omdat er in beide tabellen een *id*-kolom bestaat, moeten we die nader specificeren met de naam van de tabel.)

De volgende opdracht drukt alle contactgegevens af van de persoon met *id* 105, maar met de code voluit in plaats van in afgekorte vorm.

```
SELECT naam, adres FROM contactgegevens, contactcodes
WHERE p_id=105 AND contactgegevens.code = contactcodes.id;
```

Je kan met een `SELECT`-opdracht ook aantallen rijen tellen. Onderstaande opdracht vraagt aan de gegevensbank hoeveel personen er Janssens heten.

```
SELECT count(*) FROM personen WHERE familienaam='Janssens';
```

Het resultaat hiervan is een tabel met één rij en één kolom die enkel het gevraagde aantal bevat (in ons voorbeeld: 2).

Als laatste voorbeeld van een `SELECT` drukken we alle personen af met een e-mailadres bij het domein `ugent.be`.

```
SELECT voornaam, familienaam FROM personen,contactgegevens
WHERE personen.id = contactgegevens.p_id AND adres LIKE '%@ugent.be';
```

De `LIKE`-operator vergelijkt een string met een patroon dat jokertekens kan bevatten: een `'_'` is een joker voor één letterteken, een `'%'` voor nul of meer opeenvolgende tekens.

Er zijn ook SQL-opdrachten waarmee je de databankgegevens kan aanpassen. Om rijen toe te voegen aan de tabel gebruik je bijvoorbeeld een `INSERT`-opdracht:

```
INSERT INTO personen VALUES (119, 'Rogiers', 'Pierre');
```

Bestaande waarden kan je veranderen met `UPDATE`:

```
UPDATE personen SET voornaam='Piet' WHERE id=101;
```

Bovenstaande opdracht verandert de voornaam van één bepaalde persoon. Je kan UPDATE-opdrachten ook gebruiken om meerdere rijen tegelijkertijd te veranderen. De volgende twee opdrachten veranderen bijvoorbeeld overal de contactcode ‘T’ in ‘V’:

```
UPDATE contactgegevens SET code = 'V' where code = 'T';  
UPDATE contactcodes SET id = 'V' where id = 'T';
```

SQL laat ook allerlei rekenkundige bewerkingen toe. Zo kunnen we bijvoorbeeld de personenidentificaties van iedereen met 1 verhogen op de volgende manier:

```
UPDATE personen SET id=id+1;
```

Omdat de WHERE ontbreekt, wordt deze verandering uitgevoerd op elke rij van de tabel.

De DELETE-opdracht dient om rijen te verwijderen.

```
DELETE FROM personen WHERE id=119;
```

Nog een voorbeeld:

```
DELETE FROM contactgegevens WHERE adres LIKE '%@gmail.com';
```

Tot slot merken we nog op dat SQL ook kan gebruikt worden om de tabellen zelf te manipuleren — tabellen aan te maken of te vernietigen, kolommen bij te voegen of te hernoemen — maar dit valt buiten het bestek van deze korte inleiding.