Java Opdrachten week 3

Theorie - Four Pillars of OOP

Eerder heb je gelezen dat Java een Object Oriented Programmeertaal is. Maar wat houdt dat nou eigenlijk in?

Simpel gezegd gaat het over de coding structuur en concepten die een programmeertaal gebruikt om de applicatie vorm te geven. Het makkelijkste voorbeeld is het gebruik van Objects om data bij te houden, maar zo zijn er nog wel wat meer.

De belangrijkste concepten die elke OOP-taal gebruikt worden ook wel de **Four Pillars of Object Oriented Programming** genoemd. Het gaat dan over de volgende principes:

- **Inheritance**
- **Polymorphism**
- **Encapsulation**
- **Abstraction**

Hoe Java Inheritance en Polymorphism gebruikt heb je in de opdrachten hiervoor geleerd. In de volgende opdrachten gaan we de ***Encapsulation*** en ***Abstraction*** behandelen.

Opdracht 1 - Encapsulation

Encapsulation is het idee dat elk Object verantwoordelijk is voor zijn eigen data. Dit soort scheiding van verantwoordelijkheid maakt het makkelijker om na te gaan wanneer er iets aan een Object veranderd wordt.

Java gebruikt Encapsulation door middel van **private variables** en **Getters/Setters**. Dit houdt in dat de variabelen van een Object alleen gelezen of veranderd kunnen worden door methods van datzelfde Object. Deze methods worden dus ***getters*** en ***setters*** genoemd.

a) Maak een class School en een class Student. Geef beiden een private variable name, een method getName die deze weergeeft en een method setName die deze aanpast naar de gegeven parameter.

Je hebt nu je eerste eigen Getter en Setter gemaakt. Het is zoals je misschien merkt veel getyp voor een enkele variable. Gelukkig hebben meeste IDE's tegenwoordig shortcuts om dit makkelijker te maken - dan hoef je alleen de private variable uit te typen. Daarna kan je IDE er automatisch de bijbehorende getters en setters voor aanmaken.

b) Onderzoek wat deze shortcuts zijn voor jouw IDE.

Opdracht 2 - Abstraction

Abstraction is het opzettelijk weglaten van concrete details. Net zoals je in een korte ochtendvergadering niet alles van je collega's wil weten, is het voor een computer ook handig om de details soms te laten totdat ze nodig zijn. ***Als je code abstract maakt, beschrijf je dus alleen WAT er gebeurt, en niet HOE het gebeurt.***

Java gebruikt hiervoor ook het *keyword* ***abstract.*** Dit wordt gebruikt gebruikt om aan te geven dat een method of class abstract is.

a) Wanneer is een class abstract?
b) Maak in je Student class een abstract method *passYear().* Kan je al bedenken op wat voor plek je de code voor *passYear()* zou moeten implementeren?

Een abstract class kan je niet instantiëren. Omdat de method *passYear()* in je Student class abstract is gemaakt, is het niet mogelijk om een Student object te creëren.

c) Maak 4 subclasses van Student aan. Noem deze *FirstYear, SecondYear, ThirdYear* en *FourthYear*.
d) Implementeer de abstract method *passYear()* bij elk van deze! *(alleen een bericht in de console printen is voorlopig al genoeg).*
e) Geef elk van deze classes daarna ook het volgende:
    - String Array met lessen die ze volgen (Engels, Nederlands, Wiskunde etc.)
    - Numbers Array met cijfers die ze voor elk van deze vakken hebben
    - String Arraylist met notities van slecht gedrag die over deze student gemaakt zijn als ze bijv. te laat zijn gekomen.
    - Methods om bovenstaande Arrays te wijzigen
    - Constructor

Theorie - Interfaces

Abstraction kan je dus zien als een soort contract tussen een abstract superclass en een subclass. De superclass daarbij geeft aan welke details de subclass moet beschrijven.

Soms wil je dat je class zich aan meerdere verschillende soorten regels kan houden. Met alleen de superclass is dit een probleem - Java laat maar 1 superclass per subclass toe, en daarmee dus eigenlijk 1 set regels per superclass.

Om dit op te lossen maakt Java gebruik van ***Interfaces***. Net zoals je het gebruik van een superclass aangeeft met het keyword *extends,* geef je het gebruik van een interface aan met het keyword *implements*. Dit ziet er zo uit:

```
public class FirstYearAthletic extends FirstYear implements AthleticStudent
```

```
public class FirstYearHonor extends FirstYear implements HonorStudent
```

Je kan ook meerdere interfaces tegelijkertijd gebruiken:

```
public class FirstYearUnicorn extends FirstYear implements AthleticStudent, HonorStudent
```

Een Interface is dus vergelijkbaar met een abstract class. Een belangrijk verschil is dat een Interface compleet abstract is. Dit houdt in dat ***ALLE*** methods van een Interface abstract moeten zijn.

Opdracht 3 - Interfaces

Nu gaan we onze Studenten uit de vorige opdrachten wat meer uitbreiden. De school wil namelijk dat bepaalde studenten extra vakken kunnen volgen als ze zich extra goed gedragen. De school noemt deze studenten *Honor Students*.

a) Maak een Interface *HonorStudent*, en geef deze de volgende abstract methods:
   - *removeFromProgram()*;
   - *checkForNotes();*

b) Maak voor je vier jaarlagen Studenten elk een nieuwe Honor Student subclass aan, dus voor *FirstYear* maak je *FirstYearHonor* etc. Implementeer bij al deze classes de *HonorStudent* interface.

Misschien vraag je je af waarom er geen informatie over bijvoorbeeld de extra vakken in de Interface staat. Deze kunnen nog verschillen per jaarlaag van de studenten - in de Interface staat alleen beschreven wat voor ELKE Honor Student verplicht is.

- Elke Honor Student **moet** een manier hebben om omgezet te worden naar een normale student - dus elke Honor Student **moet** een method *removeFromProgram()* **implementeren**.
- Elke Honor Student **moet** een manier hebben om te checken of ze nog voldoen aan de voorwaarden van de school. Als de school bijvoorbeeld de voorwaarde heeft dat een Honor Student niet meer dan 1 keer per maand een notitie van slecht gedrag. Elke Honor Student **moet** dit dus kunnen checken met de method *checkForNotes();*

Bonus Theorie - Swing

Je hebt in de opgaven inmiddels een aantal Software-systemen beschreven, maar het zal waarschijnlijk nog niet echt voelen alsof je een *'echt'* programma aan het maken bent. Zo'n programma heeft namelijk niet alleen een mooi systeem van informatie beschreven, maar ook een manier om dit te draaien en door gebruikers te laten veranderen.

Sommige programma's lossen dit op door via de command-line om User Input te vragen met de scanner. Om foutjes met de input te voorkomen is het uiteindelijke resultaat vaak iets wat lijkt op een telefoonmenu - je toetst een getal in en het programma verwerkt je keuze met een switch.

Maar voor ingewikkelde programma's, zoals een schoolsysteem, is dat niet toereikend. Wat je daarvoor dan nodig hebt is een ***GUI*** - een ***Graphical User Interface***. Java heeft meerdere frameworks die dit voor je kunnen regelen - het framework wat wij voor de eindopdracht gaan gebruiken heet ***Swing***.

Daarom is het verstandig om je vast in te lezen over Swing en hoe je het gebruikt. Een goede tutorial kan je hier vinden:

https://www.javatpoint.com/java-swing

Swing gebruikt dus verschillende onderdelen die je naar wens kan gebruiken. Maar als je er nog niet bekend mee bent, is het natuurlijk erg lastig kiezen. Daarom is er ook een handige visuele uitleg van de verschillende componenten hier:

https://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html