Project Kantine

Een simulatie met database

Instituut voor Communicatie, Media & IT

2019 – 2020

Inhoudsopgave

Inleiding

De komende zes weken wordt in *tweetallen* gewerkt aan het maken van een kantine-simulatie.[1] De uiteindelijke simulatie is een text-based applicatie waarbij klanten artikelen kunnen kiezen, in de rij gaat staan en afrekenen. Allerlei zaken zoals het betalen met contant geld of pin en het berekenen van omzetgegevens komen gaandeweg aan bod. Het laatste deel is gericht op het leren begrijpen hoe met meer complexe datastructuren gewerkt kan worden en hoe data opgeslagen wordt.

Elke week bestaat uit een aantal opgaven. De eerste opgaven zijn vooral gericht op het opfrissen van de Java kennis. Vervolgens wordt het schrijven van algoritmes en programmastructuur behandeld en als laatste het werken met data.

In dit project wordt een aantal skeletcodes gegeven. Deze skeletcode kun je als zip downloaden van Blackboard. Het is de bedoeling om gezamenlijk aan de code te werken, maak dus gebruik van een repository.

Regels

- De deadline voor het inleveren van de uitwerking is steeds 18:00 op de eerste maandag volgend op de week waar de opgaven bij horen. Je levert dus bijvoorbeeld je uitwerking van week 1 uiterlijk maandag in week 2 in. De docent zal vragen om een korte demonstratie en uitleg van jouw uitwerking.
- Elke week wordt het opgeleverde werk beoordeeld met een uitmuntend, goed, voldoende of onvoldoende (10, 8, 6, 4). Het uiteindelijke cijfer zal een gemiddelde zijn van deze beoordelingen.

[1] Als een klas een oneven aantal studenten telt zal één groep uit drie leden bestaan.

- Indien een week met onvoldoende beoordeeld wordt is er de volgende week een mogelijkheid tot herkansing.
- Het is niet erg als je de opgaven verdeelt, maar je wordt wel geacht alle uitwerkingen te kunnen toelichten. Een excuus als „Dat heb ik niet gemaakt, dus dat kan ik niet uitleggen” wordt dan ook gezien als het niet hebben gemaakt van de opgave.
- Niet aanwezig zijn bij de nabespreking is hetzelfde als het niet hebben gemaakt van de opgave.
- In principe wordt één cijfer aan een groep gegeven, maar de docent zal hier van afwijken als er sprake is van onevenredige werkverdeling of het niet kunnen uitleggen van een uitwerking.

Voorbereiding

Download `KantineSimulatieBasis.zip` en pak het uit. Het project is al configureerd zodat het eenvoudig in een IDE als IntelliJ, Eclipse of VSCode (met de Java extensies) kan worden geïmporteerd. Kies voor de „Maven” configuratie mocht de IDE jou hier naar vragen.

De configuratie verwacht dat je Java SDK versie 11 (LTS) of hoger hebt geïnstalleerd. Mocht dit niet het geval zijn dan kan je deze downloaden van https://adoptopenjdk.net/.

Week 1: Typen, klassen en objecten

In deze week kijken we weer naar de basisbeginselen van object georiënteerd programmeren met Java. We doen dat aan de hand van een drietal klassen, `Artikel`, `Datum`, `Persoon` en `Dienblad`.

Opgave 1: De klasse `Artikel`

a. Maak een klasse `Artikel` waarin de volgende gegevens kunnen worden opgeslagen:

   - naam
   - prijs

   Bedenk zelf wat goede datatypen zijn voor deze gegevens.

b. Je hebt hierboven twee instantievariabelen gedeclareerd. Voordat je ze zinnig zou kunnen gebruiken moet je ze wel initialiseren. Leg uit wat de begrippen declaratie en initialisatie betekenen.

c. Eén manier om de instantievariabelen een waarde te geven is via de constructor. Maak een constructor die dezelfde gegevens uit vraag a) als parameters heeft en de instantievariabelen de meegegeven waarden geeft, dat wil zeggen een constructor van de vorm

```
public Artikel(... naam, ... prijs) {
    ...
}
```

d. Maak ook een parameter-loze constructor voor deze klasse.

e. Mutator en accessor methoden worden ook wel setters en getters genoemd vanwege de eerste drie letters die deze methoden hebben. Maak getters en setters voor de twee instantievariabelen.

Opgave 2: De klasse `Datum`

Voordat we met de klasse `Persoon` bezig kunnen moet eerst de klasse `Datum` afgerond worden zodat we in de klasse `Persoon` het veld geboortedatum goed kunnen gebruiken. We kunnen hier een `java.time.LocalDate` object van maken maar voor nu (en ter oefening) maken we een eigen type aan, `LocalDate` zullen we later in het project gaan gebruiken.

a. De velden dag, maand en jaar zijn al gegeven in de klasse `Datum`. Je IDE kan je helpen om de setters en getters en constructors automatisch aan te maken. In Eclipse bijvoorbeeld zijn deze tools te benaderen via Alt-Shift-S. Zorg er voor dat alle getters en setters aangemaakt worden, een constructor met de drie velden en een parameter-loze constructor. Zorg er voor dat in de laatste de velden op 0 worden gezet.

b. Een datum moet natuurlijk aan een aantal eisen voldoen. Maak daarom de controle methode `bestaatDatum()` die de volgende controles uitvoert:

- Dagnummers moeten altijd groter dan of gelijk zijn aan 1;
- De maanden liggen tussen 1 en 12;
- De jaren liggen tussen 1900 en 2100;
- De dag/maand combinatie moet bestaan. Zoals je wellicht weet hebben de maanden 1, 3, 5, 7, 8, 10 en 12 31 dagen, maand 2 28 dagen (met uitzondering van schrikkeljaren, zie opgave 5) en resterende maanden 30 dagen. In gewone mensentaal: de geboortedata 34 januari 1987 en 31 april 2002 zijn niet mogelijk, terwijl 31 maart 2000 wel een geldige datum is.[2]
- In een schrikkeljaar heeft de maand februari 29 in plaats van 28 dagen. Een jaar is een schrikkeljaar als het jaartal deelbaar is door 4, maar als het jaar deelbaar is door 100 is het

[2] Hint: Je kunt (dus niet verplicht) voor deze controle een switch-statement gebruiken, zie bijlage D BlueJ boek. Zie ook bijvoorbeeld http://www.faqs.org/docs/javap/c3/s6.html waarin uitgelegd wordt dat je meerdere case waarden in de switch kan combineren tot één resultaat, iets wat handig is in dit geval.

geen schrikkeljaar, tenzij het jaar deelbaar is door 400. Het jaar 1900 is dus geen schrikkeljaar, de jaren 2000, 2008, 2012 en 2016 zijn dat wel.[3]

c. Zorg er voor dat in de constructor de methode `bestaatDatum` wordt aangeroepen. Als de datum niet juist is moeten de velden dag, maand en jaar met 0 gevuld worden, dit kan eenvoudig door als eerste in deze constructor de parameter loze constructor aan te roepen, en dan als de datum correct is de waarden te vullen met de aangeleverde waarden.

Opgave 3: De klasse `Persoon`

a. Maak een klasse `Persoon` waarin de volgende gegevens kunnen worden opgeslagen:

- BSN (BurgerServiceNummer);
- Voornaam;
- Achternaam;
- Geboortedatum;
- Geslacht (M/V). Gebruik het datatype **`char`** om dit gegeven op te slaan.

b. Maak vijf setters voor deze instantievariabelen. Let hierbij op de volgende eisen:

- De setter van geslacht heeft een controle nodig. Bedenk zelf welke controle dat is en bouw deze ook in. Doe iets met de waarde van geslacht als de controle mislukt, zodat duidelijk is dat de waarde niet goed is gezet door de setter.

c. Maak de getters en de constructors met en zonder parameters weer aan door gebruik te maken van de opties van je IDE. Pas vervolgens de aangemaakte code zo aan zodat:

- De getter van geslacht een `String` teruggeeft: „Man” of „Vrouw”. Als de waarde van geslacht geen correcte waarde heeft (zie ook vraag b)) retourneer dan „Onbekend”.
- De getter van `geboortedatum` geeft ook een `String` terug. Zorg ervoor dat de methode `getDatumAsString` uit de `Datum` klasse hier wordt aangeroepen.
- De constructor met parameters ook de controle voor het veld geslacht uitvoert. Er zijn meerdere mogelijkheden, maar voorkom in ieder geval dubbele code!
- In de andere constructor moeten de instantievariabelen voor geboortedatum en geslacht een waarde krijgen zodanig dat de getters „Onbekend” teruggeven.

d. Voeg aan de klassen `Persoon` en `Artikel` een **`public`** `String toString()` methode toe waarmee je de waarden van de instantievariabelen laat zien.[4]

[3] Gebruik hierbij de modulo-operator (%).

[4] Hint: gebruik de getters voor de velden geslacht en geboortedatum.

Opgave 4: De klasse `Dienblad`

Als een persoon de kantine binnenloopt, pakt deze een dienblad, een aantal artikelen en plaatst deze op het dienblad. In de aangeleverde code staat de skeletcode voor de klasse `Dienblad`. De naam van deze klasse dekt eigenlijk niet geheel de lading van de functionaliteit. Het idee is dat objecten van deze klasse een container zijn voor de gegevens van de klant, zoals bijvoorbeeld een shopping cart op een website. Hierin worden de klantgegevens en de geselecteerde artikelen bewaard. In de skeletcode staan al de methodes `voegToe`, `getAantalArtikelen` en `getTotaalPrijs`.

```
1  public class Dienblad {
2      private ArrayList<Artikel> artikelen;
3
4      /**
5       * Constructor
6       */
7      public Dienblad() {
8          // method body omitted
9      }
10
11     /**
12      * Methode om artikel aan dienblad toe te voegen
13      *
14      * @param artikel
15      */
16     public void voegToe(Artikel artikel) {
17         // method body omitted
18     }
19
20     /**
21      * Methode om aantal artikelen op dienblad te tellen
22      *
23      * @return Het aantal artikelen
24      */
25     public int getAantalArtikelen() {
26         // method body omitted
27     }
28
29     /**
30      * Methode om de totaalprijs van de artikelen
31      * op dienblad uit te rekenen
32      *
33      * @return De totaalprijs
34      */
35     public double getTotaalPrijs() {
36         // method body omitted
37     }
38 }
```

a. Vul de bovenstaande klasse aan, zodat de methodes die gegeven zijn doen wat ze moeten doen

b. Zorg er voor dat bij het dienblad een klant opgeslagen kan worden en maak hiervoor een tweede constructor aan die een klant instantie van type `Persoon` als parameter heeft. Maak ook de bijbehorende getters en setters voor de `klant` variabele.

Opgave 5: De klasse `Kassarij`

Nadat een persoon alle gewenste artikelen op het dienblad heeft geplaatst, sluit deze zich achteraan in de rij voor de kassa. De kassarij wordt volgens het First In First Out (FIFO) principe afgewerkt. Hieronder zie je de skeletcode voor de klasse `Kassarij`:

```java
public class KassaRij {

    /**
     * Constructor
     */
    public KassaRij() {
        // method body omitted
    }

    /**
     * Persoon sluit achter in de rij aan
     *
     * @param klant
     */
    public void sluitAchteraan(Dienblad klant) {
        // method body omitted
    }

    /**
     * Indien er een rij bestaat, de eerste klant uit
     * de rij verwijderen en retourneren.
     * Als er niemand in de rij staat geeft deze null terug.
     *
     * @return Eerste klant in de rij of null
     */
    public Dienblad eerstePersoonInRij() {
        // method body omitted
    }

    /**
     * Methode kijkt of er personen in de rij staan.
     *
     * @return Of er wel of geen rij bestaat
     */
    public boolean erIsEenRij() {
        // method body omitted
    }
}
```

Implementeer deze klasse. Gebruik hierbij een `ArrayList<Dienblad>` om de personen in op te slaan. Een andere manier van opslaan komt terug in de opgaven van volgende week.

Opgave 6: De klasse `Kassa`

Hieronder zie je de skeletcode voor de klasse `Kassa`. Implementeer deze klasse.

```
1  public class Kassa {
2
3      /**
4       * Constructor
5       */
6      public Kassa(KassaRij kassarij) {
7          // method body omitted
8      }
9
10     /**
11      * Vraag het aantal artikelen en de totaalprijs op.
12      * Tel deze gegevens op bij de controletotalen die voor
13      * de kassa worden bijgehouden. De implementatie wordt
14      * later vervangen door een echte betaling door de persoon.
15      *
16      * @param klant die moet afrekenen
17      */
18     public void rekenAf(Dienblad klant) {
19         // method body omitted
20     }
21
22     /**
23      * Geeft het aantal artikelen dat de kassa heeft gepasseerd,
24      * vanaf het moment dat de methode resetWaarden is aangeroepen.
25      *
26      * @return aantal artikelen
27      */
28     public int aantalArtikelen() {
29         // method body omitted
30     }
31
32     /**
33      * Geeft het totaalbedrag van alle artikelen die de kass
34      * zijn gepasseerd, vanaf het moment dat de methode
35      * resetKassa is aangeroepen.
36      *
37      * @return hoeveelheid geld in de kassa
38      */
39     public double hoeveelheidGeldInKassa() {
40         // method body omitted
41     }
42
```

```
43      /**
44       * reset de waarden van het aantal gepasseerde artikelen en
45       * de totale hoeveelheid geld in de kassa.
46       */
47      public void resetKassa() {
48          // method body omitted
49      }
50  }
```

Week 2: De code verbeteren (refactoren)

Deze week staat in het teken van het aanpassen (refactoren) en deels uitbreiden van de code van vorige week. Voorzie al je code van zinvol commentaar en Javadoc.

Opgave 1: De klasse `Kantine`

Hieronder zie je de skeletcode voor de `Kantine` klasse.

```
1   public class Kantine {
2
3       private Kassa kassa;
4       private KassaRij kassarij;
5
6       /**
7        * Constructor
8        */
9       public Kantine() {
10          kassarij = new KassaRij();
11          kassa = new Kassa(kassarij);
12      }
13
14      /**
15       * In deze methode wordt een Persoon en Dienblad gemaakt
16       * en aan elkaar gekoppeld. Maak twee Artikelen aan
17       * en plaats deze op het dienblad. Tenslotte sluit de
18       * Persoon zich aan bij de rij voor de kassa.
19       */
20      public void loopPakSluitAan() {
21          // method body omitted
22      }
23
24      /**
25       * Deze methode handelt de rij voor de kassa af.
26       */
27      public void verwerkRijVoorKassa() {
28          while () {
```

```
            // omitted
        }
    }

    /**
     * Deze methode telt het geld uit de kassa
     *
     * @return hoeveelheid geld in kassa
     */
    public double hoeveelheidGeldInKassa() {
       // method body omitted
    }

    /**
     * Deze methode geeft het aantal gepasseerde artikelen.
     *
     * @return het aantal gepasseerde artikelen
     */
    public int aantalArtikelen() {
        // method body omitted
    }

    /**
     * Deze methode reset de bijgehouden telling van
     * het aantal artikelen en "leegt" de inhoud van de kassa.
     */
    public void resetKassa() {
        // method body omitted
    }
}
```

a. Leg uit waarom het gebruik van een while lus in de methode `verwerkRijVoorKassa()` handiger is dan een for lus.

b. Implementeer de ontbrekende methoden.

Opgave 2: De klasse `KantineSimulatie`

In deze opgave ga je de kantine-simulatie starten aan de hand van de volgende code.

```
public class KantineSimulatie {

    private Kantine kantine;

    public static final int DAGEN = 7;

    /**
     * Constructor
     */
```

```
10      public KantineSimulatie() {
11          kantine = new Kantine();
12      }
13
14      /**
15       * Deze methode simuleert een aantal dagen in het
16       * verloop van de kantine
17       *
18       * @param dagen
19       */
20      public void simuleer(int dagen) {
21
22          // herhaal voor elke dag
23          for (i = 0, ...) {
24
25              // per dag nu even vast 10 + i personen naar binnen
26              // laten gaan, wordt volgende week veranderd...
27
28              // for lus voor personen
29              for (int j = 0; j < 10 + i; j++) {
30                  // kantine.(...);
31              }
32
33              // verwerk rij voor de kassa
34
35              // toon dagtotalen (artikelen en geld in kassa)
36
37              // reset de kassa voor de volgende dag
38          }
39      }
40
41      /**
42       * Start een simulatie
43       */
44      public static void main(String[] args) {
45          int dagen;
46
47          if (args.length == 0) {
48              dagen = DAGEN;
49          } else {
50              dagen = Integer.parseInt(args[0]);
51          }
52
53          simulate(dagen);
54      }
55  }
```

a. Vul de ontbrekende delen in en voer de simulatie uit.

Opgave 3: Alternatieve opslagstructuren

De klassen `Dienblad` en `Kassarij` gebruiken allebei intern een `java.util.ArrayList` om Artikelen of Personen op te slaan. Je zou kunnen zeggen dat een dienblad een „stapelstructuur" heeft; dat wil zeggen dat het eerste artikel dat er op wordt geplaatst als laatste wordt afgehaald. Dit wordt ook wel een LIFO systeem genoemd, dit betekent Last In First Out. Zoals al eerder is opgemerkt in opgave 5) van de vorige week heeft een kassarij juist de omgekeerde eigenschap, namelijk FIFO.

Als je meer ervaring krijgt met programmeren ontdek je dat deze twee structuren veel vaker voorkomen. In de Java bibliotheek in de `java.util.*` package (zie de documentatie op oracle.com) kun je een `Stack` (stapel, het LIFO systeem) en een `Queue` (rij, het FIFO systeem) terugvinden.

a. Vervang de `ArrayList` in Dienblad door een `Stack<Artikel>`. Je kunt in Java niet direct een `Queue<Persoon>` aanmaken omdat dat een interface is (later tijdens dit thema leer wat dat precies betekent). Gebruik een `LinkedList<Dienblad>` in `Kassarij` om de `ArrayList<Dienblad>` te vervangen.

Opgave 4: Refactoren – dubbele methodes en het gebruik van een iterator

Als je goed kijkt naar de code van de eerste versie van de kantine simulatie valt je misschien op dat er soms methodes zijn die twee keer voorkomen. Eén van die twee methodes is slechts een soort doorgeefluik. Dit kun je efficiënter oplossen.

a. Bij welke methodes in `Kassa` en `Kantine` komt dit voor?

b. Verwijder deze methodes in `Kantine`. Maak een getter voor de private instantie variabele kassa in de klasse `Kantine`.

c. Als je je project nu compileert krijg je een foutmelding in de klasse `KantineSimulatie`. Los dit op door de getter uit de vorige vraag te gebruiken.

Een ander probleem komt in de klassen `Dienblad` en `Kassa` voor. Je kunt namelijk terecht opmerken dat de klasse `Dienblad` de methoden `double getTotaalPrijs()` en `int getAantalArtikelen()` helemaal niet zou moeten bevatten. Immers, `Dienblad` is niks anders dan een soort container voor klanten. Het is beter dat de klasse `Kassa` via een methode in de klasse `Dienblad` een `Iterator<Artikel>` ophaalt waarmee door de artikelen op het dienblad heen gelopen kan worden. Zo kan de klasse `Kassa` zelf de totaalprijs en het aantal artikelen berekenen.

d. Pas de code van de klassen `Dienblad` en `Kassa` aan.

Opgave 5: De klasse `KantineAanbod`

Je krijgt van ons de klasse `KantineAanbod` cadeau; deze kun je van Blackboard halen. Voeg deze klasse aan je project toe en zorg dat je de code goed begrijpt.

a. Leg uit waarom het goed is om de methodes `ArrayList<Artikel> getArrayList(String productnaam)` en `Artikel getArtikel(ArrayList<Artikel>)` *private* te maken.

b. In welke situatie gebruik je een `HashMap` en wanneer een `HashSet`?

c. Voeg een instantievariabele kantineaanbod van het type `KantineAanbod` toe aan de klasse `Kantine`. Voeg ook een *getter* en *setter* voor deze variabele toe.

In de eerste versie van de kantinesimulatie maakt de methode `loopPakSluitAan()` in `Kantine` zelf een `Persoon` en een `Dienblad` aan, om vervolgens twee Artikelen te pakken. In de nieuwe versie van de kantinesimulatie willen we dat de klasse `KantineSimulatie` zelf een persoon met een dienblad aanlevert, samen met een lijst van artikelnamen die uit het kantineaanbod moeten worden gehaald. Kortom, de signatuur van `loopPakSluitAan()` verandert in:

```
/**
 * In deze methode wordt een dienblad met artikelen
 * in de kassarij geplaatst.
 *
 * @param dienblad
 */
public void loopPakSluitAan(Dienblad dienblad, String[] artikelnamen) {
    // method body omitted
}
```

d. Implementeer bovenstaande methode.

Opgave 6: Refactoren van `Kantine` en `KantineSimulatie`

Hieronder staat de skeletcode voor een nieuwe versie van de `KantineSimulatie`. We gaan ervan uit dat er vier verschillende artikelen zijn waarbij de hoeveelheid via de klasse `java.util.Random` wordt bepaald.

```
import java.util.*;

public class KantineSimulatie {

    // kantine
    private Kantine kantine;

    // kantineaanbod
    private KantineAanbod kantineaanbod;
```

```
10
11      // random generator
12      private Random random;
13
14      // aantal artikelen
15      private static final int AANTAL_ARTIKELEN = 4;
16
17      // artikelen
18      private static final String[] artikelnamen = new String[]
19          {"Koffie", "Broodje pindakaas", "Broodje kaas", "Appelsap"};
20
21      // prijzen
22      private static double[] artikelprijzen = new double[]{1.50, 2.10,
            1.65, 1.65};
23
24      // minimum en maximum aantal artikelen per soort
25      private static final int MIN_ARTIKELEN_PER_SOORT = 10000;
26      private static final int MAX_ARTIKELEN_PER_SOORT = 20000;
27
28      // minimum en maximum aantal personen per dag
29      private static final int MIN_PERSONEN_PER_DAG = 50;
30      private static final int MAX_PERSONEN_PER_DAG = 100;
31
32      // minimum en maximum artikelen per persoon
33      private static final int MIN_ARTIKELEN_PER_PERSOON = 1;
34      private static final int MAX_ARTIKELEN_PER_PERSOON = 4;
35
36      /**
37       * Constructor
38       *
39       */
40      public KantineSimulatie() {
41          kantine = new Kantine();
42          random = new Random();
43          int[] hoeveelheden = getRandomArray(
44              AANTAL_ARTIKELEN,
45              MIN_ARTIKELEN_PER_SOORT,
46              MAX_ARTIKELEN_PER_SOORT);
47          kantineaanbod = new KantineAanbod(
48              artikelnamen, artikelprijzen, hoeveelheden);
49
50          kantine.setKantineAanbod(kantineaanbod);
51      }
52
53      /**
54       * Methode om een array van random getallen liggend tussen
55       * min en max van de gegeven lengte te genereren
56       *
57       * @param lengte
58       * @param min
59       * @param max
```

```
60          * @return De array met random getallen
61          */
62         private int[] getRandomArray(int lengte, int min, int max) {
63             int[] temp = new int[lengte];
64             for (int i = 0; i < lengte ;i++) {
65                 temp[i] = getRandomValue(min, max);
66             }
67
68             return temp;
69         }
70
71         /**
72          * Methode om een random getal tussen min(incl)
73          * en max(incl) te genereren.
74          *
75          * @param min
76          * @param max
77          * @return Een random getal
78          */
79         private int getRandomValue(int min, int max) {
80             return random.nextInt(max - min + 1) + min;
81         }
82
83         /**
84          * Methode om op basis van een array van indexen voor de array
85          * artikelnamen de bijhorende array van artikelnamen te maken
86          *
87          * @param indexen
88          * @return De array met artikelnamen
89          */
90         private String[] geefArtikelNamen(int[] indexen) {
91             String[] artikelen = new String[indexen.length];
92
93             for (int i = 0; i < indexen.length; i++) {
94                 artikelen[i] = artikelnamen[indexen[i]];
95
96             }
97
98             return artikelen;
99         }
100
101         /**
102          * Deze methode simuleert een aantal dagen
103          * in het verloop van de kantine
104          *
105          * @param dagen
106          */
107         public void simuleer(int dagen) {
108             // for lus voor dagen
109             for (int i = 0; i < dagen; i++) {
110
```

```
111             // bedenk hoeveel personen vandaag binnen lopen
112             int aantalpersonen = ... ;
113
114             // laat de personen maar komen...
115             for (int j = 0; j < aantalpersonen; j++) {
116
117                 // maak persoon en dienblad aan, koppel ze
118                 // en bedenk hoeveel artikelen worden gepakt
119                 int aantalartikelen = ... ;
120
121                 // genereer de "artikelnummers", dit zijn indexen
122                 // van de artikelnamen
123                 array int[] tepakken = getRandomArray(
124                     aantalartikelen, 0, AANTAL_ARTIKELEN-1);
125
126                 // vind de artikelnamen op basis van
127                 // de indexen hierboven
128                 String[] artikelen = geefArtikelNamen(tepakken);
129
130                 // loop de kantine binnen, pak de gewenste
131                 // artikelen, sluit aan
132
133             }
134
135             // verwerk rij voor de kassa
136
137             // druk de dagtotalen af en hoeveel personen binnen
138
139             // zijn gekomen
140
141             // reset de kassa voor de volgende dag
142
143         }
144     }
145 }
```

a. Leg de werking van de constructor uit.

b. Leg de implementatie van `int getRandomValue(int min, int max)` uit en met name waarom er +1 in voorkomt. Gebruik de Java API.[5]

Implementeer de ontbrekende delen van de code van de tweede versie van de kantine-simulator. Roep de methode `simuleer(int dagen)` aan.

[5] Hint: denk aan de betekenis van inclusief en exclusief.

Week 3: Voorraadbeheer en administratie

Als je in de gegeven code voor de kantinesimulatie kijkt zie je dat de voorraden artikelen altijd boven de 10.000 liggen als de simulatie begint. Verander de minimum en maximum waarden voor de hoeveelheid artikelen naar 10 en 20, en kijk wat er gebeurt als je de simulatie opnieuw start.

Opgave 1: Aanvullen van de voorraden

In de huidige simulatie is er geen mogelijkheid om de voorraden aan te vullen. Pas je code aan zodat als je onder een bepaald minimum voorraad komt de voorraad weer tot het beginniveau wordt aangevuld. Implementeer hiervoor de methode `vulVoorraadAan(String productnaam)` in de klasse `KantineAanbod` en zorg er voor dat deze methode wordt aangeropen iedere keer als er een artikel wordt opgehaald en daarmee de voorraad onder het minimum komt.

Opgave 2: De klasse Administratie

Hieronder zie je de skeletcode voor de klasse `Administratie`. Deze klasse wordt later gebruikt om kassagegevens uit te lezen en een paar statistische berekeningen uit te voeren. De arrays die als parameter worden gebruikt in de methoden worden later aangeleverd door een `KantineSimulatie` klasse die de kantine over een periode van bijvoorbeeld dertig dagen simuleert. Elke dag levert twee metingen op: het aantal gepasseerde artikelen en de omzet.

a. Implementeer deze klasse. Maak je implementatie wel flexibel; ga dus niet uit van arrays met een omvang van dertig elementen.

```
1  public class Administratie {
2
3      /**
4       * Deze methode berekent van de int array aantal de gemiddelde
            waarde
5       *
6       * @param aantal
7       * @return het gemiddelde
8       */
9      public double berekenGemiddeldAantal(int[] aantal) {
10         // method body omitted
11     }
12
13     /**
14      * Deze methode berekent van de double array omzet de gemiddelde
            waarde
15      *
16      * @param omzet
```

```
17       * @return het gemiddelde
18       */
19      public double berekenGemiddeldeOmzet(double[] omzet) {
20          // method body omitted
21      }
22  }
```

Het gemiddelde van een rij getallen is de som van de rij getallen gedeeld door het aantal getallen. Het gemiddelde van een lege rij getallen is 0.

b. Test de methodes met onderstaand verwacht resultaat:[6]

Methode	Input	Verwacht resultaalt
`berekenGemiddeldAantal`	`{45, 56, 34, 39, 40, 31}`	40.8333
`berekenGemiddeldeOmzet`	`{567.70, 498.25, 458.90}`	508.2833

c. Er is geen constructor gedefinieerd voor `Administratie` terwijl je gewoon `new Administratie()` kan aanroepen. Leg uit waarom dat kan.

d. Leg uit waarom de twee al bestaande methoden van `Administratie` static kunnen zijn. Verander ze in static.

e. We hebben door het static maken van de twee methodes geen instantie meer nodig van `Administratie`. Het is echter wel mogelijk om een instantie van `Administratie` aan te maken en daar de static methoden op aan te roepen. Als je dat wil voorkomen kun je een private constructor voor `Administratie` maken. Doe dat en leg uit waarom je je doel nu bereikt.

De mensen op de administratie willen naast de gemiddelden van het aantal verkochte artikelen en de omzet over een periode een nieuw overzicht zien. Ze zijn geïnteresseerd in de dagtotalen over een periode van de omzet, dat wil zeggen naast het gemiddelde over de hele periode willen ze zeven totalen over de periode, voor elke dag één. Een voorbeeld: stel dat de omzet per dag volgens de onderstaande array verloopt:

```
1  {321.35, 450.50, 210.45, 190.85, 193.25,
2   159.90, 214.25, 220.90, 201.90, 242.70, 260.35}
```

We tellen gemakshalve vanaf nul, omdat in Java dat ook gebeurt. Je mag er van uit gaan dat het „nulde" element van de array de omzet op maandag is, het eerste dinsdag, enzovoort. Na zeven dagen, dus

[6] Hint bij `berekenGemiddeldAantal`: wat gebeurt er als je twee ints deelt? Los het probleem op door te casten naar een double.

vanaf het zesde element, begin je weer van voor af aan: omzet op maandag, de volgende is dinsdag.

We gaan er gemakshalve vanuit dat de kantine zeven dagen per week open is. De totaalomzet op maandag is 321.35 + 220.90, die van dinsdag is 450.50 + 201.90. Merk op dat in dit voorbeeld vrijdag, zaterdag en zondag maar één keer voorkomen en daarmee de totaalomzet op die dagen gelijk is aan de behaalde omzet op die dagen.

f. Voeg een static methode toe aan `Administratie` met bovenstaande functionaliteit met onderstaande signatuur. Vul de code aan.

```java
/**
 * Methode om dagomzet uit te rekenen
 *
 * @param omzet
 * @return array (7 elementen) met dagomzetten
 */

public static double[] berekenDagOmzet(double[] omzet) {
    double[] temp = new double[7];
    for (int i = 0; i < 7; i++) {

        int j = 0;
        while ( ... ) {
            temp[i] += omzet[i + 7 * j];

            // omitted

        }
    }
    return temp;
}
```

g. In plaats van dat je de „magic constant" 7 gebruikt in de implementatie van `berekenDagomzet(double[] omzet)` kun je ook een `final int days_in_week` als een private instantievariabele toevoegen. Pas je code aan. Leg uit wat final doet.

h. Als het goed is klaagt de compiler over zoiets als „Cannot make a static reference to the non-static field …". Leg uit waarom de compiler hierover klaagt.

i. Een manier om het probleem te verhelpen is om het woord final te vervangen door static. Waarschijnlijk compileert het werk nu wel weer, maar is het niet meer goed. Welk „probleem" heb je nu geïntroduceerd? Hint: wat was nou ook alweer de oorspronkelijke aanleiding om `days_in_week` te introduceren?

j. Voeg nu alsnog final toe en vervang `days_in_week` door `DAYS_IN_WEEK`. Het is een conventie in Java om de naam van static final variabelen met hoofdletters te schrijven.

Opgave 3: Overerving met `Student`, `Docent` en `KantineMedewerker`

In deze opgave maken we drie subklassen van `Persoon`: `Student`, `Docent` en `KantineMedewerker`. De verschillen staan hier onder opgesomd:

- Een student heeft een studentnummer en volgt een studierichting;
- Een docent heeft een vierletterige afkorting en werkt bij een afdeling;
- Een `Kantinemedewerker` heeft een medewerkersnummer en een boolean waarde die aangeeft of hij/zij achter de kassa mag staan.

a. Maak de bovenstaande drie subklassen van `Persoon`, met constructors en getters en setters. Zorg dat in alle drie constructors alle gegevens staan en gebruik een super aanroep in de constructor.

b. Waarom moet een super aanroep in de constructor altijd bovenaan staan?

Opgave 4: Kantine Simulatie

a. Pas de kantine simulatie van vorige week zodanig aan dat in plaats van een random aantal objecten van het type `Persoon` de volgende drie type objecten met de genoemde hoeveelheden worden aangemaakt (totaal dus 100 objecten):

- Een student: 89 instanties;
- Een docent: 10 instanties;
- Een kantinemedewerker: 1 instantie.

Let op: alledrie typen klassen spelen de rol van klant, dat wil zeggen dat de kantinemedewerker niet achter de kassa staat maar zelf artikelen koopt.

b. Gebruik de methode `toString()` in je simulatie zodat je ziet wat voor type persoon de kantine binnenkomt.

c. Roep de drie methodes van `Administratie` aan en druk het resultaat af.

Opgave 5: Random soorten bezoekers

In plaats van dat je in opgave 4a) een vast aantal bezoekers van de kantine maakt is het realistischer om het aantal bezoekers nog steeds random te genereren – zoals in week 2 – en de opgegeven aantallen te interpreteren als kansen:

- Een student wordt aangemaakt met een kans 89 op 100;
- Een docent wordt aangemaakt met een kans 10 op 100;
- Een kantinemedewerker wordt aangemaakt met een kans 1 op 100.

Maak hiervoor gebruik van het resultaat van de instructie `random.nextInt(100)` en controleer of de waarde in een bepaalde range ligt.

Week 4: Overerving en foutafhandeling

Opgave 1: Betalen doe je zo

In deze opgave ga je de daadwerkelijke betaling van een persoon implementeren. Een betaling kan op twee manieren gebeuren: *contant* of met een *pinpas*. In principe kun je een interface `Betaalwijze` introduceren, maar bij beide betaalwijzen zou je wel het tegoed kunnen opslaan. We gaan ervan uit dat een betaling met een pinpas gebeurt vanaf een rekening waar een kredietlimiet op zit. Kredietlimieten bestaan niet voor een contante betaling, tenminste de bodem van je portemonnee is de kredietlimiet. Kortom, we introduceren een abstracte klasse `Betaalwijze` als volgt:

```
1  public abstract class Betaalwijze {
2
3      protected double saldo;
4
5      /**
6       * Methode om krediet te initialiseren
7       * @param saldo
8       */
9      public void setSaldo(double saldo) {
10         this.saldo = saldo;
11     }
12
13     /**
14      * Methode om betaling af te handelen
15      *
16      * @param tebetalen
17      * @return Boolean om te kijken of er voldoende saldo is
18      */
19     public abstract boolean betaal(double tebetalen);
20 }
```

Daarnaast zijn er twee concrete subklassen `Contant` en `Pinpas` die deze abstracte superklasse extenden.

```
1  public class Contant extends Betaalwijze {
2      /**
```

```
3       * Methode om betaling af te handelen
4       */
5      public boolean betaal(double tebetalen) {
6          // method body omitted
7      }
8  }
```

en

```
1  public class Pinpas extends Betaalwijze {
2
3      private double kredietlimiet;
4
5      /**
6       * Methode om kredietlimiet te zetten
7       * @param kredietlimiet
8       */
9      public void setKredietLimiet(double kredietlimiet) {
10         // method body omitted
11     }
12
13     /**
14      * Methode om betaling af te handelen
15      */
16     public boolean betaal(double tebetalen) {
17         // method body omitted
18     }
19 }
```

Merk trouwens op dat een `Persoon` een referentie heeft naar een abstracte klasse `Betaalwijze`. Dit is een „heeft-een" relatie, dus er is *geen* overervingsstructuur tussen `Persoon` en `Betaalwijze`. Je zou inderdaad niet kunnen beweren dat een persoon een betaalwijze is, maar wel dat een persoon een betaalwijze heeft.

a. Implementeer deze drie klassen.

b. Stel dat bij de pinpasbetaling zou worden gecommuniceerd met een `Bank` object. Teken een sequentie-diagram waaruit blijkt hoe je de methode `boolean betaal(double tebetalen)` in `Pinpas` implementeert.[7]

c. Waarom is de instantie variabele saldo protected gemaakt? Waarom is dat handig?

d. Maak een private instantie variabele `betaalwijze` in `Persoon`. Maak een getter en een setter. Aanpassen van de constructor is niet nodig.

e. Pas de code in `Kassa` aan zodat bij de betaling naar de betaalwijze van de persoon wordt gevraagd en de methode `betaal(double tebetalen)` wordt aangeroepen. Indien de betaling

[7] Hint: vergeet niet hoe de klasse `Pinpas` een referentie naar een `Bank` object heeft gekregen.

faalt laat dan een melding zien op het scherm.

f. Pas de code in de methode `rekenAf` in de klasse `Kassa` aan zodat bij de betaling naar de betaalwijze van de persoon wordt gevraagd en de methode `betaal(double tebetalen)` wordt aangeroepen. Indien de betaling faalt laat dan een melding zien op het scherm. De opbrengst mag natuurlijk ook alleen opgehoogd worden als de betaling gelukt is.

Opgave 2: Kortingskaarthouder

De docenten en kantinemedewerkers zijn in de gelukkige omstandigheid dat ze korting krijgen op de aangeschafte artikelen, voor docenten is dat 25%, met een maximale korting van 1.00 Euro per kantinebezoek en kantinemedewerkers krijgen zelf 35% korting, zonder een maximumbedrag. Dit is te realiseren door de onderstaande interface te definiëren:

```
1  public interface KortingskaartHouder {
2
3      /**
4       * Methode om kortingspercentage op te vragen
5       */
6      public double geefKortingsPercentage();
7
8      /**
9       * Methode om op te vragen of er maximum per keer aan de korting
            zit
10      */
11     public boolean heeftMaximum();
12
13     /**
14      * Methode om het maximum kortingsbedrag op te vragen
15      */
16     public double geefMaximum();
17 }
```

Doordat je zou kunnen zeggen dat docenten en kantinemedewerkers kortingskaarthouders zijn is het verstandig om de `Docent` en `KantineMedewerker` klasse deze interface te laten implementeren.

a. Pas de `Docent` en `KantineMedewerker` klasse aan zodanig dat ze de interface `KortingskaartHouder` implementeren.

b. Pas de code in de klasse `Kassa` aan zodat er gecheckt wordt of een `Persoon` ook een kortingskaart heeft (in objecttermen kortingskaarthouder is) en daar rekening mee wordt gehouden bij de betaling.[8]

[8] Hint: gebruik de **`instanceof`** operator en casting.

Opgave 3: Theorie over (abstracte) klassen en interfaces

a. Kun je een instantie maken van een *interface* via **new**? Leg uit waarom het logisch is dat het wel of niet kan.

b. Herhaal de vorige vraag met *abstracte* klassen.

c. Kan een klasse meerdere *klassen* overerven?

d. Kan een klasse meerdere *interfaces* implementeren?

e. Kan een klasse tegelijk een klasse overerven en interfaces implementeren?

f. Klopt de stelling dat elke methode in een interface abstract is? Licht je antwoord toe.

g. Moet een klasse abstract zijn als minstens één methode abstract is? Licht je antwoord toe.

h. Leg het begrip *polymorfisme* van klassen uit en geef twee voorbeelden (één met abstracte klassen en één met interfaces).

Opgave 4: Een paar doordenkers

a. Kan een klasse abstract zijn als geen enkele methode abstract is in die klasse? Probeer het eens uit. Leg waarom het logisch is dat dit wel of niet kan.

b. Moet een subklasse van een abstracte klasse altijd alle abstracte methodes implementeren? Leg uit waarom het logisch is dat dit wel of niet kan.

c. Als een klasse niet alle methoden van een interface implementeert kun je iets doen om een (compiler)fout te voorkomen. Wat? Waarom is de oplossing logisch?

d. Leg uit waarom het logisch is dat een instantie variabele niet abstract kan zijn.

e. (Uitdaging) Zoek uit wat een *final* methode is. Leg daarna uit waarom het logisch is dat een methode niet tegelijkertijd abstract en final kan zijn.

Opgave 5: `TeWeinigGeldException` maken

In opgave 2 van vorige week heb je een abstracte klasse met twee concrete subklassen gemaakt. De methode `betaal(double tebetalen)` geeft een **boolean** terug als indicatie dat de betaling wel of niet is gelukt.

a. Maak een eigen `TeWeinigGeldException`, met drie constructors:

- `TeWeinigGeldException()`

- `TeWeinigGeldException(Exception e)`
- `TeWeinigGeldException(String message)`

b. Verander het return type van de methode `betaal(double tebetalen)` in **`void`** en voeg een **`throws`** declaratie toe waarbij een `TeWeinigGeldException(String message)` wordt gethrowd als er onvoldoende saldo is. Als de betaalmethode een `TeWeinigGeldException` gooit wordt deze gevangen in `KantineSimulatie`. Zorg dat in het catch-blok de naam van diegene die te weinig geld heeft wordt afgedrukt.

c. Pas de code in `Kassa` aan zodat de `TeWeinigGeldException` wordt gevangen.

Week 5: Data persistentie

De data leeft zolang de simulatie loopt maar het zou mooi zijn als we resultaten kunnen opslaan om ze op een later tijdstip weer op te kunnen halen. In deze week ga je een begin maken met het opzetten van een database voor het opslaan van gegevens en zal je zien hoe je Object Relational Mapping (ORM) kan toepassen op jouw project.

Op Blackboard vind je een voorbeeldproject waar gebruik is gemaakt van de Java Persistence API (JPA) voor het „mappen" van objectdata naar een database (ORM) en hoe relaties tussen objecten onderling kunnen worden opgeslagen.[9] Het doel is dit project te gaan verkennen en bestuderen om in de laatste week de technieken die je tegenkomt zelf te gaan toepassen.

Opgave 1: Opzet en configuratie

a. Download het voorbeeldproject **JPAVoorbeeld** van Blackboard en importeer het in jouw IDE.

b. Zorg er voor dat een database server is opgestart en maak een nieuwe database aan voor dit project, bijvoorbeeld met de naam `jpavoorbeeld`.[10]

De configuratie kan je vinden in het bestand `src/main/resources/META-INF/persistence.xml` en voor de connectie zijn de volgende regels van belang:

```
1 <!-- The JDBC driver for your database -->
2 <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.
     Driver" />
3 <!-- The JDBC URL to the database instance -->
```

[9] De Java Persistence API is een *specificatie*. Dit wil zeggen dat het alleen maar een een aantal regels opstelt die moeten worden *geïmplementeerd*. In het voorbeeld wordt *Hibernate* gebruikt, een project dat deze regels implementeert. Zie https://www.baeldung.com/jpa-hibernate-difference voor een kort overzicht.

[10] Je zal waarschijnlijk nog MySQL geïnstalleerd hebben, maak daar een nieuw schema aan.

```
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://
    localhost:3306/javavoorbeeld" />
<!-- The database username -->
<property name="javax.persistence.jdbc.user" value="username" />
<!-- The database password -->
<property name="javax.persistence.jdbc.password" value="password" />
```

c. Pas in de configuratie (waar nodig) de volgende velden aan om de verbinding met jouw database te kunnen maken:

   - de connectie url
   - username en wachtwoord

De configuratie gaat er van uit dat je een MySQL database gebruikt, mocht dit niet het geval zijn dan zal je een andere *driver* moeten instellen voor `javax.persistence.jdbc.driver`.[11]

In het project vind je de klassen `Student`, `StudieInschrijving`, `StudentKaart` en `Telefoon`. `Student` is vergelijkbaar met de klasse `Student` in de kantinesimulatie met het belangrijkste verschil dat de Student in dit geval één of meerdere `Telefoon` en `StudieInschrijving` objecten heeft. Verder kan een Student natuurlijk maar één `Studentkaart` hebben.

Je zal ook zien dat in deze klassen Java *annotaties* worden gebruikt, een techniek om *metadata* aan klassen toe te voegen. Deze annotaties doen op zich niets maar geven extra informatie die kan worden gebruikt in een andere context, bijvoorbeeld door JPA als objecten moeten worden opgeslagen in een database.[12] Zie bijvoorbeeld het volgende fragment van de klasse `Telefoon` voor het gebruik van deze annotaties:

```
import javax.persistence.Id;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
public class Telefoon {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "nummer")
    private String nummer;

```

[11] Er zijn veel alternatieve databases waar je misschien al mee hebt geëxperimenteerd en het is waarschijnlijk dat daar ook een JDBC driver voor beschikbaar is. Je zal deze driver als afhankelijkheid dan ook moeten opnemen in de Maven configuratie `pom.xml`.

[12] Zie https://docs.oracle.com/javase/tutorial/java/annotations/ voor een korte inleiding in Java annotaties.

```
16      @Column(name = "type")
17      private String type;
18
19      public Telefoon() {
20          // ...
21      }
22  }
```

d. Lees de klassen `Student`, `StudieInschrijving` en `Telefoon` door. Zou je al iets kunnen vertellen over het doel van de annotaties `@Id`, `@GeneratedValue` en `@Column`?

In de klasse `Main` vind je de methode `runVoorbeeld` waar een aantal objecten worden aangemaakt, opgeslagen en verwijderd. Voer de methode `main` uit (die de methode `runVoorbeeld` aanroept). Controleer vervolgens of gegevens in de database zichtbaar zijn.

e. Een *viertal* tabellen zullen nu in de database zijn aangemaakt, welke zijn deze?

Opgave 2: Relaties

De objecten in dit voorbeeld hebben relaties met elkaar, zo heeft een Student een StudentKaart en één of meerdere StudieInschrijving(en). Je zal onderhand bekend zijn met *One-to-One*, *One-to-Many* etc. relaties in databases en hoe deze relaties in code (met JPA) zijn uit te drukken zal je in deze opgave gaan verkennen.

a. Het attribuut `telefoons` in de klasse `Student` is geannoteerd met `@OneToMany`. Waarom zou hier `@OneToMany` en niet `@OneToOne` zijn gebruikt?

b. Het attribuut `studies` in de klasse `Student` is ook geannoteerd met `@OneToMany`, dit omdat een student natuurlijk bij meerdere studies ingeschreven kan staan. In tegenstelling tot `telefoons` wordt hier geen `@JoinTable` annotatie gebruikt.

   - Hoe wordt het verschil zichtbaar in de database? Let hier op de tabellen die zijn aangemaakt en de velden in de tabellen voor een Student en StudieInschrijving.
   - Zou je een nadeel kunnen bedenken waarom voor de relatie `Student` en `StudieInschrijving` via `studies` *geen* `@JoinTable` is gebruikt?

c. In de klasse `StudieInschrijving` is het attribuut `student` met `@ManyToOne` geannoteerd. Beschrijf hoe deze relatie in de database zichtbaar wordt.

d. De klasse `StudentKaart` is geannoteerd met `@Embeddable` en het attribuut `kaart` in de klasse `Student` met `@Embedded`.

   - Waar vind je een `StudentKaart` terug in de database?

- Wat zou je hieruit kunnen afleiden met betrekking tot het gebruik en de functie van `@Embeddable` en `@Embedded`?

Opgave 3: Transacties

De meeste databases kennen het concept van een *transactie* waar meerder queries worden gegroepeerd en als één operatie worden uitgevoerd. Mocht ergens een fout optreden dan kunnen alle queries weer ongedaan worden gemaakt.

Zoek in de klasse `Main` een voorbeeld van het gebruik van een transactie. Je zal zien dat gebruik wordt gemaakt van een *try-catch* blok om een transactie bij een fout af te breken en terug te draaien (*roll-back*).

a. Een transactie wordt ook wel een *unit of atomicity* genoemd, waarbij atomiciteit in de context van databases „alles of niets" betekent. Vooruitkijkend op week 6, kan je voor de kantinesimulatie een situatie bedenken waar je dit patroon zou kunnen toepassen?[13]

Opgave 4: Voorbereiding

Nu je hebt gezien hoe je ORM kan gebruiken wordt het tijd om dit aan de simulatie toe te voegen. De opgaven van volgende week gaan over de *implementatie*, nu ga je daar alvast voorbereidingen voor treffen door een configuratie aan te maken en een *entity manager* aan de klasse `KantineSimulatie` toe te voegen.

a. Maak een nieuwe database aan voor de simulatie met de naam *kantine*. Mocht je een andere naam willen kiezen, dan zal je waarde van `javax.persistence.jdbc.url` moeten aanpassen in `src/main/resources/META-INF/persistence.xml`.

Als laatste ga je een entity manager aan de simulatie toevoegen. In het voorbeeldproject zie je in `Main` het volgende:

```
1  import javax.persistence.Persistence;
2  import javax.persistence.EntityManager;
3  import javax.persistence.EntityManagerFactory;
4
5  public class Main {
6      private static final EntityManagerFactory ENTITY_MANAGER_FACTORY =
7          Persistence.createEntityManagerFactory("KantineSimulatie");
8      private EntityManager manager;
9
10     public void runVoorbeeld() {
```

[13] Denk aan waar je exceptions hebt toegepast.

```
11          manager = ENTITY_MANAGER_FACTORY.createEntityManager();
12
13          // transactions omitted
14
15          manager.close();
16          ENTITY_MANAGER_FACTORY.close();
17      }
18  }
```

b. Voeg zoals in het bovenstaande fragment een entity manager toe aan de klasse `KantineSimulatie`.

Opgave 5: Bijzondere prijzen

Voor we verder kunnen is nog één bijzondere situatie waar rekening mee moet worden gehouden want bij de kassa hangt namelijk de volgende waarschuwing:

> **Op dagaanbiedingen is geen medewerkerskorting van toepassing.**

De kantine heeft blijkbaar dagaanbiedingen waarbij één of meerdere artikelen in prijs zijn verlaagd. We moeten dus gaan letten op welke artikelen wél en op welke géén korting van toepassing is bij afrekenen naast de algemene korting op basis van een kortingkaart!

a. Pas de klasse `Artikel` aan met een `korting` veld en maak de bijbehorende *getter* en *setter*. `korting` is een concreet bedrag en *niet* een percentage.

b. Maak een constructor aan die naast naam en prijs ook korting als parameter heeft. Houdt er verder rekening mee dat korting een standaard waarde van 0 euro heeft indien geen korting wordt doorgegeven als constructor parameter.

c. Zorg in de simulatie bij het aanmaken van de artikelen dat dagelijks tenminste één artikel in de aanbieding is – je zal hier dus weer een randomisatie moeten toepassen. De korting op een dagaanbieding is 20% van de prijs van het geselecteerde artikel.

d. De verantwoordelijkheid voor het berekenen van het totaalbedrag heb je eerder al verplaatst naar de klasse `Kassa` in de methode `rekenAf`. Pas deze methode aan zodat:

   - de korting op dagaanbiedingen wordt toegepast
   - de korting voor kortingkaarthouders wordt toegepast, uitgezonderd op dagaanbiedingen

Week 6: Bonnetjes en optellingen

In deze laatste week ga je de simulatie afronden en ga je gebruik maken van een database om data in de simulatie op te slaan. Je zal hier gebruik maken van JPA/Hibernate waar je in de vorige week al kennis mee hebt gemaakt.

Opgave 1: De klasse `Factuur`

In de simulatie heb je tot nu toe gewerkt met totalen, bijvoorbeeld de omzet die in een week is gegenereerd. Om een begin te maken met het bijhouden van individuele aankopen gaan we facturen bijhouden, oftewel het aanmaken van een *kassabon.

Op een factuur komt natuurlijk ook een datum en we zouden onze `Datum` klasse hier voor kunnen gebruiken, maar zullen dan tegen een probleem aanlopen als we deze waarde in de database willen opslaan. We hebben namelijk nergens aangegeven met welk database type ons objecttype `Datum` correspondeert en Hibernate zal het daarom opslaan als een `bytea` (byte array) type waar we verder niet veel mee kunnen doen (bijvoorbeeld queries op toepassen). We gaan om deze reden de klasse `Datum` vaarwel zeggen en het standaardtype `java.time.LocalDate` gebruiken dat Hibernate wél kent en kan *serialiseren* naar het database type `DATE`.

Voor het eenvoudig aanmaken van een `LocalDate` object kan je de volgende constructie gebruiken:[14]

```
1  LocalDate datum = LocalDate.to(2019, 5, 16);
```

De logica voor de afhandeling van een bestelling vindt plaats in de klasse `Kassa` in de methode `rekenAf`. Deze methode gaat nu een Factuur aanmaken voor elke klant. De volgende begincode is gegeven voor de nieuwe klasse `Factuur`:

```
1  import java.time.LocalDate;
2  import java.io.Serializable;
3
4  public class Factuur implements Serializable {
5
6      private Long id;
7
8      private LocalDate datum;
9
10     private double korting;
11
```

[14] Mocht je meer precisie in jouw simulatie nodig hebben dan kan je ook het type `java.time.LocalDateTime` gebruiken om ook uur, minuten en seconden vast te leggen. `LocalDateTime` zal corresponderen met het `DATETIME` database type.

```
12      private double totaal;
13
14      public Factuur() {
15          totaal = 0;
16          korting = 0;
17      }
18
19      public Factuur(Dienblad klant, LocalDate datum) {
20          this();
21          this.datum = datum;
22
23          verwerkBestelling(klant);
24      }
25
26      /**
27       * Verwerk artikelen en pas kortingen toe.
28       *
29       * Zet het totaal te betalen bedrag en het
30       * totaal aan ontvangen kortingen.
31       *
32       * @param klant
33       */
34      private void verwerkBestelling(Dienblad klant) {
35          // method body omitted
36      }
37
38      /*
39       * @return het totaalbedrag
40       */
41      public double getTotaal() {
42          return totaal;
43      }
44
45      /**
46       * @return de toegepaste korting
47       */
48      public double getKorting() {
49          return korting;
50      }
51
52      /**
53       * @return een printbaar bonnetje
54       */
55      public String toString() {
56          // method body omitted
57      }
58  }
```

a. Verplaats de berekening van het totaal en de toegepaste korting in `rekenAf` naar de methode `verwerkBestelling` in de klasse `Factuur`. Let op: betalingen blijven door de Kassa uitge-

voerd worden, dit is niet de verantwoordelijkheid van een Factuur.

b. Maak voor elke afrekening nu een Factuur object aan en gebruik de getters van `Factuur` om het totaalbedrag en de toegepaste korting van de Kassa te verhogen.

De klasse `Factuur` is nog een reguliere klasse en JPA/Hybernate weet niet dat het een entiteit is die zij moet gaan beheren:

c. *Annoteer* de klasse `Factuur` zodat deze door een entitymanager beheerd kan worden. In *JPA-Voorbeeld* kan je geschikte voorbeelden vinden, bedenk dat je nog *niet* met complexe (many-to-one, etc.) relaties rekening hoeft te houden.

d. Implementeer `toString` zodat je in de simulatie ook een bonnetje kan printen.

Opgave 2: Facturen opslaan

Het voorbereidend werk nu is gedaan en ondanks de toevoeging van een Factuur bij de afrekening zal de simulatie nog steeds werken. De refactoring was nodig want in deze opgave gaan we de factuur daadwerkelijk opslaan in een database.

Je hebt ter voorbereiding in de klasse `KantineSimulatie` een `EntityManager` object aangemaakt. Zoals je in *JPAVoorbeeld* hebt kunnen zien wordt deze gebuikt voor beheren en uitvoeren van databasetransacties. De klasse `Kassa` heeft hier nog geen weet van en we moeten dit manager object nu gaan doorgeven.

a. Voeg aan zowel de klasse `Kantine` als de klasse `Kassa` een veld `manager` toe van het type `javax.persistence.EntityManager`

b. Pas de constructors van `Kantine` en `Kassa` aan zodat het een EntityManager object als parameter accepteert en het veld `manager` in beide klassen wordt geïnitialiseerd met deze waarde.

c. Zorg bij het aanmaken van een Kassa in de klasse `Kantine` dat het `manager` object wordt doorgegeven.

De klasse `Kassa` heeft nu een EntityManager en deze kan worden gebruikt voor het opzetten van een databasetransactie om een factuur op te slaan. Natuurlijk willen we een factuur pas opslaan als de betaling succesvol is verlopen, indien dit niet slaagt zal de transactie ongedaan moeten worden gemaakt.

d. Gebruik de EntityManager om een factuur op te slaan door middel van een transactie in de methode `rekenAf` in `Kassa`. Breek de transactie af als de betaling niet is geslaagd (een *rollback*). Maak hier gebruik van de `TeWeinigGeldException` om te bepalen of een betaling wel of niet is geslaagd.

Opgave 3: Totalen en gemiddelden

Nu data in de database aanwezig is kunnen ook queries worden opgesteld en uitgevoerd in `KantineSimulatie` zodra de simulatie heeft plaatsgevonden. Wederom vind je in het *JPAVoorbeeld* project hier voorbeelden van.

a. Maak een query om de totale omzet en toegepaste korting op te vragen, voer deze uit en print het resultaat.

b. Maak een query om de gemiddelde omzet en toegepaste korting per factuur op te vragen, voer deze uit en print het resultaat.

c. Maak een query om de top 3 van facturen met de hoogste omzet op te vragen, voer deze uit en print het resultaat.

Opgave 4: `Factuur` specificatie

Totalen zijn interessant, maar we weten nog niet heel veel van individuele aankopen, op termijn zou het wellicht interessant zijn om te zien welke producten een klant vaak samen koopt, of als een product niet aanwezig is welk product als alternatief wordt gekozen. We zouden met andere worden kunnen kijken wat complementaire- en/of substitutieproducten zijn.[15]

In deze opgave ga je artikelen aan de factuur toevoegen en elk artikel zal worden bewaard in een `FactuurRegel` instantie. De code voor deze klasse wordt gegeven:

```
1  public class FactuurRegel implements Serializable {
2
3      private Long id;
4
5      private Factuur factuur;
6
7      private Artikel artikel;
8
9      public FactuurRegel() {}
10
11     public FactuurRegel(Factuur factuur, Artikel artikel) {
12         this.factuur = factuur;
13         this.artikel = artikel;
14     }
15
16     /**
17      * @return een printbare factuurregel
18      */
```

[15] In bijvoorbeeld e-commerce is dit een belangrijk onderscheid en vormt een basis voor *recommender systems*, het op maat kunnen genereren van aanbevelingen voor klanten op basis van historische data.

```
19      public String toString() {
20          // method body omitted
21      }
22  }
```

a. *Annoteer* de klasse `FactuurRegel` zodat deze door een entitymanager beheerd kan worden. Het veld `factuur` verwijst naar de factuur waar een FactuurRegel toe behoort – wat voor een relatie is dit en welke annotatie zal je hiervoor moeten gebruiken?

b. Voeg aan de klasse `Factuur` een veld `regels` toe met type `ArrayList<FactuurRegel>`. Dit veld moet ook geannoteerd worden – wat voor een relatie is dit en welke annotatie kan je hiervoor gebruiken?

c. Pas `verwerkBestelling` in `Factuur` aan zodat per Artikel een FactuurRegel instantie aan `regels` factuur wordt toegevoegd.

d. Pas `toString` in `Factuur` aan zodat ook regels met artikelen op het bonnetje worden geprint. Gebruik hier `toString` in `FactuurRegel` voor.

e. Welke redenen kan je bedenken om Artikelen in een aparte klasse `FactuurRegel` op te slaan en niet direct aan `Factuur` toe te voegen?

Opgave 5: Populaire artikelen

Nu verkochte artikelen ook worden opgeslagen kunnen we additionele queries uitvoeren op op artikelniveau queries op te stellen en uit te voeren, wederom in `KantineSimulatie` aan het einde van de simulatie.

a. Maak een query om de totalen en toegepaste korting per artikel op te vragen, voer deze uit en print het resultaat.

b. Maak een query om de totalen en toegepaste korting per artikel, *per dag* op te vragen. Voer deze uit en print het resultaat.

c. Maak een query om de top 3 van meest populaire artikelen op te vragen, voer deze uit en print het resultaat.

d. Maak een query om de top 3 van artikelen met de hoogste omzet op te vragen, voer deze uit en print het resultaat.

Opgave 6: Bonus

De simulatie is nu ten einde, maar nu je kennis hebt gemaakt met data persistentie in de vorm van ORM zijn er wellicht extra problemen waar je misschien nog aan wilt werken, bijvoorbeeld:

- Artikelen creëer je nu in de simulatie, inclusief voorraadbeheer. Hoe zou je artikelen en voorraad per artikel in een database kunnen vormgeven?
- Bewaar personen in de database en gebruik het BSN nummer als primaire sleutel. Maak een klasse BonusKaart en koppel deze aan een Persoon en bedenk hoe je deze kan gebruiken voor extra kortingen.
- Een Kassa heeft contant geld in de lade. Breid contante betalingen uit met wisselgeld en los situaties op waar een tekort aan wisselgeld kan ontstaan.
- Bedenk zelf een uitbreiding.