



samen sterk voor werk

**Java**

# **JAVA PROGRAMMING FUNDAMENTALS**

**Cursus**

Deze cursus is eigendom van VDAB Competentiecentra ©

Peoplesoftcode:

Wettelijk depot:

versie: 02/02/2016



## INHOUD

1	Inleiding .....	9
1.1	De Java-programmeertaal .....	9
1.1.1	Platformonafhankelijkheid .....	10
1.2	Doelstelling van de cursus .....	11
1.3	Vereiste voorkennis .....	11
1.4	Nodige software .....	11
1.4.1	De Java Development Kit .....	11
1.4.2	Een IDE: NetBeans - Eclipse .....	13
2	Aan de slag met NetBeans .....	14
2.1	Opstarten van NetBeans .....	14
2.2	Een eerste NetBeans project .....	15
2.2.1	Compileren en uitvoeren .....	17
2.2.2	Naamgevingconventies .....	18
2.2.3	Puntkomma's, blokken en witruimtes .....	18
3	Code en commentaar .....	19
3.1	Commentaar stijlen .....	19
3.2	Sneltoetsen .....	20
4	Variabelen en operatoren .....	21
4.1	Variabelen en literals .....	21
4.1.1	Data types .....	21
4.1.2	Variabelen, references en literals .....	22
4.1.3	Constanten (of final variabelen) .....	24
4.1.4	Identifiers .....	24
4.2	Operatoren .....	24
4.2.1	Soorten operatoren .....	24
4.2.2	Type casting van primitive types .....	27
4.2.3	Prioriteitsregels .....	27
4.2.4	Stringoperatoren .....	28

4.3	Enkele praktijkvoorbeelden .....	28
4.3.1	Eerste voorbeeld: Bmi.....	28
4.3.2	Tweede voorbeeld: Temperatuur .....	30
4.3.3	Derde voorbeeld: BTWnummer.....	30
4.4	Invoer door de gebruiker via toetsenbord .....	31
4.5	Oefeningen .....	32
5	Arrays.....	33
5.1	Arrays van primitive data types of strings .....	33
5.2	Arrays van objecten .....	36
5.3	Arrays van arrays .....	38
5.4	Oefeningen .....	38
6	Programmaverloop .....	39
6.1	Blokken van statements en scope van variabelen .....	39
6.2	Keuzestructuren.....	39
6.2.1	De if-instructie.....	40
6.2.2	De conditional operator ? .....	41
6.2.3	De switch.....	41
6.3	Lussen .....	43
6.3.1	De while-lus.....	44
6.3.2	De do...while-lus.....	45
6.3.3	De for-lus.....	45
6.3.4	De for-each-lus.....	46
6.3.5	Breaks en labels .....	46
6.4	Oefeningen .....	48
6.5	Procedures en functies .....	48
6.5.1	Procedures .....	48
6.5.2	Functies .....	50
7	OO, classes en objects.....	53
7.1	Classes en objects .....	53

7.2	Een class maken .....	54
7.2.1	Opbouw van een class .....	56
7.2.2	Main-programma dat werkt met objecten van de class .....	56
7.3	Access modifiers .....	58
7.4	Constructors .....	58
7.4.1	Default constructor .....	59
7.5	Membervariabelen versus lokale variabelen .....	59
7.6	Method overloading .....	59
7.7	Getters en setters .....	60
7.8	Oefening .....	61
7.9	Primitive types versus class types .....	61
7.9.1	Null reference .....	62
7.10	Private methods .....	63
7.11	Het sleutelwoord <code>static</code> .....	64
7.12	Oefeningen .....	64
8	Inheritance .....	65
8.1	Introductie .....	65
8.2	Het sleutelwoord <code>extends</code> .....	66
8.3	Beveiligingsniveau <code>protected</code> in een class .....	67
8.4	Constructors en inheritance .....	67
8.5	Toepassen van inheritance bij Spaarrekening .....	67
8.6	Method overriding .....	70
8.6.1	Annotations .....	72
8.7	Het sleutelwoord <code>final</code> .....	72
8.7.1	Final setters .....	73
8.8	De class <code>Object</code> .....	75
8.8.1	<code>toString()</code> .....	75
8.8.2	<code>equals()</code> .....	77

8.9	Abstracte classes en methods .....	77
8.9.1	Abstracte classes.....	77
8.9.2	Abstracte methods.....	78
8.10	Polymorfisme .....	79
8.10.1	De operator <code>instanceof</code> .....	80
8.10.1.1	<code>equals()</code> uitgewerkt in de class <code>Rekening</code> .....	81
8.11	De for-each-lus.....	82
8.12	Oefeningen .....	83
9	Strings .....	84
9.1	Introductie .....	84
9.2	Speciale tekens in een string .....	84
9.3	Bewerkingen met strings .....	85
9.3.1	Het vergelijken van strings.....	85
9.3.2	Strings wijzigen.....	86
9.3.2.1	<code>replace()</code> .....	86
9.3.2.2	<code>toLowerCase()</code> .....	86
9.3.2.3	<code>toUpperCase()</code> .....	87
9.3.2.4	<code>trim()</code> .....	87
9.3.3	Strings onderzoeken .....	87
9.3.4	Een voorbeeld .....	88
9.3.5	Strings opsplitsen .....	89
9.3.5.1	<code>split()</code> .....	89
9.4	Strings: conversie van en naar primitive types.....	90
9.4.1	Wrapperclasses.....	90
9.4.2	Een String omzetten naar een primitive type .....	91
9.4.3	Een primitive type omzetten naar een String .....	91
9.5	De class <code>StringBuilder</code> .....	91
9.6	De class <code>StringBuffer</code> .....	93
9.7	Oefeningen .....	93
10	Interfaces .....	94

10.1	Declaratie en beschrijving .....	95
10.2	Implementatie in een class .....	95
10.3	Interface als een data type .....	99
10.4	Oefeningen .....	100
11	Packages.....	101
11.1	Algemeen.....	101
11.2	Naamgeving conventie voor packages.....	101
11.2.1	Algemeen .....	101
11.3	Voorwaarden voor classes in packages.....	102
11.3.1	Een package maken in NetBeans.....	102
11.3.2	Een nieuwe class toevoegen aan een bestaande package .....	103
11.3.3	Een nieuwe class toevoegen aan een nieuwe package .....	103
11.3.4	Een class naar een andere package verplaatsen .....	103
11.4	Verwijzen naar interfaces en classes uit een package .....	103
11.4.1	Algemeen .....	103
11.4.2	Ondersteuning van NetBeans .....	104
11.5	Jar-bestand .....	106
11.6	Class path .....	107
11.6.1	Algemeen .....	107
11.6.2	De class path uitbreiden in NetBeans.....	108
11.7	Oefeningen .....	108
12	Exception Handling.....	109
12.1	Exceptions afhandelen in een try-catch-blok.....	109
12.1.1	Het finally-blok.....	112
12.2	Meerdere catch-blokken bij één try-blok. ....	112
12.3	Multi catch.....	113
12.4	Eigen exceptions maken.....	113
12.4.1	Eigen exception toegepast bij de class Rekening .....	114
12.5	Oefeningen .....	119

13	BigDecimal .....	120
13.1	Probleemstelling .....	120
13.2	De BigDecimal class .....	121
13.2.1	Creatie van een object .....	121
13.2.2	Methods .....	121
13.3	Het voorbeeld van de probleemstelling opgelost met BigDecimal .....	122
14	Collections .....	123
14.1	Generics .....	123
14.1.1	Generic classes .....	124
14.2	Inleiding tot collections .....	124
14.2.1	Wat is een collection? .....	124
14.2.2	Concepten van collections .....	126
14.3	De collection interface .....	126
14.4	De diamond operator < > .....	129
14.5	De List interface .....	129
14.5.1	ArrayList .....	130
14.5.1.1	ArrayList: een voorbeeld .....	131
14.5.1.2	Itereren met de iterator .....	132
14.5.2	LinkedList .....	133
14.5.2.1	LinkedList: een voorbeeld .....	133
14.5.2.2	Implementatie van LinkedList .....	134
14.5.2.3	Enkele extra methods .....	135
14.5.3	Verschil tussen ArrayList en LinkedList .....	136
14.6	Oefeningen .....	137
14.7	De Set interface .....	137
14.7.1	De hashcode .....	137
14.7.2	De hashtable .....	139
14.7.3	Hoe werken de HashSet en de LinkedHashSet? .....	139
14.7.4	De HashSet .....	142
14.7.5	De LinkedHashSet .....	144
14.7.6	De TreeSet .....	145



14.7.6.1	Natural ordering .....	146
14.7.6.2	Natural ordering – compareTo().....	148
14.7.6.3	Consistentie van equals() en compareTo().....	151
14.7.6.4	Meerdere sorteringen .....	152
14.8	Oefeningen .....	153
14.9	De Map Interface.....	153
14.9.1	Key-value paren .....	154
14.9.2	Interface Map .....	154
14.9.3	De HashMap .....	156
14.9.3.1	Collection views van de HashMap.....	156
14.9.4	De LinkedHashMap .....	159
14.9.5	De TreeMap .....	160
14.10	Schema .....	162
14.11	Oefeningen .....	164
15	Streams - I/O en Serialization .....	165
15.1	Byte streams.....	165
15.1.1	De classes InputStream en OutputStream.....	165
15.1.2	Try with resources .....	166
15.1.3	BufferedInputStream en BufferedOutputStream.....	167
15.2	Character streams .....	168
15.2.1	Unicode.....	168
15.2.2	De classes Reader en Writer.....	169
15.2.3	De classes BufferedReader en BufferedWriter.....	169
15.3	Andere file operaties .....	170
15.4	Object streams .....	170
15.4.1	Serialization .....	170
15.4.2	De classes ObjectInputStream en ObjectOutputStream .....	171
15.4.3	Transient variabelen .....	172
15.4.4	serialVersionUID .....	173
15.5	Oefeningen .....	174
16	Multithreading .....	175

16.1	Processen en threads.....	175
16.1.1	Proces.....	175
16.1.2	Thread .....	175
16.2	Het verdelen van threads over processoren .....	175
16.3	Threads in Java.....	177
16.3.1	Een class die erft van de class Thread.....	177
16.3.2	Een class die de interface Runnable implementeert .....	178
16.4	De method join van een Thread object .....	179
16.5	De static method sleep van de class Thread .....	180
16.6	De method interrupt van een Thread object .....	181
16.7	Daemon threads .....	182
16.8	Synchronized.....	183
16.8.1	Voorbeeldapplicatie met het probleem.....	183
16.8.2	De oplossing van dit probleem met een synchronized method .....	185
16.8.3	Een synchronized blok .....	185
16.9	Thread safe classes in de Java library .....	186
16.10	Oefeningen .....	186

# 1 Inleiding

---

## 1.1 De Java-programmeertaal

Java is een objectgeoriënteerde programmeertaal waarmee applicaties ontwikkeld worden. Het betreft o.a. administratieve bedrijfsapplicaties, webapplicaties, maar ook programma's voor mobiele apparaten zoals tablets en smartphones.

Java is oorspronkelijk ontwikkeld door Sun, maar later overgenomen door Oracle.

Het hoofdgebruik van java situeert zich in server-side toepassingen. Nochtans, sinds de opkomst van het internet, heeft Java op dit gebied de afgelopen jaren zijn kracht bewezen en wordt de taal gezien als een belangrijke omgeving voor webapplicaties.

Java is eveneens een programmeertaal die gebruikt wordt voor allerlei elektronische apparaten zoals televisies, afstandsbedieningen, wasmachines, enz.

Sinds het ontstaan van Java zijn er reeds verschillende versies uitgebracht. Hieronder een overzicht met het jaartal, de versie en de voornaamste wijziging of uitbreiding:

1996	Java 1.0	
1997	Java 1.1	Aantal verbeteringen + AWT
1998	Java 1.2	Eigenlijk eerste echt bruikbare, stabiele versie. Vanaf nu spreekt men van het Java 2 platform.
2000	Java 1.3	Vooral uitbreidingen (basis was reeds OK)
2002	Java 1.4	Idem
2004	Java 1.5	Ook bekend als <b>Java 5!</b> Veel nieuwigheden voor JPF.
2006	Java 6	Weinig nieuwigheden voor JPF, eerder uitbreidingen voor gevorderde onderwerpen.
2011	Java 7	Aantal verbeteringen en vereenvoudigingen voor JPF.
2014	Java 8	Extra functionaliteit, zoals o.a. lambda-expressies.

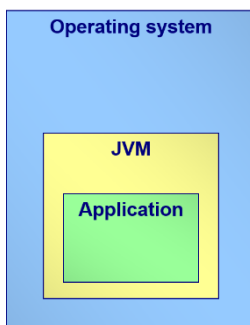
Java bestaat in meerdere edities:

	<b>JSE</b>	Java Standard Edition	Basisversie. Bevat ongeveer 3000 classes.
	<b>JEE</b>	Java Enterprise Edition	JSE + alle bibliotheken voor uitgebreide client-server toepassingen. Bevat ongeveer 6500 classes.
	<b>JME</b>	Java Micro Edition	Sterk afgeslankte versie voor mobiele apparaten.

Voor het volgen van deze cursus heb je voldoende aan de standaard editie, de JSE.

### 1.1.1 Platformonafhankelijkheid

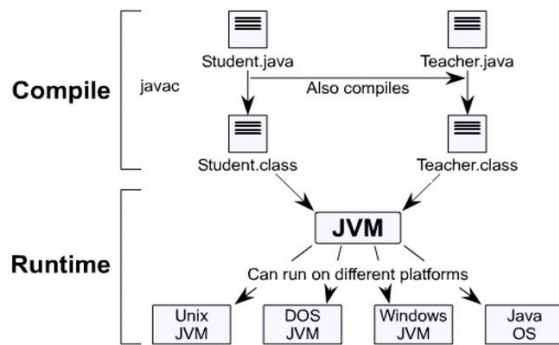
Een belangrijk kenmerk van Java is de platformonafhankelijkheid. Dat wil zeggen dat een applicatie geschreven in Java uitgevoerd kan worden op meerdere besturingssystemen. Op elk besturingssysteem, waarvoor een **JVM** (Java Virtual Machine) bestaat, kan de applicatie uitgevoerd worden.



De JVM samen met software-libraries specifiek voor het operating system, vormt de **JRE** (Java Runtime Environment). De JRE heeft drie belangrijke taken:

- laadt de code,
- verifieert de code en verzekert dat de code voldoet aan de JVM specificatie, en
- voert vervolgens de code uit

Weergegeven in een voorbeeld:



Een source-file heeft de extentie .java. De classes Student.java en Teacher.java worden gecompileerd m.b.v. javac (=javac.exe). Resultaat zijn de respectievelijke Student.class en Teacher.class files. Deze files worden uitgevoerd door de betreffende JVM.

## 1.2 Doelstelling van de cursus

Je leert de basissyntax van Java en de objectgeoriënteerde principes toepassen in Java. In hoofdzaak leer je dit door het schrijven van console-applicaties. Slechts één hoofdstuk wordt besteed aan het schrijven van een desktop-applicatie met een GUI (Graphical User Interface).

Er wordt gewerkt met Java 7.

## 1.3 Vereiste voorkennis

- Programmatielogica
- Objectgeoriënteerde principes (OOP)

## 1.4 Nodige software

Om java-applicaties te ontwikkelen is het volgende nodig:

- De Java Development Kit (JDK)
- Een IDE voor het schrijven van de code m.n. NetBeans

### 1.4.1 De Java Development Kit

Voor het ontwikkelen van een java-applicatie is de Java Development Kit, of kortweg de JDK, nodig. De JDK bevat de compiler, de JRE en alle bibliotheken nodig voor het ontwikkelen van software. Daarnaast bevat de JDK ook een aantal hulpprogramma's, bijv. voor het genereren van documentatie (javadoc), het debuggen van applicaties, ...

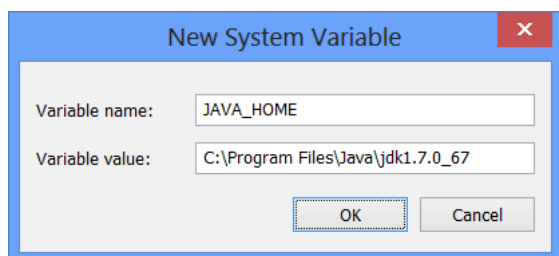
De JDK kan vrij gedownload worden van de site van Oracle ([www.oracle.com](http://www.oracle.com)). Je kiest vervolgens voor tabblad Downloads, Java, Java SE en je zorgt ervoor dat je de laatste versie van Java SE 7 downloadt. Bijv. *jdk-7u67-windows-x64.exe* betreft een JDK van java-versie 7, update 67 voor een 64-bit windows-omgeving. Vervolgens installeer je deze JDK op je PC. De default locatie voor de installatie is *C:\Program Files\Java*. Na de installatie is er dan de folder *C:\Program Files\Java\jdk1.7.0\_67*. Je kan meerdere java-versies naast elkaar installeren.

Bij de JDK hoort een API-documentatie. API staat voor Application Programming Interface. Dit bevat

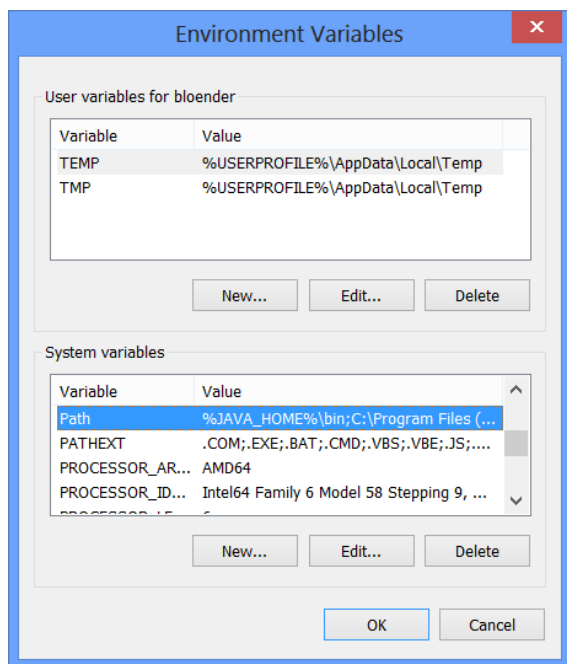
de documentatie van de classes van de Java SE. Dit is beschikbaar in de vorm van een .zip-file. Ook deze dient gedownload te worden. Klik hiervoor terug op tabblad Downloads en scrol naar beneden tot bij de rubriek Additional Resources. Vervolgens download je de documentatie van Java 7. Bijv. *jdk-7u67-apidocs.zip* betreft de documentatie die hoort bij dezelfde JDK-versie. Deze zip-file dient uitgepakt te worden. De default locatie hiervoor is *C:\Program Files\Java\jdk1.7.0\_67*. Er wordt automatisch een subfolder *docs* gecreëerd waarin de file uitgepakt wordt. Op deze manier heb je steeds de overeenkomende versie van de documentatie geplaatst bij de betreffende JDK.

**Gebruik van deze documentatie wordt toegelicht in de bijlage. (ToDo)**

Na de installatie van de JDK is het gebruikelijk om de omgevingsvariabele `JAVA_HOME` in te stellen. Deze omgevingsvariabele bevat het pad waar de JDK geïnstalleerd is. Dit doe je als volgt: Control Panel – System – Advanced system settings – knop Environment Variables... en je kiest bij System variables voor New...:



Tot slot voeg je het pad van de installatie van de JDK toe aan het path, m.a.w. aan de omgevingsvariabele Path van het systeem. Alle uitvoerbare programma's van de JDK staan in de bin-directory (*C:\Program Files\Java\jdk1.7.0\_67\bin*). Voeg daarom **vooraan** het Path het volgende toe: `%JAVA_HOME%\bin`:



`%JAVA_HOME%` = de waarde van `JAVA_HOME`, dus *C:\Program Files\Java\jdk1.7.0\_67*

### 1.4.2 Een IDE: NetBeans - Eclipse

Je kan Java programma's schrijven zonder gebruik te maken van een specifieke editor. Bijv. met notepad of een andere eenvoudige tekst-editor kan je een java-programma schrijven. Vervolgens kan je dit programma compileren en uitvoeren met de hulpprogramma's van de JDK.

Het gebruik van een IDE (Integrated Development Environment) heeft uiteraard veel voordelen. Dat is een ontwikkelomgeving die de programmeur ondersteunt bij het schrijven van code. Er bestaan diverse IDE's. De meest bekende zijn NetBeans en Eclipse. NetBeans is een product van Oracle. Eclipse is een product van de Eclipse Foundation. Voor de cursus JPF en JDBC wordt er gewerkt met NetBeans. Vanaf de cursus Servlets & JSP is er de keuze tussen het gebruik van NetBeans of Eclipse.

Je kan NetBeans downloaden van <https://netbeans.org/>. Kies voor de download bundle 'Java EE'. Dit volstaat voor het doornemen van de cursussen van het java-opleidingstraject. Op dit moment is de laatste versie NetBeans 8. De file voor het windows-platform is *netbeans-8.0.2-javaea-windows.exe*.

Wanneer je de installatie start, kunnen twee opties aangevinkt worden om geïnstalleerd te worden:

- GlassFish Server (= een applicatieserver): niet noodzakelijk
- Apache Tomcat Server (= een webserver die gebruikt wordt bij de cursus Servlets en JSP): optioneel

Tijdens de installatie wordt gevraagd naar de locatie voor NetBeans (is default ingevuld *C:\Program Files\NetBeans 8.0.2*) en naar de JDK die door NetBeans gebruikt zal worden (default wordt de laatste versie voorgesteld *C:\Program Files\Java\jdk1.7.0\_67*).

De voorbeelden en oefeningen van deze cursus zijn uitgewerkt in NetBeans.

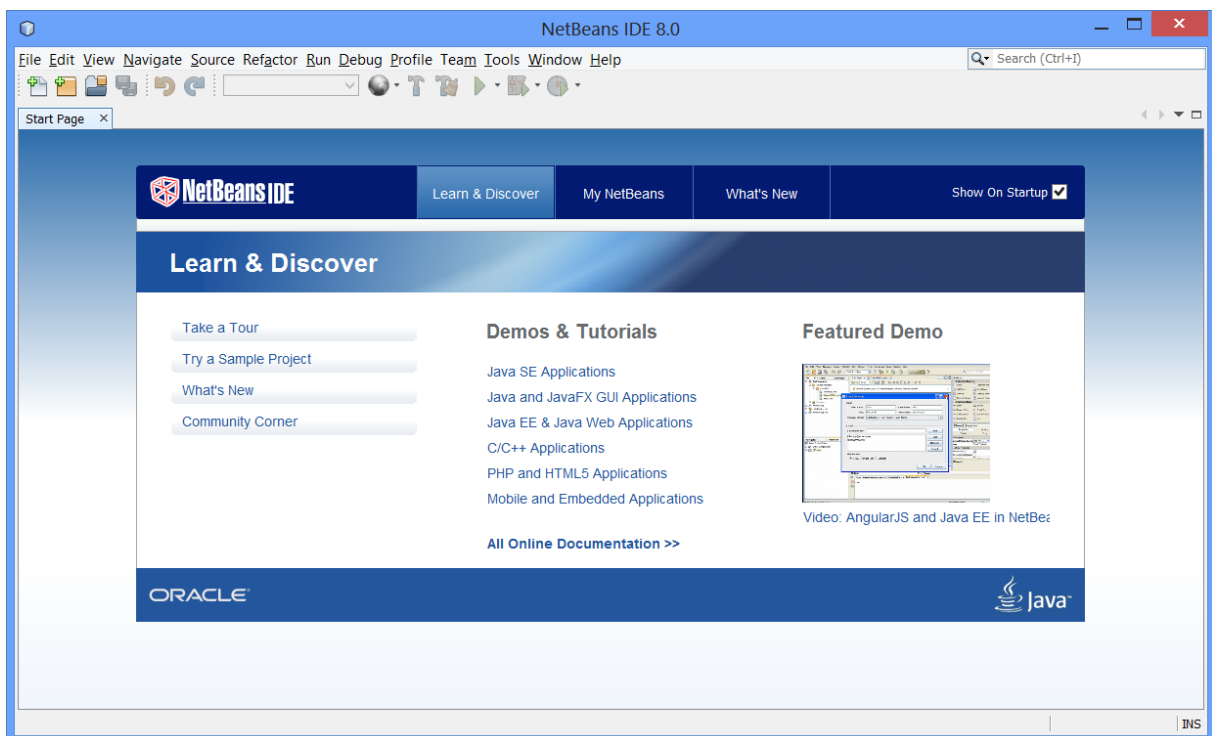
## 2 Aan de slag met NetBeans

### 2.1 Opstarten van NetBeans

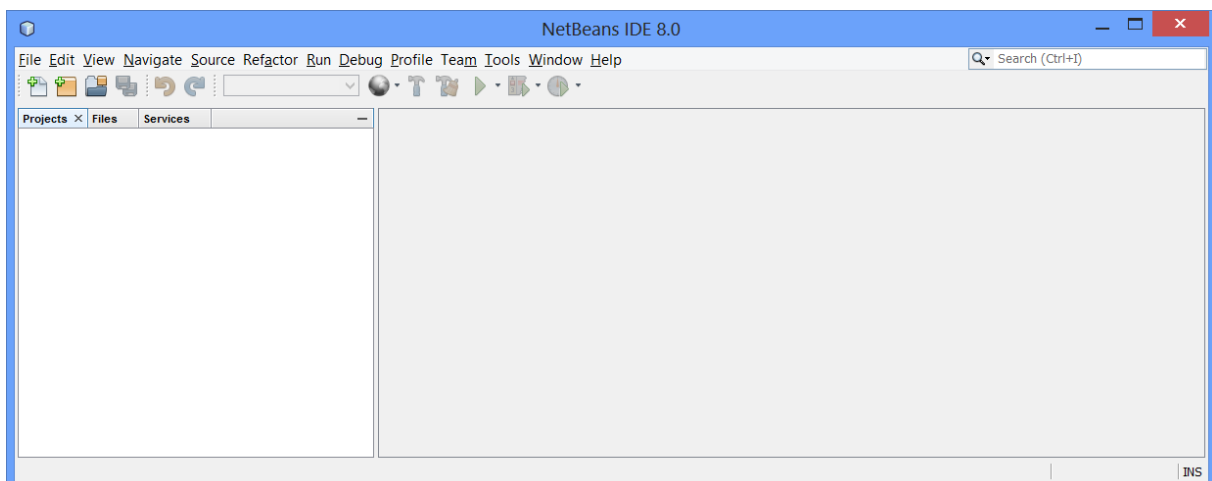
Onderstaande startpagina wordt getoond wanneer NetBeans gestart wordt. De recente projecten die je gemaakt hebt, worden getoond onder tabblad My NetBeans. Deze projecten kunnen vanuit de startpagina geopend worden.

Verder kan je hier een aantal tutorials doornemen en de laatste nieuwtjes volgen.

Rechtsboven in de hoek van dit venster kan je deze optie uitvinken, zodat de startpagina niet langer verschijnt bij het opstarten van NetBeans.



Na het sluiten van de startpagina wordt er links een venster getoond met 3 tabbladen: Projects, Files en Services.

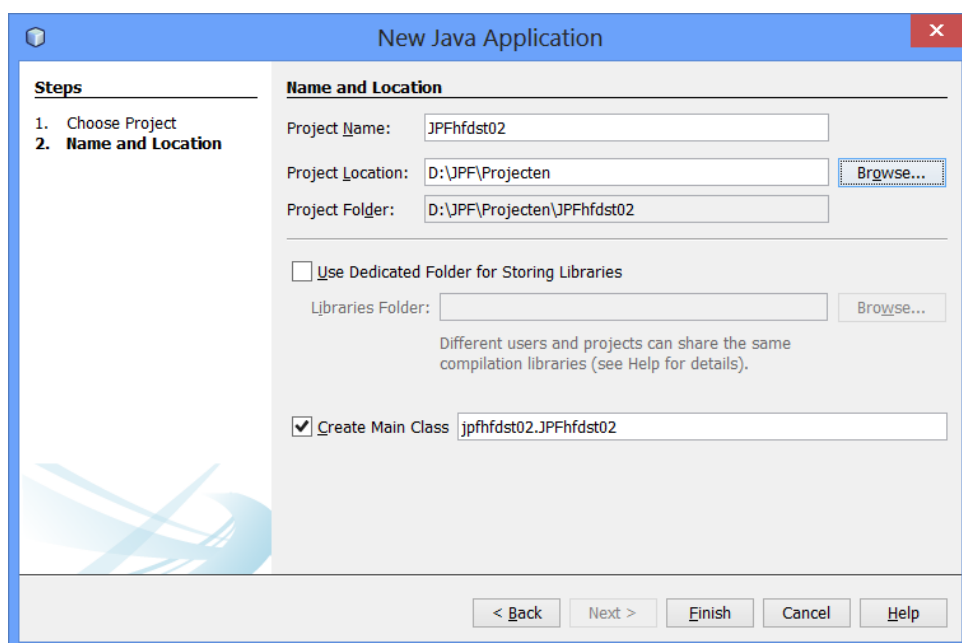




## 2.2 Een eerste NetBeans project

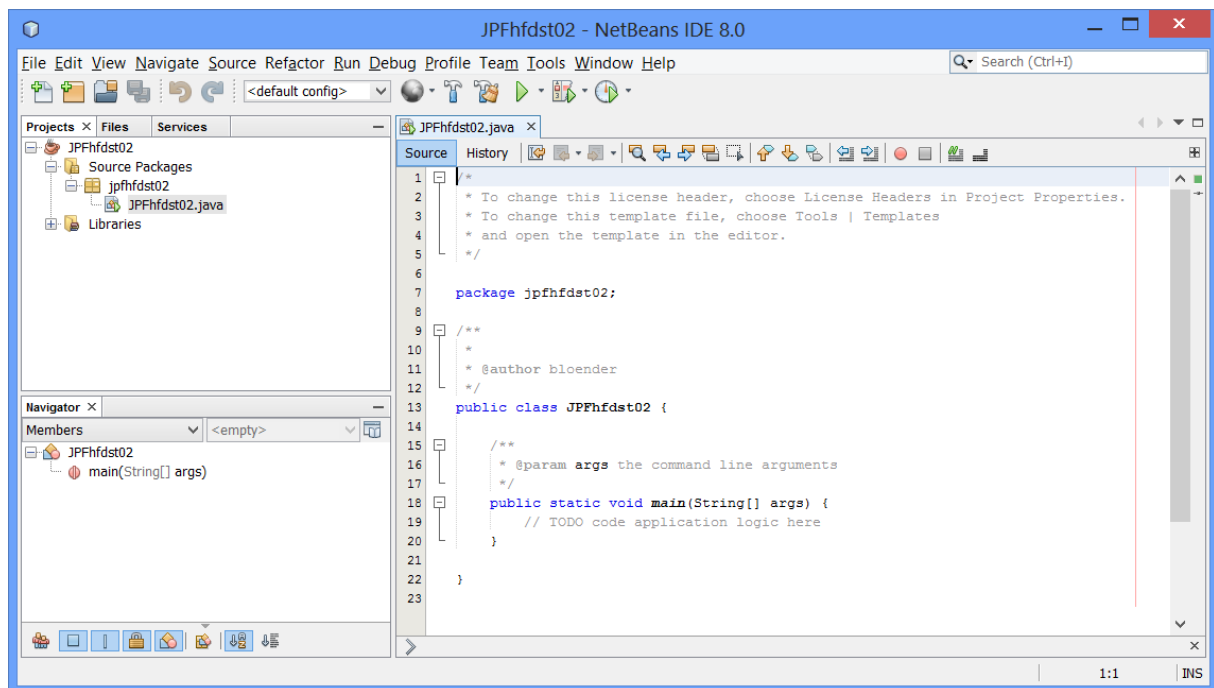
Applicaties worden met NetBeans opgebouwd in de vorm van projecten. Initieel start je met één enkel programma dat uitgevoerd kan worden. Dat programma maakt deel uit van dit project. Later worden de projecten steeds groter en zullen ze bestaan uit meerdere files, meerdere classes, eventueel met afbeeldingen, enz.

Je maakt een nieuw project via menu File, optie New Project..., kies uit de Categorie Java voor een project Java Application (zo maak je een eenvoudige console-applicatie). Klik Next. Geef het project een naam (bijv. JPFhfdst02) en selecteer de juiste plaats voor dit project. Het project wordt default geplaatst in een folder die dezelfde naam heeft als het project. Laat de optie voor het creëren van een Main Class aangevinkt. Dit levert je automatisch een main-programma op met de naam *JPFhfdst02.java*. Een Java source-file heeft altijd de extentie *.java*.



Klik op Finish en het project wordt aangemaakt.

Automatisch is er een .java file gegenereerd met de naam *JPFhfdst02.java*:



Op de harde schijf is volgende directorystructuur aangemaakt: Folder *JPfhfdst02* met subfolders *nbproject* en *src*. De subfolder *nbproject* is een folder van NetBeans waarin projectgegevens door NetBeans worden bijgehouden. De subfolder *src* zal alle source-files van het project bevatten, default in een subfolder met de naam van het project (*jpfhfdst02*). Deze subfolder is een package waarin de code is ondergebracht. Later meer over packages. Momenteel zit hierin enkel *JPfhfdst02.java*.

	Name	Date modified	Type	Size
JPF				
Projecten				
JPfhfdst02				
nbproject				
src				
jpfhfdst02				

	Name	Date modified	Type	Size
	JPfhfdst02.java	22/08/2014 14:04	JAVA File	1 KB

Pas de code van *JPfhfdst02.java* als volgt aan:

```
package jpfhfdst02;

public class JPfhfdst02 {
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("Hello World !");
    }
}
```

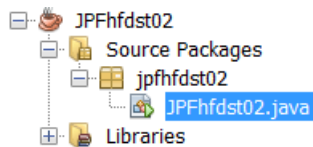
Commentaren zijn verwijderd en er is een klassieke “Hello World” applicatie van gemaakt. De opdracht `System.out.println("Hello World !");` zorgt ervoor dat de tekst “Hello World !” getoond wordt. De overige lijnen code worden later nog in detail uitgelegd.

Indien je geen regelnummering ziet, kan deze opgezet worden door rechts te klikken in de grijze balk, en Show Line Numbers aan te vinken.

### 2.2.1 Compileren en uitvoeren

Wanneer de applicatie uitgevoerd wordt, wordt de file automatisch eerst bewaard, vervolgens gecompileerd en tot slot uitgevoerd.

- Je kan deze applicatie uitvoeren door in het Projects-venster rechts te klikken op JPFhfdst02.java en vervolgens te kiezen voor Run File. Wanneer de code nog niet bewaard is, wordt deze eerst automatisch opgeslaan.



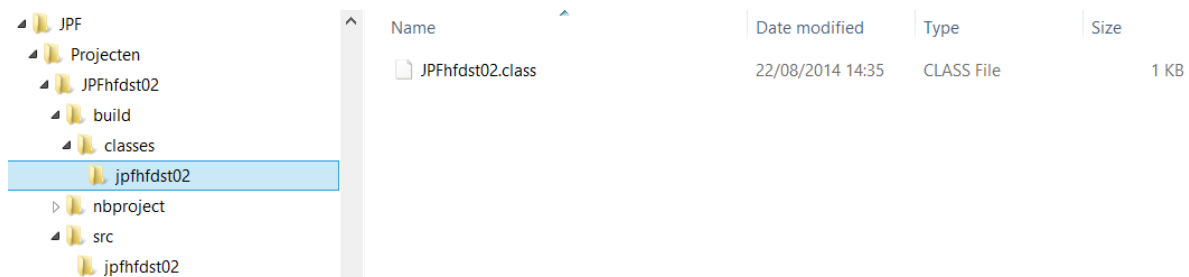
Het groene driehoekje bij de naam van de .java file duidt op het feit dat dit een uitvoerbaar programma is, concreter een uitvoerbare klasse. Deze klasse bevat de `public static void main(String[] args)` method. Later meer over de betekenis hiervan. Files die deze method niet hebben, kunnen niet uitgevoerd worden.

- Bovenaan in de menubalk, staat ook het symbool  waarmee je rechtstreeks de applicatie kan uitvoeren.
- Een derde manier voor het uitvoeren van een file of project is via het menu Run. De beschikbare opties van dit menu worden bepaald door hetgeen geselecteerd is links in het Projects venster: Run Project en/of Run File.

Onderaan in het **Output venster** wordt het resultaat weergegeven:



Het compileren van een .java file levert een .class file op. NetBeans plaatst deze gecompileerde klasse in een subfolder *build\classes*. Elke gecompileerde klasse staat in een subfolder (package) waarvan de naam overeenkomt met de folder src (hier is dat jpfhfdst02).



### 2.2.2 Naamgevingconventies

Java is hoofdlettergevoelig! De naam van elke klasse begint met een hoofdletter: JPFhfdst02.java. Dit had ook JpfHfdst02.java kunnen zijn.

Namen van packages (dit zijn (sub)folders) staan altijd volledig in kleine letters (jpfhfdst02).

De naam van een method begint steeds met een kleine letter maar wanneer de naam bestaat uit meerdere woorden, wordt vanaf het tweede woord elke eerste letter van elk woord geschreven met een hoofdletter, bijv. main() of verwerkGegevens() of invoerNieuwGetal(). Deze vorm wordt *CamelCase* genoemd.

Gaandeweg zal je de conventies rond naamgeving leren kennen.

### 2.2.3 Puntkomma's, blokken en witruimtes

In Java worden statements (ook opdrachten genoemd) afgesloten met een puntkomma ; .

Voorbeeld: `System.out.println("Hello World !");`

Een blok is een groep van statements omgeven door accolades. Voorbeeld:

```
{  x = x + 1;
  y = y + 1;
}
```

Het blok zelf hoeft niet afgesloten te worden met een puntkomma, maar de statements binnen het blok wel.

De class definitie staat in een blok. De code van een method (bijv. main()) staat eveneens in een blok.

```
public class JPFhfdst02 {
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("Hello World !");
    }
}
```

Zoals je kan zien kunnen blokken genest worden. Het aantal witruimtes heeft geen belang.

## 3 Code en commentaar

---

Het is belangrijk de code te voorzien van commentaar. Sommige bedrijven hanteren hierrond bepaalde bedrijfsafspraken, bijv. commentaar in het Engels. Commentaar ondersteunt de programmeur die later de code eventueel dient aan te passen. Bovendien bestaat er een vorm van commentaar zodat er van je project automatisch documentatie gegenereerd kan worden zoals de API-documentatie.

Commentaar om de commentaar is niet nodig. Een gulden middenweg is de boodschap.

### 3.1 Commentaar stijlen

Er bestaan drie commentaar stijlen:

- `//` commentaar tot aan het einde van de lijn
- `/*` commentaar over meerdere regels `*/`

Tijdens de ontwikkelfase van een applicatie kan het handig zijn om volledige delen code in commentaar te zetten. Dit kan je doen met de stijl `/* */`.

Via het menu van NetBeans kan dat ook: selecteer de code die je in commentaar wil zetten, kies menu *Source > Toggle Comment*. Elke geselecteerde lijn code wordt voorafgegaan door `//`. Door nogmaals hetzelfde te doen, wordt de `//` vooraan elke lijn weer weggehaald.

- `/**`
  - \* documentatie commentaar: deze stijl wordt gelezen door de javadoc utility
  - \* (is geïntegreerd in NetBeans en Eclipse) en genereert HTML files met een index-pagina
  - \* zoals de API-documentatie.
  - \*
  - \* Speciale tags kunnen gebruikt worden in deze commentaar, een overzicht van veelgebuikte
  - \* javadoc-tags:
  - \* `@author`
  - \* `@version`
  - \* `@param`
  - \* `@return`
  - \* `@see`
  - \* `@throws`
  - \* `/`

Bij het genereren van de javadoc (commentaar) wordt de informatie van deze speciale tags verwerkt. (zie [bijlage – moet nog geschreven worden - TODO](#))

## 3.2 Sneltoetsen

Wanneer een aantal regels geselecteerd zijn, kan je in NetBeans met de menuoptie *Source > Toggle Comment* deze regels in commentaar plaatsen. Op dezelfde manier haal je regels code die in commentaar staan weer uit commentaar.

Handig is zeker het gebruik van de sneltoetsen om een gedeelte van de code in commentaar te plaatsen of uit commentaar te halen. Voor NetBeans is dat volgende toetsencombinatie: **Ctrl + Shift + C**.

Er zijn meerdere sneltoetsen waarvan er vast en zeker een aantal handig zijn wanneer je volop ontwikkelt in NetBeans. Zie hiervoor de menuoptie *Help > Keyboard Shortcuts Card*.

## 4 Variabelen en operatoren

---

In dit hoofdstuk leer je kennis maken met de verschillende gegevenstypes van Java. Ook de operatoren komen hierbij aan bod.

### 4.1 Variabelen en literals

Er zijn variabelen nodig om bijvoorbeeld berekeningen te doen. Deze variabelen kunnen geïnitieerd worden door een vaste waarde of door invoer van de gebruiker. Een vaste waarde is een letterlijke waarde

#### 4.1.1 Data types

Java kent 8 zogenaamde *primitive data types*<sup>1</sup>. Dit zijn de gegevenstypes *byte*, *short*, *int* en *long* om gehele getallen voor te stellen, *float* en *double* voor getallen met cijfers na de komma, *boolean* voor logische waarden en tenslotte *char* voor tekens. Onderstaande tabel maakt duidelijk welke waarden de variabelen van elk data type kunnen bevatten:

Type	Grootte	Bereik
Gehele getallen		
byte	1 byte	Geheel getal tussen -128 en 127
short	2 bytes	Geheel getal tussen -32.768 en 32.767
int	4 bytes	Geheel getal tussen -2.147.483.648 en 2.147.483.647
long	8 bytes	Geheel getal tussen -9.223.372.036.854.775.808 en 9.223.372.036.854.775.807
Reële getallen		
float	4 bytes	Getal met cijfers na de komma tussen 3,4E-38 en 3,4E+38
double	8 bytes	Getal met cijfers na de komma tussen 1,7E-308 en 1,7E+308
Andere types		
boolean	JVM-specifiek	De waarde true of false
char	2 bytes	Individueel teken zoals een letter, cijfer, leesteken of een ander symbool

De kommagetallen bevatten een decimaal punt. Zij kunnen ook eindigen op E of e, gevolgd door een geheel getal, zijnde een macht van tien.

Naast deze primitive data types kent Java zogenaamde *class types*. Het type van de variabele is dan een class. Dit kan een class zijn uit een bestaande library zoals bijvoorbeeld de class *String*, die een tekenreeks kan bevatten. Het kan ook een class zijn die je zelf gemaakt hebt (bijvoorbeeld een class *Persoon*). Later meer over class types.

---

<sup>1</sup> In deze cursus worden zoveel mogelijk Engelse termen gebruikt in zoverre het de leesbaarheid van de cursus niet verstoort. Er wordt gesproken over primitive data types en niet over primitieve gegevenstypes, over references en niet over referenties maar wel over variabelen en niet over variables.

### 4.1.2 Variabelen, references en literals

Het declareren van een variabele gebeurt door het type te vermelden gevolgd door de naam van de variabele. Een declaratie is een statement en dient zoals alle statements afgesloten te worden met een puntkomma.

Enkele voorbeelden van declaraties:

```
int bedrag;  
byte aantalKinderen;  
double salaris;  
boolean gehuwd;  
char letter;  
String voornaam;
```

Volgens de conventie beginnen namen van variabelen met een kleine letter. Wanneer de naam bestaat uit meerdere woorden, wordt vanaf het tweede woord elke eerste letter geschreven met een hoofdletter (aantalKinderen).



Tip: Java is een hoofdlettergevoelige programmeertaal! Zorg dus voor een correct gebruik van hoofdletters en kleine letters.

```
byte aantalKinderen; ≠ byte aantalkinderen;
```

Het type van de variabele schrijf je met een kleine letter wanneer het een primitieve data type is (bijv. `int bedrag;`). Je schrijft het met een hoofdletter wanneer het een class type is (bijv. `String voornaam;`). `String` is een bestaande java klasse. In een variabele van het type `String` kan je een reeks karakters bewaren, bijv. een naam, een woonplaats... Verder in de cursus leer je meer over de klasse `String`.

Voorlopig hebben deze variabelen nog geen waarde. Vaak wordt bij de declaratie van de variabele meteen een waarde gegeven aan deze variabele. Dit kan door gebruik te maken van een `=`-teken en een zogenaamde **literal** (letterlijke waarde).

We onderscheiden 5 soorten literals:

- *integer literals*: stellen gehele waarden voor
- *floating point literals*: stellen getallen voor met cijfers na de komma
- *boolean literals*: de literals `true` en `false`
- *character literals*: individuele tekens
- *string literals*: tekstwaarden

Enkele voorbeelden van declaraties en tegelijk een initialisatie met een literal:

```
int bedrag = 100;  
byte aantalKinderen = 2;  
double salaris = 1525.50;  
boolean gehuwd = true;  
char letter = 'a';  
String voornaam = "Jan";
```



Er is een **fundamenteel verschil** tussen het **declareren** van een **variabele** van een **primitive data type** en anderzijds een variabele van het type String of **een class type**. Een variabele van het type String of een class type is een **reference variabele**. Deze variabele is een reference die wijst naar een object van het type String of een andere class. Zie verder in de cursus de paragraaf over “Primitieve types versus class types”.

#### Voor integer en floating point literals gelden volgende regels:

- Grote literals mogen voor de leesbaarheid een underscore(\_) bevatten. Bijv. op de plaats van het punt voor het duizendtal. Bijv. :  

```
int jaarloon = 47_812;  
int eenMiljoen = 1_000_000;
```
- Elke integer literal is per definitie een int. Dus ook bijvoorbeeld het getal 45 dat eveneens in een byte past.  
Wanneer je een geheel getal binnen de int-grenswaarden toch als een long wil laten beschouwen, moet je er de letter **L** (of kleine letter **l**) achter plaatsen. Bijvoorbeeld: 45L is een long.  
Ook integer literals groter dan de maximum int-grenswaarde moeten gevolgd worden door een **L** (of **l**) zodat ze als een long beschouwd worden. Anders krijg je een foutmelding.



Tip: Gebruik bij voorkeur een hoofdletter L omdat in sommige lettertypes er bijna geen onderscheid is tussen de kleine letter l en het cijfer 1. Dit voorkomt vergissingen.

- Elke floating point literal is per definitie een double. Dus ook bijvoorbeeld het getal 750.25 dat binnen de grenswaarden van een float valt.  
Wanneer je een kommagetal binnen de float-grenswaarden toch als een float wil laten beschouwen, moet je er de letter **F** (of kleine letter **f**) achter plaatsen. Bijv. 750.25F is een float.  
Achter floating point literals mag je altijd een **D** of **d** plaatsen, als je dit duidelijker vindt.



Tip: Naar analogie met de hoofdletter L, gebruik je bij voorkeur de hoofdletters F en D.

- Je kan floats en doubles ook in exponentiële notatie noteren door een e of E tussen mantisse en exponent te plaatsen. Bijv. 1.8e-13 of 0.9E77

#### Voor een boolean literal geldt:

- Een boolean literal kan je initialiseren op `true` of `false`.

#### Enkele regels voor character literals:

- Character literals geven één enkel teken weer dat tussen enkele quotes genoteerd staat.  
Voorbeeld: 'Z', '!', '@'.
- Enkele speciale tekens zoals tab, new line, carriage return en tekens zoals backslash, enkele en dubbele quotes worden voorgesteld aan de hand van een escape code: `\t`, `\n`, `\r`, `\\`, `\`, `\"`, `\"`.

### En tenslotte de string literals:

- String literals zijn een verzameling tekens tussen dubbele quotes, voorbeeld: "Java is fijn".
- Wens je speciale tekens of een backslash, quote of dubbele quote op te nemen in de string dan gebruik je de escape sequence zoals hierboven vermeld. Bijv. "Hij zei: \"Hier volgt een \nnieuwe lijn.\""

#### 4.1.3 Constanten (of final variabelen)

Sommige variabelen zijn helemaal niet variabel maar horen gedurende het ganse programma dezelfde waarde te behouden. Deze waarde kan tijdens de uitvoering van het programma niet gewijzigd worden. In de meeste programmeertalen worden dit constanten genoemd. In Java heeft men het over final variabelen.

Om een constante aan te maken creëer je een variabele maar je plaatst het keyword *final* net voor het datatype. Een constante wordt onmiddellijk geïnitieerd met een waarde.

Voorbeeld :

```
final int AANTAL_PROVINCIES = 10;  
final float PI = 3.141592F;
```



Bij conventie zet men de namen van constanten in hoofdletters en plaatst men een underscore tussen woorden.

#### 4.1.4 Identifiers

Eerder is reeds iets gezegd over naamgevingconventies. Voor het gebruik van namen van variabelen, zogenaamde identifiers, bestaan enkele regels :

- De naam van een variabele moet beginnen met een letter, een underscore-teken ( `_` ) of een dollarteken ( `$` ).  
Echter de conventie is dat namen van variabelen (en van methods) beginnen met een kleine letter en verder in CamelCase worden geschreven. Underscores ( `_` ) worden niet meer gebruikt voor het scheiden van woorden.
- Een identifier is case sensitive.

### 4.2 Operatoren

In deze paragraaf wordt er dieper ingegaan op het gebruik van allerhande *operatoren* in *expressies*. Een expressie is een opdracht of een berekening die een resultaat oplevert.

#### 4.2.1 Soorten operatoren

Er zijn vijf soorten operatoren:

- rekenkundige *operatoren*,
- toekenningsoperatoren,
- unaire *operatoren*,
- vergelijkingsoperatoren en
- logische *operatoren*.



Unaire operatoren zijn operatoren die slechts één operand gebruiken, dit in tegenstelling tot de binaire operator waar altijd twee operands nodig zijn (een linker en een rechter).

## Rekenkundige operatoren

De voornaamste binaire rekenkundige bewerkingen zijn :

+	optellen
-	afrekken
*	vermenigvuldigen
/	delen
%	modulo of restbepaling bij gehele deling

Het teken voor de aftrekking kan ook als unaire operator gebruikt worden om een getal negatief te maken.

De bewerking delen: Bij twee gehele getallen levert een deling een geheel getal op en wordt een eventuele rest genegeerd. Wanneer één van de operanden een kommagetal is, is het resultaat ook een kommagetal. Bijv.:

5 / 3 geeft als resultaat 1

5.0 / 3 geeft als resultaat 1.6666666666666667

Wanneer in een expressie meer dan één operator gebruikt wordt, worden automatisch de rekenregels toegepast om de juiste volgorde van de bewerkingen uit te voeren. Binaire operatoren met dezelfde prioriteit worden van links naar rechts uitgevoerd. Je kan haken gebruiken om af te wijken van de rekenregels.

## Toekenningsoperator (of assignment operator)

Over de toekenningsoperator (=) hoeft weinig extra vermeld te worden. Misschien nog dit: een assignment is eigenlijk een expressie en heeft een waarde. Dit betekent dat een instructie als  $a=b=5$  zin heeft. Eerst wordt  $b=5$  uitgewerkt en deze bewerking heeft als waarde 5. Vervolgens wordt de waarde 5 aan  $a$  toegekend en uiteindelijk hebben zowel  $a$  als  $b$  de waarde 5.

Unaire operatoren met dezelfde prioriteit worden van rechts naar links uitgevoerd.

Verkorte schrijfwijze: Java kent voor sommige rekenkundige bewerkingen ook een verkorte schrijfwijze. Een bewerking als  $a = a + 1$  kan ook geschreven worden als  $a+=1$ . In beide gevallen wordt  $a$  met 1 verhoogd. Op een analoge wijze kan je  $-=$ ,  $*=$ ,  $/=$  en  $\% =$  gebruiken.

Nog enkele voorbeelden :

$b /= 2$  is hetzelfde als  $b = b / 2$ ,  $c \% = 5$  is hetzelfde als  $c = c \% 5$

## Unaire operatoren ++ en --

Andere verkorte bewerkingen zijn de increment en decrement operatoren ++ en --. Deze unaire operatoren kunnen zowel vóór een variabele geplaatst worden (++a) als erna (a++). De increment

verhoogt de waarde met 1, de decrement vermindert de waarde met 1. In de prefixnotatie wordt de bewerking uitgevoerd voor de evaluatie van de expressie waarin de bewerking voorkomt. In de postfixnotatie wordt de variabele pas gewijzigd nadat het statement is uitgevoerd.

Enkele voorbeelden:

<code>int a = 2;</code>	de waarde 2 wordt toegekend aan int-variabele a
<code>int b = a++ * 3;</code>	a wordt vermenigvuldigd met 3 → $2 * 3 = 6$ → wordt toegekend aan b dan wordt a met 1 verhoogd → a wordt 3
<code>int c = ++a * 3;</code>	a wordt eerst verhoogd met 1 → a wordt 4 dan wordt a vermenigvuldigd met 3 → $4 * 3 = 12$ → wordt toegekend aan c

## Vergelijkingsoperatoren

Of ook wel relationele operatoren genoemd. Er zijn in Java 6 vergelijkingsoperatoren:

<code>==</code>	gelijk aan,
<code>!=</code>	niet gelijk aan,
<code>&lt;</code>	kleiner dan,
<code>&lt;=</code>	kleiner dan of gelijk aan,
<code>&gt;</code>	groter dan,
<code>&gt;=</code>	groter dan of gelijk aan

Het resultaat van een vergelijking is een boolean.

## Logische operatoren

Je kan meerdere vergelijkingen combineren met:

<code>&amp;</code> en <code>&amp;&amp;</code>	logische operator <i>en</i> ( <i>and</i> ),
<code> </code> en <code>  </code>	logische operator <i>of</i> ( <i>or</i> ),
<code>^</code>	exclusieve of ( <i>exclusive or</i> ) en
<code>!</code>	de logische <i>niet</i> ( <i>not</i> ) !.

Bij de eerste twee logische operatoren zijn er twee schrijfwijzen. Eén enkele `&` of `|` ofwel een dubbele (`&&` of `||`). Bij de dubbele notatie wordt de evaluatie van de expressie gestopt zodra de uitkomst met zekerheid gekend is. D.w.z. als de linker operand van `&&` `false` is, is de waarde van de rechter operand niet relevant en wordt hiervan geen evaluatie uitgevoerd. Hetzelfde geldt voor het geval waarbij de linker operand van `||` `true` is, in dat geval wordt de rechter operand niet geëvalueerd. Daarom worden deze operatoren ook wel short-circuit operatoren genoemd.

De exclusieve of (`^`) levert `true` op als beide operanden een verschillende logische waarde hebben.

De logische not (`!`) verandert het resultaat van een booleaanse expressie van `true` in `false` en omgekeerd.

Voorbeelden:

```
int a = 7;
```

```
int b = 3;
boolean x = a < 10 || ++b == 4;
// Het eerste deel van de expressie is true en dus ook de volledige
// expressie. Wat na de || komt wordt niet meer geëvalueerd en dus
// wordt b niet geïncrementeed! b blijft=3
boolean y = a < 10 | ++b == 4;
// Het eerste deel van de expressie is true, het tweede deel eveneens
// want b wordt eerst geïncrementeed tot waarde 4.
```

#### 4.2.2 Type casting van primitive types

Aan de hand van allerlei operatoren kunnen er berekeningen uitgevoerd worden op diverse gegevenstypes. Soms is het noodzakelijk om een waarde van het ene primitief type te converteren naar een waarde van het andere primitief type (meestal naar een ander type met een kleiner bereik). Dit wordt *casten* genoemd. Casten in Java is toegestaan tussen numerieke types, maar het is niet mogelijk om met een cast een boolean-waarde te converteren naar een ander type of omgekeerd.

Er kunnen zich twee situaties voordoen. Ofwel wil je een variabele omzetten naar een groter type waar meer opslagmogelijkheden zijn. Bijvoorbeeld van een *byte* naar een *int* of van een *float* naar een *double*. Ofwel converteer je naar een kleiner, beperkter type met mogelijk een verlies aan gegevens.

In de eerste situatie is er geen specifieke actie van de programmeur vereist. Java neemt de waarde van de variabele en stopt deze waarde in de grotere variabele. Men noemt dit een *impliciete cast*. Er gaan geen gegevens verloren.

In het tweede geval, dus als je wil converteren naar een beperkter type, spreek je over een *expliciete cast*. Er is wel verlies mogelijk van gegevens (van precisie) indien je converteert van een reël getal naar een geheel getal. Dus bijvoorbeeld van *float* naar *int* of *long*, of van *double* naar *long*.

Een expliciete cast geef je aan door het doeltype tussen ronde haken vóór de variabele of vóór de bewerking te plaatsen: (doeltype) waarde.

Voorbeelden:

```
float fgetal = 15698.45F;
int igitel = (int) fgetal; //hier gaat de precisie verloren

byte resultaat = (byte) (17*2); //de literals 17 en 2 zijn default integers
```

#### 4.2.3 Prioriteitsregels

Belangrijk om weten is in welke volgorde de diverse operatoren worden uitgewerkt. Eerst worden de increment en decrement operatoren uitgewerkt, vervolgens de rekenkundige, dan de vergelijkingsoperatoren vóór de logische operatoren en tenslotte de toekenningsoperatoren. Binnen de rekenkundige operatoren heeft \*, / en % voorrang op + en -. Bij gelijke voorrang wordt de expressie van links naar rechts geëvalueerd.

De volledige lijst :

Prioriteit	Operator
Hoog	unaire operatoren in postfix notatie: <code>xyz++</code> , <code>xyz--</code> unaire operatoren in prefix notatie: <code>++xyz</code> , <code>--xyz</code> , <code>!</code> typeconversie: <code>(nieuwtype)xyz</code> <code>*</code> <code>/</code> <code>%</code> <code>+</code> <code>-</code> binaire shift-operatoren <code>&lt;&lt;</code> <code>&gt;&gt;</code> . Deze vallen buiten de scope van de cursus. vergelijkingsoperatoren <code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code> gelijkheidsoperatoren <code>==</code> , <code>!=</code> <code>&amp;</code> exclusieve of <code>^</code> <code> </code> <code>&amp;&amp;</code> <code>  </code> de conditionele operator <code>?</code> :
Laag	de toekeningsoperatoren: <code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>

#### 4.2.4 Stringoperatoren

Ook met strings kan je bewerkingen doen, namelijk het samenvoegen van strings met de `+` operator. We noemen dit concateneren. Java kan ook moeiteloos strings samenvoegen met variabelen van andere types. Strings worden later nog behandeld in een apart hoofdstuk.

### 4.3 Enkele praktijkvoorbeelden

#### 4.3.1 Eerste voorbeeld: Bmi

De bedoeling is om de bmi te berekenen uitgaande van een opgegeven lengte en gewicht. Maak in de package `jpfhfdst02` een nieuwe main class aan met naam *Bmi*. Klik rechts op de package *jpfhfdst02*, kies *Java Main Class...* (eventueel via optie *Other*) en voeg volgende code toe:

```
package jpfhfdst02;                                     (1)

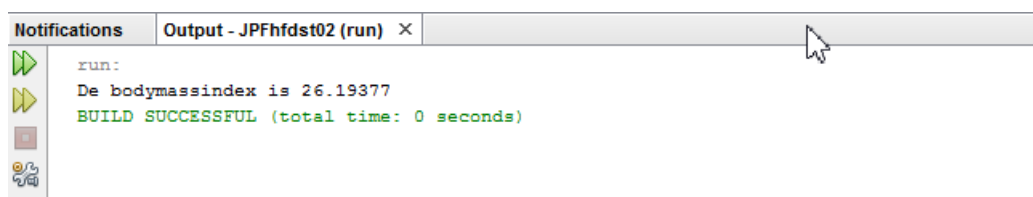
public class Bmi {                                       (2)
    public static void main(String[] args) {           (3)
        float gewicht = 75.7F;                         (4)
        float lengte = 1.70F;                         (5)
        float bmi;                                     (6)

        bmi = gewicht / (lengte * lengte);             (7)
        System.out.println("De bodymassindex is " + bmi); (8)
    }
}
```

Een woordje uitleg over de code:

- (1) Bovenaan het programma geeft de regel `package jpfhfdst02;` aan dat de class *Bmi.java* in een package (folder) met naam *jpfhfdst02* staat.
- (2) Hier begint de class definitie: `public class Bmi {` met eronder een aantal regels code die geïndenteerd zijn en tenslotte een `}`. Alles wat tussen accolades staat noemt men een *blok*. In dit blok vind je de beschrijving van een class met de naam *Bmi*.
- (3) De publieke statische functie: `public static void main(String[] args)` maakt van deze klasse een uitvoerbare klasse. Het is dus de bedoeling om hier uitvoerbare code te schrijven. Deze functie, de `main()`, wordt als eerste uitgevoerd wanneer je het programma runt.
  - `public`: betekent letterlijk publiek: kan aangesproken worden vanaf de “buitenwereld”.  
De JRE, die het programma uitvoert, moet toegang hebben tot de `main()`.
  - `static`: betekent dat dit een method is met class-bereik (dus uit te voeren zonder dat er een instance bestaat van de class). Hierop komen we later terug.
  - `void`: is de returnwaarde van de method: `void` betekent letterlijk ‘leeg’ of ‘niets’. Dat wil zeggen dat deze `main()` method geen waarde teruggeeft.
  - `main`: is de naam van de method. `main` is per definitie de naam van een uitvoerbare method. Wanneer een class deze method bevat, is het een uitvoerbare class.
  - `String[] arg`: Dit zijn de argumenten van de method. Eventueel kunnen er argumenten als parameter worden meegegeven. Ook hierop komen we verder in de cursus terug.
- (4) De declaratie en initialisatie van de variabele *gewicht* met type *float*, dus een reël getal. Deze variabele wordt meteen geïnitieerd op de waarde 75.7
- (5) Idem, doch een andere naam, nl. *lengte* en een andere waarde.
- (6) Dit is enkel een declaratie van de variabele *bmi*. Deze variabele heeft nog geen waarde. Je kan met deze variabele nog geen berekeningen maken, of de waarde ervan afdrukken.
- (7) De *bmi* wordt berekend en het resultaat van de berekening wordt toegekend aan de variabele *bmi*. Alle variabelen hebben hetzelfde type, nl. *float*. Hierdoor zijn er geen conflicten bij de berekening.
- (8) Met deze regel wordt het resultaat getoond, zijnde de tekst tussen de aanhalingstekens met daarna de waarde van de variabele *bmi*.

Wanneer je dit programma uitvoert, krijg je volgend output venster:



```
run:
De bodymassindex is 26.19377
BUILD SUCCESSFUL (total time: 0 seconds)
```

### 4.3.2 Tweede voorbeeld: Temperatuur

De bedoeling is om een opgegeven temperatuur in graden Celcius om te rekenen naar een temperatuur in graden Fahrenheit. Maak in de package `jpfhfdst02` een nieuwe main class aan met naam *Temperatuur*. Klik rechts op de package `jpfhfdst02`, kies *Java Main Class...* en voeg volgende code toe:

```
package jpfhfdst02;

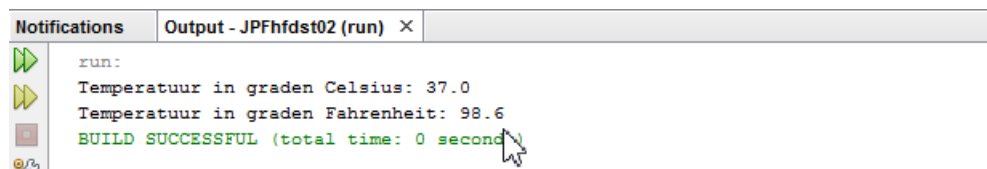
public class Temperatuur {
    public static void main(String[] args) {
        float tempCelsius = 37.0F;
        float tempFahr;
        tempFahr = tempCelsius * 9 / 5 + 32;

        System.out.println("Temperatuur in graden Celsius: " + tempCelsius);
        System.out.println("Temperatuur in graden Fahrenheit: " + tempFahr);
    }
}
```

Een woordje uitleg over de berekening:

- (1) Hier wordt de berekening uitgevoerd gebaseerd op de formule  $F = C \times 9 / 5 + 32$ . De literals 9, 5 en 32 zijn default van het type integer. De berekening wordt uitgevoerd van links naar rechts. Vermits de variabele `tempCelsius` van het type float is, levert de berekening van `tempCelsius * 9` ook een float op. Dit resultaat wordt gedeeld door 5 en levert ons opnieuw een float op. Hier wordt 32 bij opgeteld, dus het resultaat blijft een float. Daarom dat de variabele `tempFahr` een reël type moet zijn.

Wanneer je dit programma uitvoert, krijg je volgend output venster:



### 4.3.3 Derde voorbeeld: BTWnummer

De bedoeling is om een BTWnummer te controleren op geldigheid. Regels voor de controle van de geldigheid:

- Neem de eerste 7 cijfers van het BTWnummer en deel dit door 97
- 97 - de rest van deze deling = de laatste 2 cijfers van het BTW-nummer

Maak in de package `jpfhfdst02` een nieuwe main class aan met naam *BTWnummer*. Klik rechts op de package `jpfhfdst02`, kies *Java Main Class...* en voeg volgende code toe:

```
package jpfhfdst02;

public class BTWnummer {
    public static void main(String[] args) {
        int btwNummer = 213252520;
    }
}
```

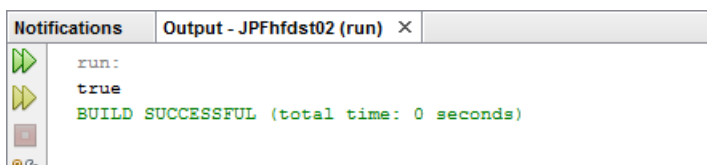


```
int deeltal = btwNummer / 100;           (2)
byte rest = (byte) (deeltal % 97);      (3)
byte laatste2Cijfers = (byte) (btwNummer % 100); (4)
System.out.println(laatste2Cijfers == 97-rest); (5)
}
```

Een woordje uitleg over de code:

- (1) De declaratie en initialisatie van een BTWnummer (een int is groot genoeg)
- (2) Om de eerste 7 cijfers van het BTWnummer te nemen, deel je het BTWnummer door 100. Vermits beide operanden van de deling een geheel getal zijn, is het resultaat ook een geheel getal, m.a.w. zondermeer de eerste 7 cijfers. De rest wordt genegeerd.
- (3) Vervolgens bepaal je de rest van de deling van dit getal door 97. Door casting zet je het type van het resultaat van de bewerking om van *int* naar *byte*. Vermits beide operanden van de % bewerking van het type *int* zijn, is het resultaat ook een *int*. Omdat je het resultaat in een *byte* wil bewaren, is de casting naar *byte* noodzakelijk. Wanneer je dit niet doet en volgende regel code typt: `byte rest = deeltal % 97;` geeft NetBeans in de marge een fout aan. Deze fout slaat op het conversieprobleem en de mogelijkheid van gegevensverlies. Om er voor te zorgen dat het resultaat geconverteerd wordt naar een *byte* en niet enkel de variabele *deeltal*, dien je de bewerking tussen haken te plaatsen.
- (4) De rest van de deling van het BTWnummer door 100 levert je de laatste 2 cijfers op. Ook hier zet je door casting het type om van *int* naar *byte*.
- (5) `laatste2Cijfers == 97-rest` levert je *true* of *false* op. Je gaat na of beide operanden aan elkaar gelijk zijn. Als dit zo is, is dat *true*. In het andere geval is dat *false*.

Wanneer je dit programma uitvoert, krijg je volgend output venster:



In het hoofdstuk Programmaverloop krijgen we nog meer voorbeelden te zien van bewerkingen, operatoren, casting en strings.

## 4.4 Invoer door de gebruiker via toetsenbord

In de voorbeelden zijn de variabelen meteen geïnitieerd door er een waarde aan toe te kennen. Soms kan het nodig zijn dat de gebruiker een waarde dient in te geven. Dit kan je doen door gebruik te maken van een Scanner. Dit is een bestaande Java klasse in de verzameling *java.util*.

Volgend voorbeeld toont je hoe dit kan.

Maak in de package *jpfhfdst02* een nieuwe main class aan met naam *Invoer*. Klik rechts op de package *jpfhfdst02*, kies *Java Main Class...* en voeg volgende code toe:

```
package jpfhfdst02;
```

```
public class Invoer {  
    public static void main(String[] args) {  
        System.out.print("Geef een getal: ");  
        Scanner sc = new Scanner(System.in);           (1)  
        int getal = sc.nextInt();                       (2)  
    }  
}
```

- (1) NetBeans geeft een fout aan op deze regel. Scanner is niet gekend. Wanneer je de muis laat rusten op het lampje in de grijze regelnummering krijg je de eigenlijke foutmelding. Wanneer je Alt-Enter typt (of gewoon klikt op het lampje) geeft NetBeans een aantal suggesties om het probleem op te lossen. In dit geval kies je voor *Add import for java.util.Scanner*. Onder de package instructie maar boven de class definitie wordt de regel

```
import java.util.Scanner;
```

toegevoegd waardoor de klasse *Scanner* gekend is. Door de import wordt de bestaande klasse *Scanner* geïmporteerd. Belangrijk is dat je de volledige naam van de klasse opgeeft, dus *java.util.Scanner* (de naam inclusief de package-naam). Zonder import kan er geen gebruik gemaakt worden van deze bestaande klasse.

De variabele *sc* is een scanner-object en we geven aan dat we het toetsenbord (*System.in*) gebruiken voor invoer.

- (2) Eigenlijke invoer van een getal gebeurt met de method *nextInt()*. Dit levert je een geheel getal op (een integer). Er bestaan nog andere methods zoals *nextFloat()* voor een float, *next()* voor een String, enz...
- De invoer van de gebruiker wordt bewaard in een variabele van het juiste type en kan verder in het programma gebruikt worden.

Later krijg je nog meer uitleg over klassen en objecten.

## 4.5 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 4:

- Student scores
- Omrekening seconden
- Snoepautomaat
- Bankrekeningnummer

## 5 Arrays

---

In dit hoofdstuk behandelen we arrays. Arrays zijn tabellen van primitive data types of van klassen. Ze worden ook wel reeksen of lijsten genoemd.

### 5.1 Arrays van primitive data types of strings

Arrays hebben twee belangrijke eigenschappen. Het zijn geordende reeksen van genummerde elementen. Je kan dus elk element uit een array apart benaderen via de naam van de array en het volgnummer van het element. Daarnaast moeten alle elementen van de array van hetzelfde type zijn. Bijv. alle elementen zijn van het type `int`, of van het type `float`, of van het type `char`, of van het type `String`,...

Om arrays te kunnen gebruiken, dienen deze net zoals gewone variabelen eerst gedeclareerd te worden. Een arraydeclaratie heeft de volgende syntax:

```
type[] naamarray;
```

Ter vollediging wordt vermeld dat `type naamarray[];` ook werkt. Dit is niet de voorkeur syntax en wordt daarom verder niet gebruikt.

Enkele voorbeelden:

```
byte[] lottogetallen;  
float[] lonen;  
double[] metingen;  
String[] voornamen;  
char[] alfabet;
```

Met deze declaratie reserveer je in het geheugen een verwijzing of reference naar een array. Op dit moment heb je dus nog geen array met elementen maar wel een reference die naar een array verwijst.

Om geheugen te reserveren voor de elementen in de array dien je de array te creëren met de volgende syntax:

```
naamarray = new type[aantalelementen];
```

Enkele voorbeelden:

```
lottogetallen = new byte[7];  
lonen = new float[50];  
metingen = new double[10];  
voornamen = new String[3];  
alfabet = new char[26];
```

Je kan de declaratie en creatie ook in **één statement** plaatsen:

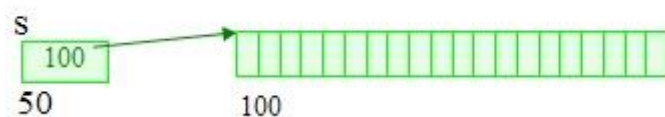
```
type[] naamarray = new type[aantalelementen];
```

Voor de voorbeelden geldt dan:

```
byte[] lottogetallen = new byte[7];
float[] lonen = new float[50];
double[] metingen = new double[10];
String[] voornamen = new String[3];
char[] alfabet = new char[26];
```

Een grafische voorstelling:

```
char[] s = new char[20];
```



Geheugenplaats 50 is gereserveerd voor de arrayvariabele **s** die verwijst naar geheugenplaats 100 waar er 20 chars kunnen gestockeerd worden. 100 is het beginadres van de array en dus ook van het eerste element van de array.

In tegenstelling met andere variabelen, worden de elementen van een tabel automatisch geïnitieerd. Afhankelijk van het type van de tabelelementen krijgen zij een andere default-waarde:

- Elementen van het type **int** worden geïnitieerd op **0**
- Elementen van het type **char** worden geïnitieerd op **0**
- Elementen van het type **float** worden geïnitieerd op **0.0**
- Elementen van het type **double** worden geïnitieerd op **0.0**
- Elementen van het type **boolean** worden geïnitieerd op **false**
- Elementen van het type **String** worden geïnitieerd op **null**

Een voorbeeld:

```
package jpfhfdst05;

public class JPFhfdst05 {

    public static void main(String[] args) {

        System.out.println("--- Impliciete initialisatie array ---");
        int[] getallen = new int [3];
        System.out.println("1e element uit int tabel:" + getallen[0]);
        System.out.println("2e element uit int tabel:" + getallen[1]);
        System.out.println("3e element uit int tabel:" + getallen[2]);
        System.out.println("");

        int[] chars = new int [3];
        System.out.println("1e element uit char tabel:" + chars[0]);
        System.out.println("2e element uit char tabel:" + chars[1]);
        System.out.println("3e element uit char tabel:" + chars[2]);
        System.out.println("");
    }
}
```

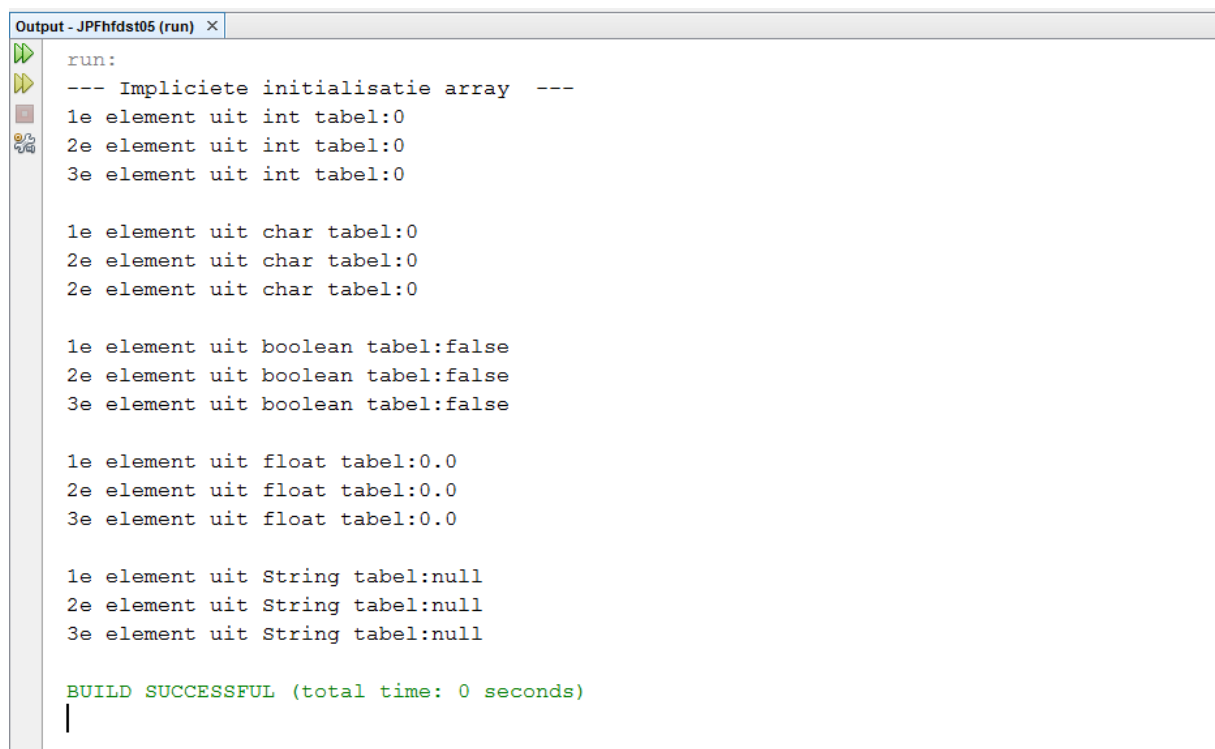
```
boolean[] vlaggen = new boolean [3];
System.out.println("1e element uit boolean tabel:" + vlaggen[0]);
System.out.println("2e element uit boolean tabel:" + vlaggen[1]);
System.out.println("3e element uit boolean tabel:" + vlaggen[2]);
System.out.println("");

float[] kommaGetallen = new float [3];
System.out.println("1e element uit float tabel:" + kommaGetallen[0]);
System.out.println("2e element uit float tabel:" + kommaGetallen[1]);
System.out.println("3e element uit float tabel:" + kommaGetallen[2]);

System.out.println("");

String[] namen = new String [3];
System.out.println("1e element uit String tabel:" + namen[0]);
System.out.println("2e element uit String tabel:" + namen[1]);
System.out.println("3e element uit String tabel:" + namen[2]);
System.out.println("");
}
}
```

Na uitvoer van dit programma zie je volgend resultaat in het **Output** venster:



```
Output - JPFhdst05 (run) x
run:
--- Impliciete initialisatie array ---
1e element uit int tabel:0
2e element uit int tabel:0
3e element uit int tabel:0

1e element uit char tabel:0
2e element uit char tabel:0
2e element uit char tabel:0

1e element uit boolean tabel:false
2e element uit boolean tabel:false
3e element uit boolean tabel:false

1e element uit float tabel:0.0
2e element uit float tabel:0.0
3e element uit float tabel:0.0

1e element uit String tabel:null
2e element uit String tabel:null
3e element uit String tabel:null

BUILD SUCCESSFUL (total time: 0 seconds)
|
```

Je kan bij het reserveren van het geheugen de arrayelementen ook onmiddellijk initialiseren.  
Bijvoorbeeld:

```
lottogetallen = new byte[] {12,15,21,23,30,40,17};
lonen = new float[] {1250.63, 1310.25, 1546.88};
voornamen = new String[] {"Lode", "Vie"};
```

Op deze manier is het niet nodig om de grootte van de array op te geven. Dit wordt bepaald door het aantal elementen dat tussen de accolades staat. De array lottogetallen krijgt hier een grootte van 7 elementen en de array lonen is slechts 3 elementen groot. Ook op deze manier ligt het aantal

elementen van de array onherroepelijk vast.

Het aanspreken van de diverse elementen van een array gebeurt aan de hand van de naam van de array en het volgnummer van het element binnen de array. De volgnummers van de elementen gaan van 0 tot en met  $n-1$ , waarbij  $n$  het aantal elementen is.

Het wijzigen van het 3<sup>e</sup> lottogetal gaat dus als volgt :

```
lottogetallen[2] = 19;
```

Het volgnummer dat elk element in de array bepaalt wordt **index** genoemd. De index kan dus gaan van 0 tot  $n-1$ .



Het gaat om **eendimensionale arrays**. Een eindimensionale array is een array waarbij een element benaderd kan worden via **één index**.

Je kan makkelijk nagaan hoeveel elementen er in de array zitten via de `length`-property van de array:

```
naamarray.length
```

Het wijzigen van het laatste lottogetal kan dan als volgt:

```
lottogetallen[lottogetallen.length-1] = 42;
```



De grootte van een array ligt vast en kan niet gewijzigd worden.

Om de tabel toch groter te maken, kan je gebruik maken van de method `arraycopy`. Deze method maakt deel uit van de library `System` (`System.arraycopy`). Zo bestaan er nog vele andere methods. Ze maken deel uit van de `java-libraries`. Voor het gebruik ervan wordt verwezen naar de API documentatie.

## 5.2 Arrays van objecten

In een later hoofdstuk komen klassen aan bod. Toch wordt er nu reeds even aandacht besteed aan een array van objecten. De declaratie van een array van objecten (klassen) verschilt niet van deze van primitive data types of Strings. Een String is bovendien ook een klasse. Een voorbeeld:

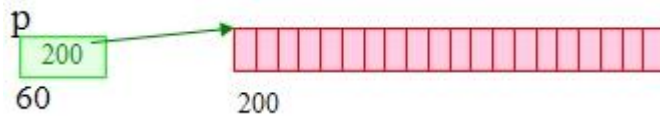
```
Coordinaat[] p;
```

Er is nu nog geen array maar wel een reference *p* die verwijst naar een array van Coördinaat-objecten. Deze reference kan je laten verwijzen naar een bestaande array van Coördinaat-objecten of naar een nieuwe array. Bijvoorbeeld :

```
p = new Coördinaat[20];
```

Op deze manier is er een array van 20 references naar objecten van de class Coördinaat gemaakt. De 20 references verwijzen voorlopig nog nergens naar.

Schematisch :



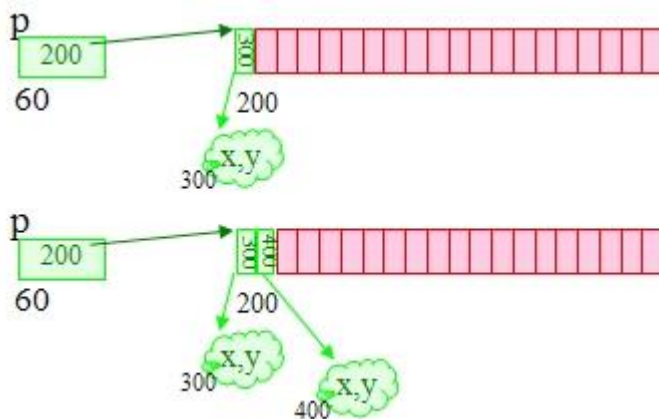
In de figuur zien we de arrayvariabele *p* op geheugenplaats 60 die verwijst naar een array van references naar objecten van de class Coördinaat vanaf geheugenplaats 200.

Je kan nu de references naar een nieuw object of een reeds bestaand object van de class Coördinaat laten verwijzen. Bijvoorbeeld:

```
p[0] = new Coördinaat(); //creatie van een nieuw object Coördinaat  
p[1] = q                 //q is een bestaand object
```

Schematisch:

Veronderstel dat de variabele *q* verwijst naar het geheugenadres 400 waar reeds een object van Coördinaat bewaard wordt



In bovenstaande figuur zien we opnieuw de arrayvariabele *p*. Na het eerste statement is er een nieuw object van de class Coördinaat gemaakt (op geheugenadres 300) en het eerste element van de array verwijst hiernaar.

Het onderste deel van de afbeelding is het resultaat na het tweede statement. Het tweede element van de array verwijst naar geheugenadres 400. Dit is de plaats waarnaar de reference *q* verwijst.

### 5.3 Arrays van arrays

Tot nu toe zijn eendimensionale arrays besproken. Java kent geen meerdimensionale arrays, of arrays waarvan je de elementen aanspreekt met 2 of meer indexen.

Een arrayelement `coordinaten[1, 3]` bestaat dus niet.

Daarentegen kan je wel arrays van arrays maken. Een voorbeeld :

```
float[][] temperaturen = new float [10][5];  
temperaturen[0][0] = 18.25;
```

In bovenstaand voorbeeld wordt een array *temperaturen* van het type float gedeclareerd. De array heeft 10 elementen die alle 10 naar een array van 5 floats verwijzen.

Het eerste element van de eerste array krijgt de waarde 18.25.

De verschillende elementen van de array hoeven echter niet allemaal naar een array van eenzelfde aantal elementen te wijzen (zoals in het voorbeeld naar een array van 5 floats). De dimensie van deze arrays kan verschillend zijn. Een voorbeeld :

```
int[][] getallen = new int[4][];  
//voor elk element in de array wordt er een array van ints gemaakt  
getallen[0] = new int[5];  
getallen[1] = new int[7];  
getallen[2] = new int[3];  
getallen[3] = new int[4];
```

In dit voorbeeld wordt een array van 4 elementen gedefinieerd, waarvan elk element verwijst naar een array van integers. Vervolgens wordt er geheugenplaats voorzien voor 4 arrays, allemaal met een verschillende dimensie. Merk op dat bij de declaratie het aantal elementen bij de tweede dimensie niet wordt opgegeven. Het aantal elementen van de eerste dimensie is verplicht.

Uiteraard dient wel de lengte van elke array goed in de gaten gehouden te worden, zodat je geen onbestaand element aanspreekt.

`getallen[1][5]` verwijst naar rij 2, het 6e element

`getallen[1]` verwijst naar rij 2, op zich dus naar een éénimensionale array



In het hoofdstuk Programmaverloop worden nog meer voorbeelden van het gebruik van arrays gezien.

### 5.4 Oefeningen



Zie takenbundel: maak de oefening die hoort bij hoofdstuk 5:  
- Array van vijf integers



## 6 Programmaverloop

---

### 6.1 Blokken van statements en scope van variabelen

In Java worden statements gegroepeerd in blokken. Accolades geven het begin en het einde van een blok aan. Blokken kunnen ook genest worden. Dus binnen een blok kan je een ander blok schrijven. Herneem het eerste voorbeeld:

```
package jpfhfdst02;
public class JPFhfdst02 {
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("Hello World !");
    }
}
```

De main() method staat in een blok en dit blok staat op zich in het blok van de class JPFhfdst02.

Het nut van blokken binnen blokken heeft zijn doel voor de **scope** van variabelen. Met scope bedoelen we het bereik en de levensduur van een variabele. Variabelen die binnen een blok zijn gedefinieerd 'leven' slechts tot aan het einde van dat blok. In het onderstaande voorbeeld merk je dat er een foutaanduiding optreedt bij variabele b. Je wil immers deze variabele gebruiken buiten het blok waar b gedeclareerd is.

```
package jpfhfdst06;
public class JPFhfdst06 {
    public static void main(String[] args) {
        int a = 5;
        {
            //Dit is een blok binnen het mainblok
            int b = 6;
            System.out.println("a is " + a);
            System.out.println("b is " + b);
        }
        System.out.println("a is " + a);
        System.out.println("b is " + b); → bij deze regel verschijnt een foutaanduiding
    }
}
```

De foutaanduiding geeft aan dat de variabele b niet gekend is in dit blok. De variabele b is enkel gekend in het binnenste blok.

### 6.2 Keuzestructuren

Er bestaan in java drie keuzestructuren:

- de *if*,
- de *conditional operator* ?
- en de *switch*.

### 6.2.1 De if-instructie

Afhankelijk van de situatie moet er code soms wel of niet uitgevoerd worden. Evenzeer kan het zijn dat afhankelijk van de situatie andere code uitgevoerd moet worden.

Hiervoor kan de *if-instructie* gebruikt worden. Na de *if* plaats je een *expressie* die moet resulteren in een *boolean*. Afhankelijk van de waarde van deze boolean-expressie wordt een regel code (statement) of een blok code al dan niet uitgevoerd. De expressie of voorwaarde moet tussen *ronde haken* staan. De syntax:

```
if (expressie) {  
    statements  
}
```

Een voorbeeld met één enkel statement:

```
int getal = 6;  
if (getal % 2 == 0)  
    System.out.println("Het getal " + getal + " is even.");
```

Wanneer de expressie of voorwaarde onwaar is, en er dient in dat geval andere code uitgevoerd te worden, kan de *if*-instructie uitgebreid worden met een *else*. De else-tak is niet verplicht. De syntax:

```
if (expressie) {  
    statements  
}  
else {  
    statements  
}
```

Vorig voorbeeld uitgebreid met *else*:

```
int getal = 7;  
if (getal % 2 == 0)  
    System.out.println("Het getal " + getal + " is even.");  
else  
    System.out.println("Het getal " + getal + " is oneven.");
```

In geval er meerdere statements uitgevoerd dienen te worden, hetzij in de *if*, hetzij in de *else*, dienen deze statements in een blok geplaatst te worden. Tussen accolades dus, zoals bij de syntax is weergegeven.

Het voorbeeld uitgebreid met een blok :

```
int getal = 6;  
int getal2;  
if (getal % 2 == 0) {  
    System.out.println("Het getal " + getal + " is even.");  
    getal2 = getal * 3;  
}
```

Een volledige if-else-structuur is op zich ook één statement.



**Best practice:** gebruik **altijd** een blok (accolades) bij de if-else structuur.  
Het bevordert de leesbaarheid en dus ook de onderhoudbaarheid van de code.

If-else-constructies kan je ook nesten. Een if-blok en/of een else-blok kan een nieuwe if-constructie of if-else-constructie zijn.

### 6.2.2 De conditional operator ?

In sommige gevallen is de *conditional operator* ? een alternatief voor de if-instructie. De ? operator is een ternaire operator wat betekent dat de operator drie operanden wenst: een test, een waarde indien voldaan en een waarde indien niet voldaan. De syntax :

```
expressie ? waardeIndienVoldaan : waardeIndienNietVoldaan
```

Het resultaat van deze constructie is een waarde. Een voorbeeld :

```
int a = 5;  
int b = 7;  
int grootste = a < b ? b : a;  
System.out.println("grootste is " + grootste);
```

De variabele *grootste* wordt in dit voorbeeld geïnitieerd op de grootste waarde van *a* en *b*. In bovenstaand geval zal *grootste* de waarde van *b* krijgen, dus 7.

### 6.2.3 De switch

Bij een if-instructie zijn er maar 2 mogelijkheden: de test is *true* of *false*. Indien er een keuze gemaakt dient te worden tussen meerdere mogelijkheden, dan kan een geneste if-instructie een oplossing bieden, maar meestal wordt de constructie dan zeer onleesbaar.

De switch kan hier een uitweg bieden. Een *expressie* wordt geëvalueerd en vergeleken met een aantal mogelijkheden. Is het resultaat van de expressie verschillend van elk van deze mogelijkheden, dan kan er een *defaultwaarde* worden uitgevoerd.

De syntax :

```
switch (expressie) {  
    case waarde1:  
        statements;  
        break;  
    case waarde2:  
        statements;  
        break;  
}
```

```

        case waarde3:
        case waarde4:
        case waarde5:
            statements;
            break;
        default:
            statements;
            break;
    }

```

Werking van de switch:

- De expressie van switch controleert steeds op gelijkheid. Een vergelijkingsoperator kan hier niet gebruikt worden.
- Er kunnen een willekeurig aantal case-labels zijn.
- De instructie *break* na de statements is nodig omdat dan de switch beëindigd wordt. Wanneer *break* er niet staat, loopt de uitvoering van de switch verder en worden alle statements uitgevoerd totdat een eerstvolgende break wordt bereikt. Het is een good practice om ook het break-statement te schrijven bij het *default*-label.
- Indien dezelfde statements uitgevoerd dienen te worden in meerdere gevallen, volstaat het om voor deze gevallen de case-labels te schrijven en de statements slechts één keer te schrijven bij het laatste case-label (de statements beschreven bij waarde5 gelden ook voor waarde3 en waarde4).
- Het *default*-label is optioneel. De statements hier worden uitgevoerd wanneer geen van de andere gevallen voorkomen.
- Mogelijke types voor de variabele van de expressie zijn *byte*, *short*, *char*, *int* en *enum*. Vanaf Java 7 kunnen de *expressie* en de *case*-labels ook strings zijn. De vergelijking op gelijkheid bij strings gebeurt dan volgens de `String.equals` method (zie verder in de cursus). Dat betekent dus ook case-sensitive.

Een voorbeeld met integers:

```

System.out.println("Geef een keuze in van 1 tot 3: ");
Scanner sc = new Scanner(System.in);
int menuKeuze = sc.nextInt();                                     (1)
switch (menuKeuze) {
    case 1:                                                         (2)
        System.out.println("Keuze 1");                             (3)
        break;                                                       (4)
    case 2:                                                         (5)
        System.out.println("Keuze 2");
        break;
    case 3:                                                         (6)
        System.out.println("Keuze 3");
        break;
    default:                                                         (7)
        System.out.println("Geen keuze 1 tot 3 gegeven");
        break;
}

```

- (1) Een menukeuze kan ingegeven worden via het toetsenbord en wordt bewaard in de variabele menuKeuze

- (2) Wanneer menuKeuze gelijk is aan 1 worden de statements bij dit case-label uitgevoerd;
- (3) De opgegeven tekst wordt getoond
- (4) En de switch wordt beëindigd
- (5) Idem maar voor menuKeuze gelijk aan 2
- (6) Idem maar voor menuKeuze gelijk aan 3
- (7) Wanneer de menuKeuze niet gelijk is aan de voorgaande case-labels, dus in geval van 1, 2 en 3, worden de statements bij het defaultlabel uitgevoerd.

Een voorbeeld met strings:

```
System.out.println("Geef een korte letterreeks in om een dag aan te  
duiden (bijv. ma, di, woe, don, vr, zat, zon): ");  
Scanner sc = new Scanner(System.in);  
String dagKort = sc.next();
```

 (1)

```
String dagLang;  
switch (dagKort) {  
    case "ma":  
        dagLang = "maandag";  
        break;  
    case "di":  
        dagLang = "dinsdag";  
        break;  
    case "woe":  
        dagLang = "woensdag";  
        break;  
    case "don":  
        dagLang = "donderdag";  
        break;  
    case "vr":  
        dagLang = "vrijdag";  
        break;  
    case "zat":  
        dagLang = "zaterdag";  
        break;  
    case "zon":  
        dagLang = "zondag";  
        break;  
    default:  
        dagLang = "onbekend";  
        System.out.println("Geen geldige dag");  
        break;  
}
```

 (2)

```
System.out.println("Dag is " + dagLang);
```

 (3)

- (1) Zo kan er een dag in korte notatie ingegeven worden.
- (2) Het switch-statement zet de korte dagnotatie om in een lange dagnotatie.
- (3) Na de switch wordt deze lange dagnotatie getoont.

## 6.3 Lussen

Lussen of herhalingen geven je de mogelijkheid om één of meerdere statements een herhaald aantal keer te laten uitvoeren. Hoeveel keer hangt af van de voorwaarde. Een herhaling wordt ook een iteratie genoemd. Java kent 4 soorten iteraties:

- **while:** een iteratie met de test vóór het te herhalen blok

```
while (expressie) {  
    statements;  
}
```

- **do...while:** een iteratie met de test ná het te herhalen blok

```
do {  
    statements;  
}  
while (expressie);
```

- **for:**

```
for (initialisatie ; voorwaarde ; increment) {  
    statements;  
}
```

- **for-each-lus:**

```
for (type variabele : verzameling) {  
    statements;  
}
```

### 6.3.1 De while-lus

Je gebruikt een *while-lus* om een aantal statements te herhalen zolang aan een bepaalde voorwaarde is voldaan. Deze voorwaarde wordt telkens getest vooraleer de statements worden uitgevoerd.

Een voorbeeld :

```
int teller = 1; (1)  
while (teller <= 10) { (2)  
    System.out.println(teller); (3)  
    teller++; (4)  
}
```

- (1) Een teller wordt geïnitieerd op 1.
- (2) De voorwaarde dient tussen ronde haken te staan. Het resultaat van de voorwaarde is van het type boolean, dus true of false. Zolang de voorwaarde waar (true) is, worden de statements van het blok uitgevoerd.

- (3) De waarde van teller wordt getoond.
- (4) De waarde van teller wordt verhoogd met 1. Dit is het einde van het blok statements en dus zal de lus bovenaan hervatten. De expressie wordt opnieuw gevalideerd en afhankelijk van het resultaat van de expressie wordt de code in de lus opnieuw uitgevoerd of wordt er verder gegaan met de code na de lus.

Het is aanbevolen om de statements van de lus steeds tussen accolades te plaatsen, ook al gaat het om één enkel statement.

Indien de beginwaarde van de teller bij aanvang van de lus een waarde  $> 10$  heeft, zal er niets uitgevoerd worden. Bij aanvang van de lus is de eerste expressie al meteen onwaar (false) waardoor er geen statements van de lus worden uitgevoerd. Er wordt meteen verder gegaan met de code na de lus.

### 6.3.2 De do...while-lus

In tegenstelling tot de *while-lus* staat de test bij de *do...while* helemaal onderaan, na het iteratieblok. De lus wordt dus minstens één maal doorlopen.

Een voorbeeld :

```
int teller = 1;
do {
    System.out.println(teller);
    teller++;
}
while (teller <= 10);
```

De expressie wordt steeds getest nadat de statements van de lus zijn uitgevoerd. Dit zorgt er voor dat de lus altijd minimaal één keer wordt uitgevoerd. Dit is een belangrijk verschil met de *while-lus*!

Wanneer in het bovenstaand voorbeeld de teller bij aanvang bijv. waarde 15 heeft, worden de statements van het blok toch één keer uitgevoerd.

### 6.3.3 De for-lus

Bij de *for-lus* worden een aantal statements herhaald zolang aan een bepaalde voorwaarde voldaan is. De *for-lus* bestaat uit 3 delen: een **initialisatie**, een **voorwaarde** om in de lus te blijven en een **increment** (iteratie-expressie):

- de **initialisatie** wordt precies één enkele keer uitgevoerd voordat de lus start. Je kan er dus perfect een variabele in declareren en initialiseren. Wanneer je hier een variabele declareert, is deze na de *for-lus* niet meer gekend.
- De **voorwaarde** wordt steeds geëvalueerd voordat de body van de lus opnieuw wordt uitgevoerd. Zo is het mogelijk dat de statements van de lus zelfs nooit worden uitgevoerd.
- Telkens na het uitvoeren van de lus, wordt het **increment**-statement uitgevoerd.

Een voorbeeld :

```
int teller;
for (teller = 1 ; teller <= 10; teller++) {
    System.out.println(teller);
}
```

```
}
```

Bemerk dat deze code bondiger is dan de twee vorige lusstructuren. Bovendien zou je hier zelfs de accolades kunnen weglaten, aangezien het te herhalen blok slechts uit één enkel statement bestaat. Dit laatste wordt nochtans niet aanbevolen.

Tot slot kan nog gesteld worden dat een for-lus vaak gebruikt wordt om de body van de lus een specifiek aantal maal uit te voeren.



**Best practice:** gebruik **altijd** een blok (accolades) voor de body van de lus.  
Het bevordert de leesbaarheid en dus ook de onderhoudbaarheid van de code.

### 6.3.4 De for-each-lus

Een variant op de for-lus is de for-each-lus. Met de for-lus kan je iets herhaaldelijks uitvoeren, maar je geeft een startpunt en een eindpunt op. Met de for-each-lus itereer je over een verzameling, bijv. een array. Een voorbeeld:

```
int[] getallen = new int[5];  
for (int teller=0 ; teller <=4; teller++) {  
    getallen[teller] = teller + 20;  
}
```

(1)

```
for (int getal : getallen ) {  
    System.out.println(getal);  
}
```

(2)

- (1) Deze for-lus wordt vijf maal doorlopen. Ze vult de gedeclareerde int-array met de waarden 20 tot en met 24. Het eerste element in de array heeft de waarde 20, het tweede element de waarde 21, enz. tot het laatste, het vijfde, element de waarde 24 krijgt.
- (2) Hier is de for-each-lus gebruikt om te itereren over deze array. De volledige array *getallen* wordt doorlopen van het begin tot het einde. Elk element uit de array wordt beschouwd als een integer, zijnde *getal*. In de `System.out.println(getal)` ; wordt dit *getal* weergegeven.



Deze **variant** van de for-lus, de **for-each-lus**, wordt veel gebruikt bij het itereren over objecten in collections. Zie later in deze cursus.

### 6.3.5 Breaks en labels

Tot nu toe werd de lus telkens beëindigd wanneer niet langer aan een bepaalde voorwaarde werd voldaan. Toch kan het zijn dat de lus om één of andere onvoorziene omstandigheid eerder moet beëindigd worden. Dit kan je doen door het break statement te gebruiken.



Volgens de principes van gestructureerd programmeren zou je die 'onvoorziene omstandigheid' mee moeten testen in de lus-voorwaarde. Er zijn tal van redenen te bedenken waarom je dit niet zou doen. Eén ervan is dat de testvoorwaarde, die bijgevolg een samengestelde voorwaarde wordt, er al snel heel complex kan uitzien.

Toepassing van het break-statement:

```
for (initialisatie ; voorwaarde ; increment) {
    ... statements ...
    if (test)
        break;
    ... statements ...
}
// uitvoering gaat hier verder na de break
```

Wanneer de test *waar* oplevert wordt het break-statement uitgevoerd wat betekent dat de lus gestopt wordt en de uitvoering verdergaat na het blok van de for-lus.

De break beëindigt de omsluitende lus. Hiervan kan afgeweken worden wanneer je gebruik maakt van een label. De naam voor dit label bepaal je zelf.

```
breekpunt:
for (initialisatie ; voorwaarde ; increment) {
    ... statements ...
    for (initialisatie ; voorwaarde ; increment) {
        if (test)
            break breekpunt;
        ... statements ...
    }
}
// uitvoering gaat hier verder na de break breekpunt
```

Het verloop van de lus kan ook beïnvloed worden door gebruik te maken van het *continue-statement*. In tegenstelling tot de break verlaat je de lus niet definitief, maar spring je naar het einde van de lus, dan wordt de increment uitgevoerd en vervolgens wordt er opnieuw getest om eventueel in de lus te blijven.

Toepassing van het continue-statement:

```
for (initialisatie ; voorwaarde ; increment) {
    ... statements ...
    if (test)
        continue;
    ... statements ...
}
// de bovenstaande statements worden overgeslagen door continue
```

Wanneer de test *waar* oplevert wordt het continue-statement uitgevoerd wat betekent dat de statements van de lus niet verder worden uitgevoerd. De lus wordt niet beëindigd, doch er wordt naar het einde van de lus gegaan, de incrementopdracht wordt uitgevoerd en vervolgens wordt nagegaan of de lus verder kan lopen.



Je kan met breaks, continues en labels de code nodeloos complex maken. Tracht dit zoveel mogelijk te vermijden en dus enkel gepast te gebruiken.

## 6.4 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 6:

- Array van vijf integers (met iteraties)
- Randomgenerator
- Randomgenerator2
- Lotto
- Huisdieren

## 6.5 Procedures en functies

### 6.5.1 Procedures

Aan de hand van een voorbeeld wordt er uitgelegd hoe je in Java procedures schrijft.

Neem volgend voorbeeld: een tabel van 50 elementen wordt opgevuld met randomgetallen tussen 1 en 1000. Deze tabel wordt ongesorteerd getoond. Vervolgens wordt de tabel gesorteerd en opnieuw getoond.

```
import java.util.Arrays;
public class VoorbeeldProcedure {
    public static void main(String[] args) {
        int[] getallen = new int[50];

        // GENEREREN van 50 willekeurige getallen
        for(int i = 0; i < getallen.length; i++) {
            getallen[i] = (int) (Math.random()*1000 + 1);
        }
        // Toon ONGESORTEERD resultaat
        System.out.println("Ongesorteerd");
        for (int i = 0; i<getallen.length; i++) {
            System.out.print(getallen[i] + "\t" );
        }
        System.out.println("");

        // SORTEREN van de 50 getallen
        Arrays.sort(getallen);

        // Toon GESORTEERD resultaat = HERHALING VAN CODE!!
        System.out.println("Gesorteerd");
        for (int i = 0; i<getallen.length; i++) {
            System.out.print(getallen[i] + "\t" );
        }
    }
}
```

```
}
```

Het tonen van de tabel is een stukje code dat dubbel geschreven is. Hoe weinig code het ook is, het schrijven van dubbele code dient zoveel mogelijk vermeden te worden. Dit kan m.b.v. een procedure opgelost worden:

```
import java.util.Arrays;
public class VoorbeeldProcedureB {
    public static void main(String[] args) {
        int[] getallen = new int[50];

        // GENEREREN van 50 willekeurige getallen
        for(int i = 0; i < getallen.length; i++) {
            getallen[i] = (int) (Math.random()*1000 + 1);
        }
        // Toon ONGESORTEERD resultaat
        toonTabel(getallen, "Ongesorteerd");           (1)

        // SORTEREN van de 50 getallen
        Arrays.sort(getallen);

        // Toon GESORTEERD resultaat
        toonTabel(getallen, "Gesorteerd");           (2)

    } //einde main

    private static void toonTabel( int[] teTonenTabel, String titel) {
        System.out.println("\n" + titel);
        for (int i = 0; i < teTonenTabel.length; i++) {
            System.out.print(teTonenTabel[i] + "\t" );
        }
    }
} //einde class VoorbeeldProcedureB
```

De uitvoer van het programma is als volgt (enkel de getallen worden op één regel getoond):

```
Ongesorteerd
369  51   5   131  670  317  59   682  305  106  995  433  959
172  354  53   909  318  274  474  617  601  818  221  190  547
985  522  496  152  192  706  572  564  638  46   664  829  149
695  134  807  818  612  268  387  7    397  937  415

Gesorteerd
5    7    46   51   53   59   106  131  134  149  152  172  190
192  221  268  274  305  317  318  354  369  387  397  415  433
474  496  522  547  564  572  601  612  617  638  664  670  682
695  706  807  818  818  829  909  937  959  985  995
```

Er is een aparte procedure geschreven voor het tonen van de tabel. Deze procedure wordt geschreven na de main(), maar binnen de class. Een beetje toelichting:

```
private static void toonTabel( int[] teTonenTabel, String titel)
```

- **private:** betekent letterlijk privaat: kan niet aangesproken worden vanaf de “buitenwereld”. Deze method kan enkel aangesproken worden vanuit de class waar ze in geschreven is, dus hier vanuit VoorbeeldProcedureB.

- `static`: deze method dient static te zijn vermits ze wordt aangeroepen vanuit de main, die op zich ook static is. We komen later nog terug op het begrip static.
- `void`: is de returnwaarde van de method: void betekent letterlijk 'leeg' of 'niets'. Dat wil zeggen dat deze method geen waarde teruggeeft. Deze method voert enkel de code uit, maar geeft niets terug. We spreken dan ook van een procedure en geen functie.
- `int[] teTonenTabel`: dit is het eerste argument van de method. Het is een array van integers met naam *teTonenTabel*. Bij het aanroepen van deze method dient er een int-array als eerste argument doorgegeven te worden.  
Zie bij (1) en (2): telkens wordt de int-array *getallen* doorgegeven aan de int array *teTonenTabel*. Dit zorgt er voor dat bij de eerste aanroep van de method (1) de tabel getallen, die dan nog ongesorteerd is, wordt getoond via de method. Bij de tweede aanroep van de method (2), wordt opnieuw de tabel getallen doorgegeven aan de method. De tabel getallen, die dan gesorteerd is, wordt doorgegeven aan *teTonenTabel* en wordt dus opnieuw getoond via de method.
- `String titel`: dit is het tweede argument van de method. Het is een string met de naam *titel*. Bij het aanroepen van deze method dient er als tweede argument een string doorgegeven te worden.  
Zie bij (1) en (2): telkens wordt er een string doorgegeven aan de string *titel*.

De volgorde van de argumenten van de method dient gerespecteerd te worden bij het aanroepen van deze method. Argumenten worden respectievelijk van links naar rechts doorgegeven.



De header van een method wordt de **signatuur** van de method genoemd. Ze bevat informatie die van belang is om die method aan te roepen.

### 6.5.2 Functies

Het verschil met een procedure is dat een functie steeds een waarde teruggeeft, terwijl een procedure dat niet doet.

Aan de hand van een voorbeeld wordt er uitgelegd hoe je in Java functies schrijft.

Neem volgend voorbeeld: De examenuitslagen voor 3 vakken m.n. wiskunde, boekhouden en informatica dienen ingelezen te worden. Elk vak staat op 10 punten. De student is geslaagd wanneer hij voor wiskunde minstens 6/10 behaalt en voor boekhouden en informatica samen minstens 12/20. Op het scherm wordt getoond of de student geslaagd is of niet en eventueel waarom niet.

Het cijfer voor de 3 vakken dient dus ingegeven en gevalideerd te worden. Het cijfer is een waarde die ligt tussen 0 en 10 (grenzen inbegrepen). Een functie is in dit geval zeer handig, vermits er een controle dient te gebeuren op elk cijfer dat moet liggen tussen 0 en 10. Het is geen goed idee om die controle 3 keer te schrijven, telkens voor de punten van de 3 vakken.

```

import java.util.Scanner;
public class Examens {
    public static void main(String[] args) {
        int wisk;
        int inf;
        int boekh;
        int somBI;

        wisk = geefPunten("wiskunde");           (1)
        inf  = geefPunten("informatica");        (2)
        boekh = geefPunten("boekhouden");        (3)

        somBI = boekh + inf;

        if (wisk >= 6 && somBI >= 12)
            System.out.println("Student is geslaagd");
        else {
            if (wisk < 6) {
                System.out.println("Student is niet geslaagd voor
                                   wiskunde");
            }
            if (somBI < 12) {
                System.out.println("Student is niet geslaagd voor
                                   boekhouden en informatica samen");
            }
        }
    } //einde main

    private static int geefPunten(String vak) {
        System.out.println("Geef de punten voor " + vak + " (op 10)");
        Scanner sc = new Scanner (System.in);
        int punten = sc.nextInt();

        while (punten < 0 || punten > 10) {
            System.out.println("Punten gaan op 10, dus gelieve een cijfer
                                tussen 0 en 10 in te geven aub");
            punten = sc.nextInt();
        }
        return punten;
    }
} //einde class examens

```

De uitvoer van de functie kan als volgt zijn:

```

Geef de punten voor wiskunde (op 10)
6
Geef de punten voor informatica (op 10)
8
Geef de punten voor boekhouden (op 10)
7
Student is geslaagd

```

Toelichting van de functie *geefPunten*:

```
private static int geefPunten(String vak)
```

- **private**: betekent letterlijk privaat: kan niet aangesproken worden vanaf de "buitenwereld". Deze method kan enkel aangesproken worden vanuit de class waar ze in

geschreven is, dus hier vanuit `Examens`.

- `static`: deze method dient static te zijn vermits ze wordt aangeroepen vanuit de main, die op zich ook static is. We komen later nog terug op het begrip static.
- `int`: is de returnwaarde van de method. De method zal een int teruggeven.
- `String vak`: dit is het enige argument van de method. Het is een string met naam *vak*. Bij het aanroepen van deze method dient er een string als argument doorgegeven te worden.

Zie bij (1), (2) en (3): steeds wordt de naam van het vak als een tekenreeks doorgegeven naar de functie, m.n. naar de variabele *vak*. In de functie wordt gevraagd voor een cijfer voor het betreffend vak. Het cijfer dat ingegeven wordt, wordt gevalideerd op een juiste waarde tussen 0 en 10. Wanneer de waarde juist is, geeft de functie het ingegeven cijfer terug aan het hoofdprogramma, de `main()`.

Bij (1) wordt het ingegeven cijfer teruggegeven aan de int variabele *wisk*, bij (2) aan *inf* en bij (3) aan *boekh*. Vermits de functie een int teruggeeft is het logisch dat deze waarde in de `main()` toegekend dient te worden aan een variabele van het type `int`.

Het voordeel van deze functie ligt hem in het feit dat de validatie op het ingegeven cijfer slechts één keer geschreven dient te worden i.p.v. 3 keer. Zo wordt herhaling van code vermeden.



De naamgevingconventies schrijven voor dat namen van procedures en functies beginnen met een kleine letter. In geval de naam uit meer dan een woord bestaat, zal de eerste letter van elk volgend woord beginnen met een hoofdletter.

## 7 OO, classes en objects

---

De kern van objectoriëntatie (OO) wordt gevormd door twee basisconcepten: classes (klassen) en objects (objecten). Beide vormen de basis van programmeren in objectgeoriënteerde talen.

### 7.1 Classes en objects

Objects worden gemaakt uit classes. Ze worden gemaakt in de code zelf, op basis van een sjabloon. Dit sjabloon wordt een class genoemd. Algemeen kan je stellen dat een class een ding uit de werkelijkheid beschrijft. Het beschrijft hiervan **eigenschappen** en **gedrag**.

Enkele voorbeelden van dingen uit de werkelijkheid:

- een Persoon uit de werkelijkheid heeft een naam, een geboortedatum, een adres... Dit zijn eigenschappen van een persoon.  
Verder kan een persoon verhuizen waardoor zijn adres wijzigt. Hij gaat werken. Hij doet eventueel aan sport. Dit bepaalt het gedrag van deze persoon of handelingen die de persoon doet.
- een Rekening uit de werkelijkheid heeft een rekeningnummer, een rekeninghouder, een saldo, een intrestpercentage, ... Ook dit zijn eigenschappen van de rekening.  
Verder kan je geld afhalen van de rekening, geld sorten op deze rekening, geld overschrijven van deze rekening naar een andere rekening. Dit beschrijft het gedrag van een rekening of handelingen die uitgevoerd kunnen worden.

Eigenschappen worden ook data members, properties, object variabelen of membervariabelen genoemd. Voortaan zal er in de cursus altijd gesproken worden over **membervariabelen**.

Gedrag beschrijft de handelingen die uitgevoerd kunnen worden op het object. Dit wordt beschreven in procedures of functies. Voortaan noemen we dit algemeen **methods**.

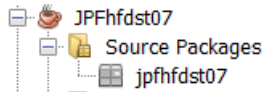
Classes zijn dus een uitbreiding van de taal. Ze zijn bruikbaar als een datatype. Objecten zijn opzichzelfstaande exemplaren van een class.

Enkele voorbeelden van objecten, ook wel instanties van een class genoemd:

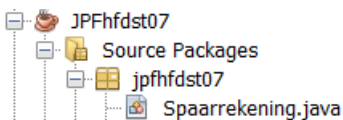
- Piet is een opzichzelfstaand exemplaar van de class Persoon: Hij heeft de naam Piet, zijn geboortedatum is 15-2-1978 en hij woont in de Stationsstraat te Genk. Hij geeft les en voetbalt.
- Mijn zichtrekening is een opzichzelfstaand exemplaar van de class Rekening. Deze heeft een rekeningnummer, een saldo, ikzelf ben rekeninghouder...
- De zichtrekening van mijn buurman is opnieuw een opzichzelfstaand exemplaar van de class Rekening. Deze rekening heeft een ander rekeningnummer, een andere rekeninghouder, een ander saldo, ...

## 7.2 Een class maken

Neem de class Spaarrekening, die de eigenschappen en het gedrag beschrijft van een spaarrekening. Maak hiervoor in NetBeans een nieuw project aan (bijv. JPFhfdst07). Het vinkje om tevens een Main Class te creëren, verwijder je. Dit zorgt ervoor dat onder Source Packages een <default package> staat. Dit is echter geen goede naam. Maak zelf een package aan door rechts te klikken op Source Packages > New > Java Package en kies voor een betere naam (bijv. jpfhfdst07) > Finish.



Klik vervolgens rechts op deze nieuwe package > New > Java Class, Class Name is Spaarrekening > Finish.



De naamgevingconventies schrijven voor dat namen van packages volledig in kleine letters geschreven worden.

Namen van classes daarentegen beginnen met een hoofdletter.

Er is nu een 'lege' class Spaarrekening:

```
package jpfhfdst07;
public class Spaarrekening {
}
```

De naam van de class (Spaarrekening) moet identiek zijn aan de naam van de source-file, zijnde *Spaarrekening.java*

Voeg onderstaande code toe aan de class Spaarrekening:

```
public class Spaarrekening {
    private String rekeningNummer;
    private double saldo;
    private double intrest;

    public Spaarrekening(String reknr, double intrest) {
        rekeningNummer = reknr;
        this.intrest = intrest;
    }

    public String getRekeningNummer() {
        return rekeningNummer;
    }

    public void setRekeningNummer(String reknr) {
        this.rekeningNummer = reknr;
    }

    public double getSaldo() {
```



```

        return saldo;
    }

    public void storten (double bedrag) {
        saldo += bedrag;
    }

    public void afhalen (double bedrag) {
        saldo -= bedrag;
    }

    public void overschrijven(Spaarrekening spaarRek, double bedrag) { (6)
        saldo -= bedrag;
        spaarRek.storten(bedrag);
    }
}

```

- (1) De class heeft 3 membervariabelen, nl. een *rekeningNummer* van het type String, een *saldo* en *intrest*, beide van het type double.

Deze membervariabelen worden automatisch op hun defaultwaardes geïnitieerd.

Numerieke variabelen krijgen de waarde 0 of 0.0, String variabelen krijgen de waarde "",

booleans krijgen de waarde *false* en references krijgen de waarde *null*.

Het zijn private membervariabelen wat betekent dat de buitenwereld niet zonder meer aan deze membervariabelen kan.

- (2) Dit is een constructor. Een constructor is een method met een identieke naam als de class (en begint dus ook met een hoofdletter). De method heeft geen returnwaarde! Meestal is een constructor public.

Objecten die gemaakt worden van een class, worden gemaakt m.b.v. een constructor.

Deze constructor heeft 2 argumenten waarvan de waarde toegekend wordt aan de membervariabelen. Omdat het argument *intrest* dezelfde naam heeft als de membervariabele *intrest*, wordt met het keyword **this** gerefereerd naar de membervariabele *intrest*:

```
this.intrest = intrest;
```

this verwijst altijd naar het huidige (of actuele) object, waardoor de variabele *intrest* na de toekenning het argument van de constructor is.

De membervariabele *saldo* wordt op de default waarde geïnitieerd, zijnde 0.0. De toekenning `saldo = 0;` hoeft niet expliciet opgenomen te worden in de constructor.

- (3) **getRekeningNummer()** is een public method die de waarde teruggeeft van de membervariabele *rekeningNummer*. *RekeningNummer* is een private membervariabele en kan niet benaderd worden door de buitenwereld. Dit kan wel met een public method die de waarde van de membervariabele teruggeeft. Een dergelijke method wordt ook een **getter** genoemd. Naar analogie is er ook een **getSaldo()**. Later meer over getters.
- (4) **setRekeningNummer()** is een method om de waarde van de membervariabele te wijzigen, of "te setten". Nadat het object gecreëerd is, kan het nodig zijn om de waarde van de membervariabele te wijzigen. Dit kan gebeuren via de set-methods of ook wel setters

genoemd. De waardes van de membervariabelen moeten geldige waardes zijn. Dat betekent dat een validatie van deze waardes geschreven kan worden in de setters. Later meer over setters.

Er is geen setter geschreven voor de membervariabele `saldo`. Dat betekent dat deze membervariabele enkel gewijzigd kan worden m.b.v. de methods `storten` en `afhalen`.

- (5) `storten` is een public method die een gedrag van een spaarrekening beschrijft. Het is een method van de class `Spaarrekening`. Het argument van deze method is een bedrag dat in de method bijgeteld wordt bij het saldo van de spaarrekening. Het is een void method en heeft dus geen returnwaarde. Het is de membervariabele `saldo` van de class die verhoogd wordt. Naar analogie is er ook de method `afhalen`.
- (6) De public method `overschrijven` beschrijft een ander gedrag van de `Spaarrekening`. Deze method heeft twee argumenten: het eerste argument is de spaarrekening waarnaar een bedrag overgeschreven zal worden en het tweede argument is het bedrag zelf. Een bedrag overschrijven naar een andere spaarrekening houdt o.a. in dat dit bedrag bijgeteld wordt bij het saldo van deze spaarrekening. Dit doe je door de method `storten` uit te voeren op die andere spaarrekening. Die andere spaarrekening is ook een object. Een method uitvoeren op dit object kan via het plaatsen van een punt en vervolgens de naam en argument(en) van de betreffende method (`spaarRek.storten(bedrag);`). En tot slot dient het saldo van de huidige spaarrekening verminderd te worden met het bedrag.

### 7.2.1 Opbouw van een class

De class bevat in de body een beschrijving van:

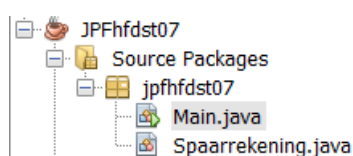
- de membervariabelen,
- een constructor,
- get-methods (ook getters genoemd),
- set-methods (ook setters genoemd),
- andere methods.

De volgorde van deze zaken in de body mag ook anders zijn. Doch deze volgorde is een volgorde die vaak terugkomt. Tracht consequent te zijn met de opbouw van de classes voor een project!

### 7.2.2 Main-programma dat werkt met objecten van de class

De class `Spaarrekening` is voorlopig klaar. Met een main-programma kunnen er vervolgens objecten gemaakt worden van deze class. Dit worden ook instances (of instanties) van de class genoemd.

Maak in de package een main-programma als volgt: klik rechts op de package `JPFhfdst07` > New > Java Main Class...> de Class Name is `Main` > Finish



Merk op dat bij een main class een groen driehoekje staat: dit is een uitvoerbare class. De class Spaarrekening is geen main class (heeft geen main() method) en kan dus niet uitgevoerd worden.

Vul de Main class aan:

```
package jpfhfdst07;
public class Main {
    public static void main(String[] args) {
        double standSpaarrekening;

        Spaarrekening spaar1 = new Spaarrekening("BE12 3456 7890 1234", 1.5); (1)
        Spaarrekening spaar2 = new Spaarrekening("BE98 7654 3210 9876", 1.5);

        spaar1.storten(100.0); (2)
        standSpaarrekening = spaar1.getSaldo(); (3)
        System.out.println("Saldo van spaarrekening 1: " + standSpaarrekening);

        System.out.println("Saldo van spaarrekening 2: " + spaar2.getSaldo() ); (4)

        spaar1.overschrijven(spaar2, 25.0); (5)

        System.out.println("Saldo van spaarrekening 1 " +
            spaar1.getRekeningNummer() + " is " + spaar1.getSaldo() ); (6)
        System.out.println("Saldo van spaarrekening 2 " +
            spaar2.getRekeningNummer() + " is " + spaar2.getSaldo() );

        spaar2.afhalen(5.0); (7)
        System.out.println("Saldo van spaarrekening 2 " +
            spaar2.getRekeningNummer() + " is " + spaar2.getSaldo() );
    }
}
```

Uitvoer van dit programma geeft het volgende :

```
Saldo van spaarrekening 1: 100.0
Saldo van spaarrekening 2: 0.0
Saldo van spaarrekening 1 BE12 3456 7890 1234 is 75.0
Saldo van spaarrekening 2 BE98 7654 3210 9876 is 25.0
Saldo van spaarrekening 2 BE98 7654 3210 9876 is 20.0
```

Toelichting van de code:

- (1) *spaar1* is een variabele van het type *Spaarrekening*. Het is niet zomaar een variabele, het is een reference variabele die verwijst naar een geheugenplaats waar de gegevens van een spaarrekening worden bewaard.  
`new` is het keyword waarmee een object wordt gemaakt. `new` wordt gevolgd door de naam van de constructor met de nodige argumenten zoals beschreven in de class *Spaarrekening*. Hier zijn dat 2 argumenten, nl. een String voor het rekeningnummer en een double voor de intrest.
- (2) De method **storten** wordt uitgevoerd op het object *spaar1*. Deze method zal het saldo van de spaarrekening verhogen met bedrag. Het bedrag is het argument van de method *storten*, hier 100.0 .

- (3) Met de method `getSaldo()` wordt het saldo van de spaarrekening opgevraagd. Dit saldo wordt hier toegekend aan de variabele `standSpaarrekening` om met de volgende regel code getoond te worden op het scherm.
- (4) Het is niet nodig om het saldo eerst te stockeren in een variabele en vervolgens te tonen. Hier gebeurt dit in een stap.  
Stel tevens vast dat de waarde van het saldo inderdaad geïnitieerd is op 0.0, de default waarde voor een double.
- (5) De method `overschrijven` wordt uitgevoerd op object `spaar1`. Deze method vraagt twee argumenten, nl. een spaarrekening en een bedrag. De spaarrekening is de rekening waarnaar het bedrag wordt overgeschreven.
- (6) De gegevens van de spaarrekeningen worden getoond. Je ziet het resultaat van de overschrijving die gebeurt is in de vorige opdracht.
- (7) De method `afhalen` wordt uitgevoerd op het object `spaar2`. Deze method zal het saldo van de spaarrekening verminderen met bedrag. Het bedrag is het argument van de method `afhalen`, hier 5.0.

### 7.3 Access modifiers

Java voorziet een aantal access modifiers om het niveau van toegang te bepalen voor classes, variabelen, methods en constructors. Gaandeweg is er reeds kennis gemaakt met `private` en `public`. `Private` werd tot nu toe voornamelijk gebruikt voor membervariabelen en `public` voor constructors en methods. Er zijn 4 niveaus:

- **public**: toegankelijk voor de buitenwereld,
- **private**: enkel toegankelijk binnen de class,
- **protected**: public binnen de package en alle subclasses (zie later), private daarbuiten.
- **default**: d.w.z. geen access modifier: public binnen de package, private daarbuiten. De default access modifier is van toepassing indien er geen expliciete keuze gemaakt wordt tussen public, private of protected. In dat geval krijgen de variabelen en de methods package-access. Dat betekent public binnen de package, private buiten de package. Dus voor alle classes die in dezelfde package zitten is dit hetzelfde als public, voor classes uit een andere package is dit hetzelfde als private.

### 7.4 Constructors

Een constructor is een speciale method die zorgt voor het creëren van een object van de class. Maar het gaat verder, de constructor is verantwoordelijk voor het creëren van een **geldig** object van de class. Een constructor is dus bedoeld om de membervariabelen van het object te initialiseren. Deze membervariabelen worden automatisch op hun defaultwaardes geïnitieerd:

- numerieke variabelen krijgen de waarde 0 of 0.0,
- String variabelen krijgen de waarde `""`,
- booleans krijgen de waarde `false` en
- references krijgen de waarde `null`.

tenzij de argumenten van de constructor een andere waarde bevatten.

Wat je verder nog weet. Een constructor:

- is een method met exact dezelfde naam als de class (en begint dus ook met een hoofdletter)
- is meestal public
- heeft geen returntype, dus ook niet void!

#### 7.4.1 Default constructor

Wanneer er geen constructor geschreven wordt, zal de compiler een default constructor voorzien. Een default constructor is een constructor zonder parameters en met een lege body. Bij het voorbeeld van de Spaarrekening is dit dan:

```
public Spaarrekening() {  
}
```

Membervariabelen van objecten die op deze manier worden gecreëerd, worden geïnitieerd op hun default waardes.

Een constructor zorgt ervoor dat er van de class een object kan worden gemaakt. Daarom is er minstens één constructor nodig. Je kan zelf een constructor schrijven, doch wanneer er geen constructor geschreven is, zal de compiler automatisch de default constructor voorzien.

#### 7.5 Membervariabelen versus lokale variabelen

Er is een verschil op vlak van initialisatie van variabelen.

Eerder is gesteld dat membervariabelen automatisch geïnitieerd worden op defaultwaardes indien er geen toekenning is voorzien. Dit geldt alleen voor de membervariabelen van een class.

Variabelen die gedeclareerd worden in methods, lokale variabelen dus, worden niet automatisch geïnitieerd.

#### 7.6 Method overloading

Er kan meer dan één constructor geschreven worden in een class. Denk terug aan de class Spaarrekening. Er is een constructor geschreven met 2 parameters. Je kan daarbij ook de defaultconstructor schrijven. Er zijn dan 2 methods geschreven met dezelfde naam, maar met een verschillend aantal parameters. Dit wordt method overloading genoemd.

Method overloading geldt niet alleen voor constructors, maar kan voor alle methods toegepast worden. Je kan dus meerdere varianten schrijven van een method. De varianten van de method hebben **dezelfde naam**, maar moeten **verschillen** van elkaar **in type en/of aantal parameters**. Afhankelijk van de parameters, bepaalt de JRE welke method uitgevoerd wordt.

Method overloading kent zeker zijn toepassing bij constructors.

Breid de class Spaarrekening uit met volgende constructor:

```
public Spaarrekening(String reknr, double intrest, double saldo) {  
    rekeningNummer = reknr;  
    this.intrest = intrest;  
    this.saldo = saldo;  
}
```

Er wordt een tweede constructor voorzien waarmee een spaarrekening aangemaakt kan worden die onmiddellijk een waarde toekent aan de membervariabele saldo (i.p.v. een initialisatie op 0.0).

Merk op dat de eerste twee statements in deze constructor gelijk zijn aan de vorige constructor. Daarom kan je gebruik maken van die vorige constructor bij deze tweede constructor:

```
public Spaarrekening(String reknr, double intrest, double saldo) {  
    this(reknr, intrest);  
    this.saldo = saldo;  
}
```

We gebruiken hier de eerste of vorige constructor door het sleutelwoord `this`. Het sleutelwoord `this` verwijst naar het actuele object. Uiteraard worden de argumenten op hun beurt doorgegeven aan de oproep van deze eerste constructor.

Het oproepen van een andere constructor binnen een constructor noemt “constructor chaining”. Dit moet tevens de eerste instructie zijn! In geval van extra geschreven code in de eerste constructor, zoals o.a. validatie, levert dit zeker zijn voordelen op en is dergelijk gebruik absoluut aangewezen.

## 7.7 Getters en setters

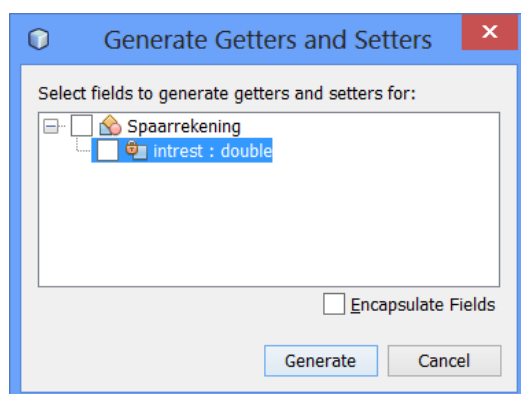
Een method om de waarde van een membervariabele op te vragen is een get-method of een getter. Een method om de waarde van een membervariabele te wijzigen is een set-method of een setter. De naamgevingconventie schrijft voor dat deze method begint met **get** of **set** en gevolgd wordt door de naam van de membervariabele beginnend met een hoofdletter.

Zo kennen we reeds `getSaldo()` en `setRekeningNummer()`.

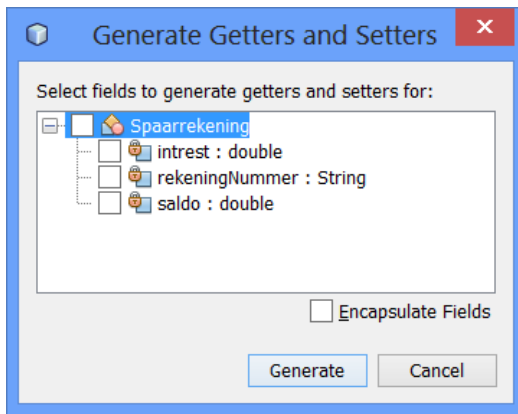
De IDE kan ons helpen om deze getters en/of setters te genereren. In NetBeans doe je dit als volgt:

De betreffende class is geopend in de editor en de membervariabelen zijn reeds gedeclareerd. Klik op menu *Source > Insert Code... >* en vervolgens heb je de keuze uit o.a. *Generate Getter...* (alleen getters), *Generate Setter...* (alleen setters), *Generate Getter and Setter...* (getters en setters).

Wanneer je dit probeert voor de class `Spaarrekening`, krijg je een venster om voor de membervariabele `intrest` een getter en setter te genereren. Dit is de enige membervariabele van deze class waarvoor noch een getter, noch een setter aanwezig is.



Wanneer er nog geen enkele getter en setter geschreven was, zou je de mogelijkheid gegeven worden om tegelijkertijd voor alle gewenste membervariabelen een getter en setter te genereren.



Selecteer de membervariabelen en klik op de knop Generate.

In de code wordt van elke membervariabele een getter en setter gegenereerd. Deze methods kunnen uiteraard gewijzigd worden.

## 7.8 Oefening



Zie takenbundel: maak volgende oefening die hoort bij hoofdstuk 7:  
- Een class Getal en een main-programma GetalMain.

## 7.9 Primitive types versus class types

Er is een **fundamenteel verschil** tussen het **declareren** van een **variabele** van een **primitive data type** en anderzijds een variabele van een **class type**. Een variabele van een class type (bijv. String) is een reference variabele. Deze variabele is een referentie die wijst naar een geheugenplaats voor een object van deze class.

Wanneer een variabele van een primitief type gereserveerd wordt, wordt er in het geheugen plaats voorzien voor dat primitief type (byte, int, long, float, double, char, boolean).

Wanneer een variabele van een class type (bijv. String) gereserveerd wordt, wordt er in het geheugen enkel plaats voorzien voor de referentievareabele die het geheugenadres zal bevatten van het object (bijv. String-object). Zodoende verwijst deze variabele naar de plaats in het geheugen waar het Stringobject is bewaard.

Geheugenruimte reserveren voor het object zelf doe je dan met het sleutelwoord *new*:

```
Spaarrekening spaar = new Spaarrekening("BE12 3456 7890 1234", 1.5);
```

De code hierboven betekent dan: "Reserveer voldoende geheugenruimte voor een object Spaarrekening en laat de reference (referentie) *spaar* wijzen naar deze gereserveerde geheugenruimte. Tevens is er op deze geheugenplaats een Spaarrekening object gemaakt met de opgegeven argumenten".

De class String is een uitzondering. Het sleutelwoord *new* hoeft niet gebruikt te worden voor het maken van een String-object. Variabelen van het type String zijn echter wel reference variabelen en

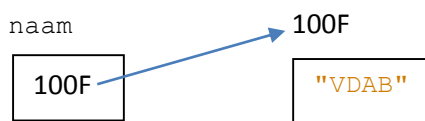
geen variabelen van het primitief type!

Visuele voorstelling:

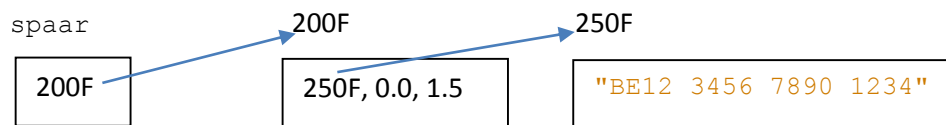
- `int x = 7;`  
In het geheugen bevat de variabele *x* de waarde 7.



- `String naam = "VDAB";`  
In het geheugen bevat de variabele *naam* het geheugenadres waar in het geheugen het String-object "VDAB" bewaard is.



- `Spaarrekening spaar = new Spaarrekening("BE12 3456 7890 1234", 1.5);`  
In het geheugen bevat de variabele *spaar* het geheugenadres waar in het geheugen het object van Spaarrekening bewaard is. Dit object bevat een String voor het rekeningnummer, en twee doubles voor een saldo en een intrest.

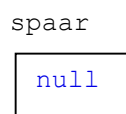


Merk op dat er voor het rekeningnummer, een variabele van het type String, ook een reference variabele is die verwijst naar het geheugenadres (250F) waar het rekeningnummer gestockeerd is.

### 7.9.1 Null reference

Wat gebeurt er nu wanneer enkel de referentie gedeclareerd wordt? Dus in geval van  
`Spaarrekening spaar;`

In het geheugen wordt een reference voorzien die wijst naar een Spaarrekening, maar zolang er geen Spaarrekeningobject is, is er geen geheugenadres dat verwijst naar het Spaarrekeningobject. Hierdoor is *spaar* een null reference.



Dergelijke references kunnen een *NullPointerException* veroorzaken wanneer er bijv. een method op uitgevoerd wordt.



## 7.10 Private methods

Eerder is reeds het verschil uitgelegd tussen procedures en functies. Algemeen worden dit methods genoemd. Methods hebben een access modifier. Methods kunnen public, private of protected zijn. Daarnaast bestaat ook de default access (zie eerder in de cursus bij de paragraaf van Access modifiers).

Voor de class Spaarrekening wordt een private method voorzien om te controleren of het bedrag van de method *storten*, *afhalen* en *overschrijven* groter is dan 0.0 . Een negatief bedrag storten, afhalen of overschrijven heeft echter geen zin.

Deze method kan er uitzien als volgt:

```
private boolean checkBedrag(double bedrag) {
    if (bedrag > 0.0)
        return true;
    else
        return false;
}
```

NetBeans zal in de marge aangeven dat het if-statement redundant is, d.w.z. dat je onmiddellijk het resultaat van de vergelijking kan returnen:

```
private boolean checkBedrag(double bedrag) {
    return bedrag > 0.0;
}
```

Voeg bovenstaande method toe aan de class Spaarrekening.

Deze method zal gebruikt worden om het bedrag te controleren. Pas daarom in de class Spaarrekening de volgende methods als volgt aan:

```
public void storten (double bedrag) {
    if (checkBedrag(bedrag)) {
        saldo += bedrag;
    }
}

public void afhalen (double bedrag) {
    if (checkBedrag(bedrag)) {
        saldo -= bedrag;
    }
}

public void overschrijven(Spaarrekening spaarRek, double bedrag) {
    if (checkBedrag(bedrag)) {
        saldo -= bedrag;
        spaarRek.storten(bedrag);
    }
}
```

Wijzig in de main bij één van de methods het bedrag naar een negatief bedrag en laat opnieuw uitvoeren. Je zal merken dat er dan niets gebeurt bij het uitvoeren van deze method.

## 7.11 Het sleutelwoord `static`

Met het sleutelwoord `static` worden in Java **classvariabelen** gedefinieerd. Classvariabelen zijn variabelen die opgeslagen worden in de class zelf en niet in een object. Men spreekt van variabelen met **classbereik**. Het zijn variabelen die dezelfde waarde hebben voor alle objecten geïntanceerd van de class.

Bij de class `Spaarrekening` kan je de membervariabele `intrest` static maken. Elke spaarrekening heeft dezelfde `intrest`, maar niet elke spaarrekening heeft hetzelfde rekeningnummer en hetzelfde saldo!

Wijzig dit in de class `Spaarrekening`:

```
private static double intrest;
```

Alle instanties die gemaakt worden van de class `Spaarrekening` delen in feite de variabele `intrest`. Het is een gemeenschappelijke variabele voor alle objecten.

In de constructor wijzig je best ook de regel `this.intrest = intrest;` in `Spaarrekening.intrest = intrest;`

Deze schrijfwijze benadrukt het feit dat het een classvariabele is.

Ook een method kan static zijn. Men spreekt dan van classbereik voor methods of classmethods. De method is niet afhankelijk van de toestand (de waarden van de membervariabelen) van een object. Net zoals classvariabelen bij de class horen en niet bij een instantie, horen classmethodes ook bij een class en niet bij een instantie van de class.

Dit kan toegepast worden bij de method voor het opvragen van de `intrest` bij de class `Spaarrekening`. Voeg onderstaande method toe aan de class `Spaarrekening`:

```
public static double getIntrest() {  
    return intrest;  
}
```

Hou er bij het toevoegen van de method rekening mee dat getters en setters meestal bij elkaar staan.

Het verschil tussen een method en een classmethod is dat classmethods aangeroepen kunnen worden zonder dat er een instantie bestaat van de class. Voor de method `getIntrest` geldt dus bijv.:

```
double intrestPercentage = Spaarrekening.getIntrest();
```

De method wordt niet opgeroepen op een object van de class, maar op de class zelf.

## 7.12 Oefeningen



Zie takenbundel: maak volgende oefeningen die horen bij hoofdstuk 7:

- Student
- Waarnemer
- Kaart

## 8 Inheritance

---

### 8.1 Introductie

Inheritance betekent in het Nederlands overerving.

In het algemeen spreek je bij het maken van classes over generalisatie en specialisatie. Bij generalisatie worden er algemene zaken gedefinieerd in een class. Bij specialisatie wordt er iets specifiek gemaakt op basis van een algemene class.

Met overerving kan je een class definiëren op basis van een andere class. Je kan dus een class maken die gebaseerd is op een bestaande class. Men noemt dit ook een class afleiden van een andere class. De nieuwe class wordt dan een uitbreiding van de bestaande class. Dit wil zeggen dat de nieuwe class de membervariabelen en methods erft van de bestaande class. Daarnaast kan de nieuwe class nog eigen membervariabelen en methods bevatten.

De bestaande class wordt **basisklasse**, **superclass**, **base class** of **parent class** genoemd.

De nieuwe class wordt **afgeleide klasse**, **subclass**, **derived class** of **child class** genoemd.

Je kan een class afleiden van een andere class door in de header van de afgeleide class het woord **extends** te vermelden gevolgd door de naam van de base class.

Een voorbeeld waar inheritance toegepast kan worden is bij de classes Zichtrekening en Spaarrekening:

- beide classes bevatten de membervariabelen rekeningNummer en saldo
- beide classes bevatten de methods: storten(), afhalen(), overschrijven() en geefSaldo().

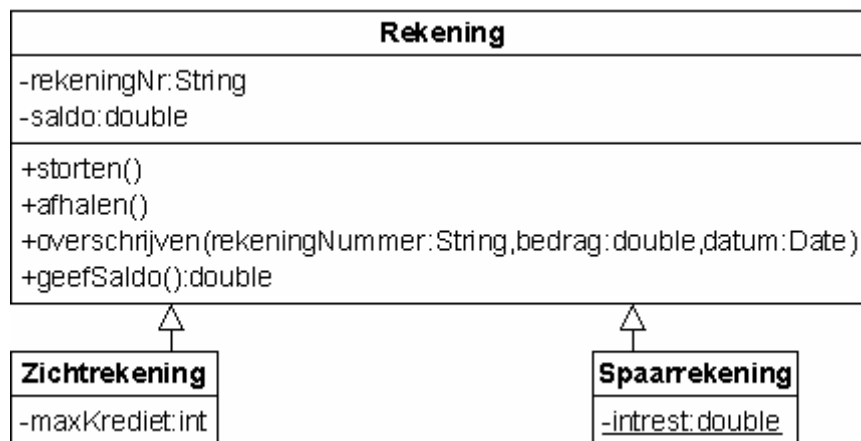
In plaats van deze gemeenschappelijke membervariabelen en methods twee keer uit te werken, zowel in de class Zichtrekening als in de class Spaarrekening, is het beter om deze gemeenschappelijke membervariabelen en methods slechts één keer uit te werken in een base class Rekening.

Daarna maak je de afgeleide class Zichtrekening en Spaarrekening, die erven van de class Rekening. Deze classes hebben dan automatisch de membervariabelen en methods van de base class Rekening ter beschikking, m.a.w. ze erven die. Constructors worden nooit geërfd!

Membervariabelen die enkel voor een spaarrekening gelden, beschrijf je in de class *Spaarrekening*. Dit is bijvoorbeeld de membervariabele *intrest*.

Membervariabelen die enkel voor een zichtrekening gelden, beschrijf je in de class *Zichtrekening*. Dit is bijvoorbeeld de membervariabele *maxKrediet*, zijnde de limiet van het bedrag dat je in het rood mag gaan op een zichtrekening.

Inheritance wordt in UML voorgesteld als een lijn tussen de afgeleide class en de base class, waarbij de lijn aan de kant van de base class een pijl bevat:



De class **Zichtrekening** **is een** **Rekening**, erft dus van de class **Rekening** en voegt nog een membervariabele toe die enkel geldt voor een zichtrekening en niet voor andere rekeningen, nl. *maxKrediet*.

De class **Spaarrekening** **is een** **Rekening**, erft dus van de class **Rekening** en voegt nog een membervariabele toe, die enkel geldt voor een spaarrekening, nl. *intrest*. Aangezien het intrestpercentage voor alle **Spaarrekening** objecten gelijk is, is dit een membervariabele met classbereik (static).

Als achteraf de base class **Rekening** uitgebreid wordt met extra membervariabelen en/of methods, worden deze automatisch geërfd in alle afgeleide classes, zijnde **Zichtrekening** en **Spaarrekening**.



Er bestaat altijd een **is een-relatie** tussen een afgeleide klasse en een basisklasse.

Van een afgeleide class kan op zich opnieuw een class afgeleid worden. Doch een afgeleide class kan slechts van één class afgeleid worden. Een dergelijke opbouw van classes en afgeleide classes wordt een **inheritance structuur**, **overervingstructuur** of **overervingshiërarchie** genoemd.

## 8.2 Het sleutelwoord `extends`

Je kan een class afleiden van een andere class door in de header van de afgeleide class het woord **`extends`** te vermelden gevolgd door de naam van de base class.

Het bovenstaande class-diagram toont:

- Base class is **Rekening**
- **Zichtrekening** extends **Rekening**
- **Spaarrekening** extends **Rekening**

Vertaald naar Java code bekom je:

```
public class Rekening {...}
public class Zichtrekening extends Rekening {...}
public class Spaarrekening extends Rekening {...}
```

Verderop in de cursus worden de classes volledig gemaakt.

### 8.3 Beveiligingsniveau `protected` in een class

In de paragraaf over access modifiers in een vorig hoofdstuk zijn de verschillende beveiligingsniveaus reeds aan bod gekomen. Het eerder beschreven beveiligingsniveau `protected` wordt even hernomen.

Wanneer je in een class een membervariabele of method `protected` declareert, dan is deze variabele of method alleen rechtstreeks toegankelijk binnen de class zelf en binnen de afgeleide classes van die class. Dit lijkt op de access modifier `public`. Voor de buitenwereld lijkt het echter `private`.

### 8.4 Constructors en inheritance

De afgeleide class erft membervariabelen en methods van de base class, maar constructors worden nooit geërfd!

Elke class heeft zijn eigen constructor(s):

- hetzij een default constructor (door de compiler gegenereerd)
- hetzij een eigen constructor of eigen geoverloade constructor(s). Dit kan een constructor met of zonder parameters zijn.

De door de compiler gegenereerde constructor is niet hetzelfde als een zelf geschreven constructor zonder parameters (een no-argument constructor).

Indien de base class een expliciete constructor heeft, dan moeten de afgeleide classes ook een expliciete constructor hebben. De default constructor of de constructor zonder parameters van de base class wordt impliciet opgeroepen als eerste statement in de constructor van de afgeleide class. Je kan dit ook zelf doen, met het keyword `super()`, maar alleen als eerste statement in de constructor:

- `super();` roept de default constructor (of die zonder parameters) van de base class op
- `super(parameters);` roept de geoverloade constructors van de base class op. Dit moet je altijd zelf doen.

### 8.5 Toepassen van inheritance bij Spaarrekening

Inheritance wordt toegepast bij de reeds gemaakte class `Spaarrekening`:

- Maak een nieuwe class `Rekening`:

```
package jpfhfdst08;
public class Rekening {
    private String rekeningNummer;
    private double saldo;

    public Rekening(String rekeningNummer) {
        this.rekeningNummer = rekeningNummer;
    }
}
```

```

    }

    public Rekening(String rekeningNummer, double saldo) {
        this.rekeningNummer = rekeningNummer;
        this.saldo = saldo;
    }

    public String getRekeningNummer() {
        return rekeningNummer;
    }

    public void setRekeningNummer(String reknr) {
        rekeningNummer = reknr;
    }

    public double getSaldo() {
        return saldo;
    }

    public void storten (double bedrag) {
        if (checkBedrag(bedrag)) {
            saldo += bedrag;
        }
    }

    public void afhalen (double bedrag) {
        if (checkBedrag(bedrag)) {
            saldo -= bedrag;
        }
    }

    public void overschrijven(Rekening rek, double bedrag) {
        if (checkBedrag(bedrag)) {
            saldo -= bedrag;
            rek.storten(bedrag);
        }
    }

    private boolean checkBedrag(double bedrag) {
        return bedrag > 0.0;
    }
}

```

- Wijzig nu de class Spaarrekening als volgt:

```

package jpfhfdst08;
public class Spaarrekening extends Rekening {
    private static double intrest;

    public Spaarrekening(String rekeningNummer, double intrest) {
        super(rekeningNummer);
        Spaarrekening.intrest = intrest;
    }

    public Spaarrekening(String rekeningNummer, double intrest, double saldo) {
        super(rekeningNummer, saldo);
        Spaarrekening.intrest = intrest;
    }

    public static double getIntrest() {

```

(1)

(2)

(3)

(4)

```

        return intrest;
    }
}

```

- (1) De header van de class wordt uitgebreid met **extends** en de naam van de class waarvan deze class afgeleid wordt. De class Spaarrekening erft alle membervariabelen en methods van de class Rekening.  
Ze erft geen constructors!
- (2) De class Spaarrekening heeft de membervariabele *intrest*. Aangezien de intrest voor alle objecten van deze class gelijk is, is de membervariabele static.
- (3) Indien de base class een expliciete constructor heeft (zoals hier het geval is), dan moeten de afgeleide classes ook een expliciete constructor hebben. In deze constructor moet als **eerste instructie** de constructor van de base class opgeroepen worden. Dit gebeurt via het keyword **super** (wat altijd een verwijzing is naar de base class van een class). Hier wordt dus de constructor van de class rekening opgeroepen met het argument rekeningnummer.
- (4) Er is een getter geschreven voor het opvragen van de waarde van intrest.

De class Spaarrekening is een afgeslankte versie geworden in vergelijking met de vorige versie van de class. Deze class erft alle membervariabelen en methods van de class Rekening. Uitzondering zijn de constructors, die worden nooit geërfd.

- Maak een nieuwe class Zichtrekening:

```

package jpfhfdst08;
public class Zichtrekening extends Rekening {
    private int maxKrediet;

    public Zichtrekening(String rekeningNummer, int bedrag) {
        super(rekeningNummer);
        maxKrediet = bedrag;
    }

    public Zichtrekening(String rekeningNummer, double saldo, int bedrag) {
        super(rekeningNummer, saldo);
        maxKrediet = bedrag;
    }

    public int getMaxKrediet() {
        return maxKrediet;
    }

    public void setMaxKrediet(int maxKrediet) {
        this.maxKrediet = maxKrediet;
    }
}

```

- (1) De header van de class wordt uitgebreid met *extends* en de naam van de class waarvan deze class afgeleid wordt. De class Zichtrekening erft alle membervariabelen en methods van de class Rekening. Ze erft geen constructors!
- (2) De class Zichtrekening heeft de membervariabele *maxKrediet*. Er is tevens een getter en setter voorzien voor maxKrediet.

- Schrijf de main() class BankBediende om te experimenteren met rekeningen:

```

package jpfhfdst08;
public class BankBediende {
    public static void main(String[] args) {
        System.out.println("Intrestpercentage van de spaarrekening: " +
            Spaarrekening.getIntrest());

        Spaarrekening spaar = new Spaarrekening("BE12 3456 7890 1234", 1.5);
        Zichtrekening zicht = new Zichtrekening("BE98 7654 3210 9876", 1000);

        System.out.println("Intrestpercentage van de spaarrekening: " +
            Spaarrekening.getIntrest());

        System.out.println("MaxKrediet van de zichtrekening is: " +
            zicht.getMaxKrediet() );

        spaar.storten(600.0);
        zicht.storten(200.0);

        spaar.afhalen(25.0);
        System.out.println("Saldo van de spaarrekening: " +
            spaar.getSaldo());

        zicht.afhalen(50.0);
        System.out.println("Saldo van de zichtrekening: " +
            zicht.getSaldo());

        zicht.afhalen(2000);
        System.out.println("Saldo van de zichtrekening (na poging " +
            "afhalen 2000): " + zicht.getSaldo());

        spaar.afhalen(600.0);
        System.out.println("Saldo van de spaarrekening (na poging " +
            "afhalen 600): " + spaar.getSaldo());
    }
}

```

Uitvoer van de main() class BankBediende:

```

Intrestpercentage van de spaarrekening: 0.0
Intrestpercentage van de spaarrekening: 1.5
MaxKrediet van de zichtrekening is: 1000
Saldo van de spaarrekening: 575.0
Saldo van de zichtrekening: 150.0
Saldo van de zichtrekening (na poging afhalen 2000): -1850.0
Saldo van de spaarrekening (na poging afhalen 600): -25.0

```

## 8.6 Method overriding

Het bovenstaande voorbeeld werkt zoals verwacht, maar is eigenlijk nog niet af. Er wordt nog geen rekening gehouden met het maxKrediet van een zichtrekening. En er gebeurt ook geen controle op het saldo bij het afhalen van een spaarrekening. Je zou kunnen stellen dat je niet onder 0 mag gaan bij een spaarrekening.

Dit kan je oplossen door een aparte versie van de method *afhalen* te voorzien voor de class Zichtrekening en Spaarrekening.

- Voeg volgende method toe aan de class ZichtRekening:



```

@Override
public void afhalen(double bedrag) {
    if (bedrag > 0.0) {
        double testSaldo = getSaldo() - bedrag + maxKrediet;
        if (testSaldo >= 0)
            super.afhalen(bedrag);
    }
}

```

- (1) De method *afhalen* wordt geërfd van de class Rekening, maar aangezien deze method voor een zichtrekening anders is, wordt deze method uitgeschreven in de class Zichtrekening zelf. Er wordt hier rekening gehouden met het maxKrediet, de limiet van het saldo van een zichtrekening die in het rood kan gaan.

De eigen versie van deze method wordt aangegeven met de annotation **@Override**. De method *afhalen* wordt overriden of anders gezegd, wordt overschreven, in deze class. Deze method heeft dezelfde signatuur als deze in de base class (zelfde naam, zelfde returntype en zelfde parameters).

- (2) Omdat in de base class de variabele saldo private is, kan deze niet rechtstreeks aangesproken worden in de afgeleide class, en dient het saldo opgevraagd te worden via de public method van de base class.

- (3) Met het keyword **super** wordt de method afhalen van de base class aangesproken en dus uitgevoerd. De method afhalen in de class Rekening is een public method die het saldo van een rekening vermindert en die hier dus gebruikt kan worden.

- Voeg volgende method toe aan de class Spaarrekening:

```

@Override
public void afhalen(double bedrag) {
    if (bedrag > 0.0) {
        double testSaldo = getSaldo() - bedrag;
        if (testSaldo >= 0) {
            super.afhalen(bedrag);
        }
    }
}

```

- (1) De method *afhalen* wordt geërfd van de class Rekening, maar aangezien deze method voor een spaarrekening anders is, wordt deze method uitgeschreven in de class Spaarrekening zelf. Er wordt hier rekening gehouden met het saldo en dit kan niet onder 0 gaan.

- Voeg volgende regels code toe aan de class BankBediende:

```

zicht.afhalen(1100);
System.out.println( "Saldo van de zichtrekening (na poging " +
                    "afhalen 1100): " + zicht.getSaldo());

zicht.setMaxKrediet(2000);
System.out.println( "MaxKrediet van de zichtrekening is: " +
                    zicht.getMaxKrediet() );

zicht.afhalen(900);

```

```
System.out.println( "Saldo van de zichtrekening (na poging " +
                    "afhalen 900): " + zicht.getSaldo());
```

Uitvoer van de main() class BankBediende:

```
Intrestpercentage van de spaarrekening: 0.0
Intrestpercentage van de spaarrekening: 1.5
MaxKrediet van de zichtrekening is: 1000
Saldo van de spaarrekening: 575.0
Saldo van de zichtrekening: 150.0
Saldo van de zichtrekening (na poging afhalen 2000): 150.0
Saldo van de spaarrekening (na poging afhalen 600): 575.0
Saldo van de zichtrekening (na poging afhalen 1100): -950.0
MaxKrediet van de zichtrekening is: 2000
Saldo van de zichtrekening (na poging afhalen 900): -1850.0
```

Bij de spaarrekening kan het saldo niet onder 0 gaan en bij de zichtrekening wordt er rekening gehouden met het maxKrediet.



Method overriding is dus een manier om een afgeleide class een eigen variant te geven van een bestaande method uit de base class.

Voorwaarden: identieke signatuur (access modifier, naam, returntype en parameterlijst zijn hetzelfde).

### 8.6.1 Annotations

De annotation `@Override` heb je intussen leren kennen. Een annotation plaats je steeds vooraan de header van een method. Dit is informatie voor de compiler om de nodige controles uit te voeren. De compiler zal nagaan of de base class een dergelijke method heeft die overriden kan worden in de huidige class. De method moet overeenkomen wat betreft de access modifier, de naam, het returntype en de argumenten.

Er bestaan meerdere annotations die je gaandeweg zal leren kennen.

## 8.7 Het sleutelwoord `final`

Het sleutelwoord `final` is eerder aan bod gekomen bij de beschrijving van `final` variabelen, ook constanten genoemd. Dit zijn variabelen die tijdens de uitvoering van het programma niet gewijzigd kunnen worden. Een voorbeeld:

```
final float PI = 3.141592F;
```

Een constante wordt onmiddellijk geïnitieerd met een waarde.

Het is gebruikelijk om de namen van constanten volledig in hoofdletters te plaatsen.

Het sleutelwoord `final` kan ook gebruikt worden bij methods en classes:

- Om te vermijden dat in een afgeleide class een method van de base class overriden wordt, gebruik je het keyword **final**. Wanneer een method `final` is, kan deze method nooit overriden worden in de afgeleide classes.

Een voorbeeld van de header voor een dergelijke method:

```
public final double berekenBedrag() {...}
```

- Het keyword `final` kan ook gebruikt worden bij een class. Wanneer een class `final` is, kan van deze class nooit een class afgeleid worden. Dit is dan het einde van de overervingstructuur. Er kan niet verder gespecialiseerd worden.

Om te vermijden dat je van de class `Rekening` een afgeleide class kan maken, doe je het volgende:

```
public final class Rekening {...}
```

Om te vermijden dat je van de class `Spaarrekening`, die afgeleid is van `Rekening`, niet opnieuw een class kan afleiden, doe je het volgende:

```
public final class Spaarrekening extends Rekening {...}
```

- Ook een variabele kan `final` zijn. Dat betekent dat deze variabele een waarde heeft die niet gewijzigd kan worden. Stel dat de intrest van een spaarrekening niet gewijzigd kan worden, dan zou je deze variabele als volgt kunnen declareren:

```
private final double intrest;
```

Wanneer deze variabele ook nog `static` is, krijg je het volgende:

```
private static final double intrest;
```

### 8.7.1 Final setters

Een constructor is de method die zorgt voor het creëren van een object van de class. Beter, de constructor is verantwoordelijk voor het creëren van een **geldig** object van de class. Een geldig object betekent een object met geldige waardes voor de membervariabelen. Bijv. geen negatieve leeftijd, geen 'lege' naam, enz. Dit dient gecontroleerd te worden. Dit kan in de constructor gebeuren. Maar aangezien er ook setters geschreven worden om de membervariabele aan te passen, is het een goed idee om deze controles in de setter te schrijven en de setters aan te roepen in de constructor. Zo vermijd je herhaling van code (dezelfde code in de constructor(s) + in de setters).

Pas het volgende aan in de class `rekening`. De setter van `rekeningNummer`:

```
public final void setRekeningNummer(String reknr) {
    if (reknr != null && !reknr.isEmpty() ) {
        rekeningNummer = reknr;
    }
}
```

(1)

- (1) Er wordt gecontroleerd of het argument *reknr* wel degelijk een waarde heeft, nl. niet *null* is en niet leeg is. De method *isEmpty()* geeft *true* terug wanneer de string een lege waarde ("" ) zou bevatten. Opgelet: de volgorde van deze vergelijking is belangrijk. De method *isEmpty()* geeft een *NullPointerException* wanneer ze uitgevoerd wordt op een *null*-reference. Het rekeningnummer kan nog verder gecontroleerd worden op een geldig IBAN nummer. Hiervoor zijn nog andere methods van de *String* class nodig. Dit wordt later toegevoegd wanneer het hoofdstuk van *Strings* behandeld wordt.

Pas ook de constructors aan:

```
public Rekening(String rekeningNummer) {
    setRekeningNummer(rekeningNummer);
}
```

(1)

```

    }

    public Rekening(String rekeningNummer, double saldo) {
        setRekeningNummer(rekeningNummer);
        if (saldo >= 0) {
            this.saldo = saldo;
        }
    }

```

- (1) De setter van rekeningnummer wordt aangeroepen. Voordeel is dus dat de controle van het rekeningnummer ook gebeurt in de constructor.
- (2) Aangezien er voor *saldo* geen setter geschreven is, wordt de controle geschreven in de constructor. Zo kan het saldo van de rekening bij de creatie van het object niet negatief zijn. De membervariabele saldo wordt nl. eerst geïnitieerd op de defaultwaarde. Vervolgens wordt in de constructor het argument saldo gecontroleerd. Enkel in geval van een positief saldo of 0 gebeurt de toekenning. Een negatief saldo wordt niet toegekend.

Indien je het keyword `final` bij `setRekeningNummer` weghaalt, verschijnt er een opmerking (geel lampje) bij de constructor. “Overridable method call in constructor” is de melding. Je wordt gewaarschuwd dat je een method oproept die overriden kan zijn, terwijl je nog geen geïnitieerde instantie hebt. Daarom is het gebruikelijk om setters die je ook gebruikt in de constructor `final` te maken.

Ter vollediging pas je ook de constructors van `Spaarrekening` aan met de controle van het argument `intrest`. Daarvoor wordt ook de method `setIntrest` geschreven:

```

public Spaarrekening(String rekeningNummer, double intrest) {
    super(rekeningNummer);
    setIntrest(intrest);
}

public Spaarrekening(String rekeningNummer, double intrest, double saldo) {
    super(rekeningNummer, saldo);
    setIntrest(intrest);
}

public final void setIntrest(double intrest) {
    if (intrest > 0.0) {
        Spaarrekening.intrest = intrest;
    }
}

```

En ook deze van `Zichtrekening` met de controle van het argument `bedrag`:

```

public Zichtrekening(String rekeningNummer, int bedrag) {
    super(rekeningNummer);
    setMaxKrediet(bedrag);
}

public Zichtrekening(String rekeningNummer, double saldo, int bedrag) {
    super(rekeningNummer, saldo);
    setMaxKrediet(bedrag);
}

public final void setMaxKrediet(int maxKrediet) {

```

```
    if (maxKrediet > 0) {  
        this.maxKrediet = maxKrediet;  
    }  
}
```

## 8.8 De class Object

Elke class is impliciet afgeleid van de class Object. De class Object is daardoor de “moeder” van alle classes. Men spreekt ook van de super class Object. Dit betekent dat wanneer je in de header geen keyword *extends* gebruikt, je de class dus niet afleidt van een andere class, de compiler automatisch “*extends Object*” toevoegt. Via deze overervingstructuur zijn er standaard enkele methods ter beschikking. Echter vaak zijn deze methods dan onbruikbaar. Het is aan te raden om deze methods te overriden in je class. Zo bekom je wel bruikbare methods. Voorbeelden hiervan zijn *toString()* en *equals()*. Verder in de cursus komen dergelijke andere methods nog aan bod.

### 8.8.1 toString()

De method *toString()* geeft een stringrepresentatie van het object. De stringrepresentatie betreft meestal een string met de waardes van alle membervariabelen gescheiden door een komma, spatie of een ander symbool. Meestal komt hier geen of weinig opmaak aan te pas. Het returntype van deze method is dus een *String*. Deze method kent zeker zijn waarde bij console-applicaties. Later kent deze method zijn waarde bij het debuggen.

De *toString()*-method wordt automatisch aangeroepen telkens wanneer een object in een string-context gebruikt wordt, bijv. `System.out.println(...)`.

Neem het main-programma *BankBediende*. Voeg onderaan volgende regels als laatste regels toe:

```
System.out.println(spaar);  
System.out.println(zicht);
```

De uitvoer van het programma voor deze regels is:

```
jpfhfdst08.Spaarrekening@531ae81d  
jpfhfdst08.Zichtrekening@b7cf28b
```

De uitvoer bestaat uit de packagenaam, gevolgd door een punt met daarna de naam van de class, vervolgens een @ en daarna een intern id van het object.

Wat gebeurt er precies? Bij `System.out.println(spaar);` is *spaar* een object en de *println()* method van *out* roept eerst `String.valueOf(spaar)` aan om de stringrepresentatie van het *spaar* object te bekomen. Dat bekom je met de *toString()* method.

De method *toString()* wordt geërfd van class Object, maar is in feite onbruikbaar in het programma *BankBediende*. De method *toString()* is nog niet overriden in de class *Spaarrekening*. Om een bruikbare versie van de *toString()* te bekomen, dien je dus in de class *Rekening*, *Spaarrekening* en *Zichtrekening* de *toString()* te overriden.



Het is aanbevolen om in elke class de method **toString()** te overriden. Dit helpt in ieder geval bij het debuggen van je applicatie.

- Voeg in de class Rekening volgende *toString()* method toe.

```
@Override
public String toString() {
    return rekeningNummer + ", " + saldo;
}
```

- Voeg in de class Spaarrekening volgende *toString()* method toe.

```
@Override
public String toString() {
    return super.toString() + ", " + intrest;
}
```

(1)

- (1) `super.toString()` roept de *toString()* method aan van de base class Rekening. Dit levert een string op en deze string wordt aangevuld met de membervariabele van de class Spaarrekening, zijnde intrest.

- Voeg in de class Zichtrekening volgende *toString()* method toe.

```
@Override
public String toString() {
    return super.toString() + ", " + maxKrediet;
}
```

De uitvoer van het programma BankBediende voor de twee laatst toegevoegde regels zal nu als volgt zijn:

```
BE12 3456 7890 1234, 575.0, 1.5
BE98 7654 3210 9876, -1850.0, 2000
```

De eerste bevat de waardes van het spaarrekeningobject, zijnde rekeningnummer, saldo en intrest. Ze zijn gescheiden door een komma + spatie.

De tweede regel bevat de waardes van het zichtrekeningobject, zijnde rekeningnummer, saldo en maxKrediet. Ze zijn eveneens gescheiden door een komma + spatie.



Een vuistregel voor de **toString()** method is dat de method een string teruggeeft die enkel de waardes bevat van de membervariabelen, gescheiden door een symbool, zonder opmaak. Het betreft een **stringrepresentatie van het object**.

Indien je meer opmaak wenst, is het aanbevolen om hiervoor een andere method te schrijven, bijv. een method `toon()` of `display()`. In een dergelijke method bepaal je volledig zelf welke informatie er getoond wordt met al dan niet veel of weinig opmaak. Deze methods zullen echter enkel van toepassing kunnen zijn bij console-applicaties. Je zal gaandeweg applicaties leren maken waarbij de uitvoer, de GUI of het uitzicht van een webapplicatie elders gecodeerd wordt. Hoe de gegevens worden weergegeven is uiteindelijk niet de verantwoordelijk van deze class. Maar dat is voor later!

### 8.8.2 equals()

Een andere method die je erft van Object is `equals()`. De method `equals()` is een method die bepaalt of twee objecten aan elkaar gelijk zijn. Het returntype van deze method is een boolean. Ook hier geeft de geërfde versie van de method geen betrouwbare informatie. Het is daarom aan te bevelen om deze method in je class te overriden. In deze method leg je vast wanneer twee objecten aan elkaar gelijk zijn. Denk even terug aan de werkelijkheid:

- Twee personen (persoon-objecten) zijn aan elkaar gelijk wanneer het rijksregisternummer aan elkaar gelijk is. Dan weet je met zekerheid dat het om dezelfde persoon gaat.
- Twee werknemers (werknemer-objecten) zijn aan elkaar gelijk wanneer het personeelsnummer aan elkaar gelijk is.
- Twee wagens (wagen-objecten) zijn aan elkaar gelijk wanneer de nummerplaat aan elkaar gelijk is.
- enz.

Je bepaalt hier **inhoudelijke gelijkheid**!

De `equals` method, die je erft van class Object, heeft als argument een variabele van type Object. Het is de bedoeling om na te gaan of het huidige object gelijk is aan het object van het argument van de method.

Verder in dit hoofdstuk wordt deze method uitgewerkt in de class Rekening.



Het is aanbevolen om in elke class de method **equals** te overriden.

## 8.9 Abstracte classes en methods

### 8.9.1 Abstracte classes

Neem het voorbeeld van de hiërarchie van Rekening. Welke soort rekeningen bestaan er? Je kan in de werkelijkheid geen rekening openen bij de bank, maar wel een zichtrekening of een spaarrekening. Een rekening zonder meer kan je niet openen. Om te vermijden dat er objecten van type Rekening gemaakt worden, maak je de class abstract.

```
public abstract class Rekening {  
    ...  
}
```

Zo kunnen geen objecten van Rekening geïntanceerd worden, maar wel objecten van Spaarrekening en Zichtrekening.

Een abstracte class komt vaker voor bij een overervingstructuur. Deze class wordt dan gebruikt als basis voor de overervingstructuur. Dit biedt de mogelijkheid om algemene kenmerken te groeperen in één class. Tevens voorkom je dat er objecten gemaakt worden die eigenlijk geen zin hebben. Een rekening op zich bestaat niet, het gaat om een zichtrekening of een spaarrekening, niet om een rekening.

Er kunnen wel referenties van een abstracte class gemaakt worden om later polymorf te gebruiken. Zie verder bij polymorfisme.

### 8.9.2 Abstracte methods

Een abstracte method is een method waar geen zinnige implementatie aan kan gegeven worden in de huidige class, maar wel in de afgeleide classes.

Voor de situatie van de rekeningen is het zo dat de intrest die je krijgt verschilt afhankelijk van het type van de rekening. Een zichtrekening zal minder intrest opleveren dan een spaarrekening. Een method om de intrest te berekenen heeft dus weinig zin in de class Rekening, maar heeft wel zin in de class Zichtrekening en Spaarrekening. Om er zeker van te zijn dat de classes Zichtrekening en Spaarrekening een method bevatten om de intrest te berekenen, voorzie je hiervoor in de class Rekening een abstracte method.

- Een abstracte method moet in de afgeleide classes overriden worden (de compiler bewaakt dit),
- Een abstracte method mag geen body hebben,
- Een abstracte method maakt de class tot abstracte class,
- Een abstracte method geeft faciliteiten bij polymorfisme (zie verder in de cursus).

Om dit toe te passen in de hiërarchie van Rekening doe je het volgende:

- Voeg in de class Rekening volgende abstracte method toe:  
`public abstract double berekenIntrest();`
  - Je dient dan ook de class Rekening abstract te maken, indien dit nog niet gebeurd is:  
`public abstract class Rekening {  
 ...  
}`
  - Voorzie de method `berekenIntrest()` in de class Spaarrekening:  
`@Override  
public double berekenIntrest() {  
 return getSaldo() * intrest / 100;  
}`
  - Voeg aan de class Zichtrekening een constante (static variabele) toe met het intrestpercentage voor een zichtrekening:  
`private static double INTREST_ZICHTREKENING = 0.25;`
  - Voorzie de method `berekenIntrest()` in de class Zichtrekening:  
`@Override  
public double berekenIntrest() {  
 if (getSaldo() > 0) (1)  
 return getSaldo() * INTREST_ZICHTREKENING / 100;  
 else  
 return 0.0;  
}`
- (1) Er wordt enkel een intrest berekend voor een positief saldo. Voor het gemak wordt er geen rekening gehouden met het saldo dat gedurende bepaalde dagen op de rekening staat. Daarvoor is een historiek van het saldo per dag noodzakelijk en dat zou het voorbeeld hier veel te ver leiden.  
Hetzelfde geldt voor een negatief saldo. Dat levert je geen intrest op, maar kosten. Ook



deze kosten worden niet berekend.

- Breid het main-programma BankBediende uit om het resultaat van de abstracte method te kunnen zien:

```
System.out.println("Intrest op de spaarrekening: " + spaar.berekenIntrest());
System.out.println("Intrest op de zichtrekening: " + zicht.berekenIntrest());
```

Deze lijnen code leveren volgende uitvoer:

```
Intrest op de spaarrekening: 8.625
Intrest op de zichtrekening: 0.0
```

## 8.10 Polymorfisme

Van een abstracte class kan geen object geïnstantieerd worden, maar er kunnen wel referenties gemaakt worden. Deze referenties kunnen dan polymorf gebruikt worden. De term polymorf slaat op het feit dat een variabele objecten van verschillende types kan bevatten, nl. het gedeclareerde type of één van de subtypes van het gedeclareerde type. Variabelen in Java die objecttypes bevatten, worden daarom polymorfe variabelen genoemd.

Terug naar het voorbeeld: voor de bank bestaan er geen rekeningen, maar wel zichtrekeningen en spaarrekeningen. De class Rekening is om deze reden abstract. Er kan bijv. wel een verzameling rekeningobjecten gemaakt worden die feitelijke zichtrekening-objecten en spaarrekening-objecten bevat.

Voeg volgende code onderaan toe aan het main-programma BankBediende:

```
//toepassing polymorfisme
Rekening[] rekeningen = new Rekening[4]; (1)
rekeningen[0] = new Spaarrekening("BE11 2233 4455 6677", 1.5); (2)
rekeningen[1] = new Spaarrekening("BE99 8877 6655 4433", 1.5);
rekeningen[2] = new Zichtrekening("BE19 2837 4655 6473", 2000);
rekeningen[3] = new Zichtrekening("BE91 8273 6455 4637", 1500);

rekeningen[0].storten(500.0); (3)
rekeningen[1].storten(550.0);
rekeningen[1].afhalen(120.0); (4)
rekeningen[2].storten(200.0); (3)
rekeningen[2].afhalen(20.0); (4)
rekeningen[3].storten(300.0);

int i = 0;
while (i < rekeningen.length && rekeningen[i] != null) {
    System.out.println("Saldo van de rekening: " + rekeningen[i].getSaldo()); (5)
    i++;
}
```

- (1) Er wordt een array gemaakt van 4 Rekening-referenties. Alhoewel de class Rekening abstract is en er dus geen objecten kunnen van geïnstantieerd worden, kan je wel referenties maken van deze abstracte classe.

- (2) De eerste twee elementen (referenties) van de array wijzen naar een Spaarrekening. De twee volgende elementen wijzen naar een Zichtrekening. Dit kan aangezien een Spaarrekening en

een Zichtrekening ook Rekeningen zijn. Men spreekt in dit verband over een 'is een' relatie van de inheritance-structuur.

- (3) Hier wordt de method *storten()* opgeroepen. De method *storten()* is niet expliciet geschreven in de classes Spaarrekening en Zichtrekening maar wordt dus geërfd van de base class Rekening. Deze method is dus voor beide objecten gelijk.

Het proces om een Spaarrekening-object (een meer gedetailleerde versie van een algemene rekening) te beschouwen als een Rekening-object, noemt men **upcasting** (Rekening staat hoger in de overervingsstructuur). Dit gebeurt hier impliciet. Een Spaarrekening is ten slotte een Rekening.

- (4) Hier wordt de method *afhalen()* opgeroepen en er wordt, per object, de specifieke versie van de method uitgevoerd. Voor een object dat een Spaarrekening-object is, wordt bij het uitvoeren van *afhalen()* de method *afhalen()* van de class Spaarrekening uitgevoerd. Bij een object dat een Zichtrekening-object is, wordt de method *afhalen()* uit de class Zichtrekening uitgevoerd. M.a.w. afhankelijk van het object, zal de method *afhalen()* van Spaarrekening of Zichtrekening uitgevoerd worden.

Het is de JVM die er voor zorgt dat de juiste versie van de method wordt uitgevoerd.

De algemene Rekening-referenties gedragen zich dus soms als een Spaarrekening, soms als een Zichtrekening. Ze gedragen zich polymorf. Dit wordt polymorfisme genoemd.

- (5) Zoals bij (3) beschreven, wordt ook hier de versie van de method *getSaldo()* van de base class uitgevoerd.

Dit stukje code geeft volgende uitvoer:

```
Saldo van de rekening: 500.0
Saldo van de rekening: 430.0
Saldo van de rekening: 180.0
Saldo van de rekening: 300.0
```



Met polymorfisme wordt de functionaliteit van een object behouden, ook al spreekt men het aan vanuit een referentie naar de base class.

Voorwaarde: Polymorfisme werkt alleen maar voor die methods die in alle afgeleide classes aanwezig zijn **EN** in de base class. In de base class mag die method eventueel abstract zijn.

### 8.10.1 De operator **instanceof**

Om de Rekening referentie, die verwijst naar een Spaarrekening, ook te kunnen gebruiken als een Spaarrekening referentie, moet de referentie getypecast worden naar zijn eigen type. Dit is **downcasting** (Spaarrekening staat lager in de overervingsstructuur). Dit is het omgekeerde van

upcasting. Dit gebeurt nooit impliciet. Het kan dus enkel expliciet gebeuren. Een Rekening is niet per definitie een Spaarrekening, het kan ook een Zichtrekening zijn. Bijv.

```
Spaarrekening spaarrek = (Spaarrekening) rekeningen[0];
```

Om te vermijden dat er een fout optreedt bij deze casting, controleer je best steeds of de referentie inderdaad van het type is waarnaar je wenst te typecasten. De fout die kan optreden is een *ClassCastException*.

Om de referentie op het type te controleren, gebruik je de *instanceof* operator:

```
Spaarrekening spaarrek;
if (rekeningen[0] instanceof Spaarrekening) {           (1)
    spaarrek = (Spaarrekening) rekeningen[0];           (2)
}
```

- (1) De operator *instanceof* gaat na of de referentie *rekeningen[0]* van het type *Spaarrekening* is.
- (2) Indien (1) *true* oplevert, kan je de referentie *rekeningen[0]* casten naar een *Spaarrekening* en de referentie ervan toekennen aan een *Spaarrekening*-referentie.

Bij *instanceof* moeten zowel het type in het rechterlid als het type in het linkerlid referentietypes zijn.

Indien je *instanceof* toepast op een *null* referentie, zal dit *false* opleveren.

Bijv.

```
if (rekeningen[4] instanceof Spaarrekening) {
    ...
}
```

Dit levert *false* op aangezien *rekeningen[4]* enkel gedeclareerd is en dus nog een *null*-referentie is, *rekeningen[4]* is nog niet geïnitieerd.

#### 8.10.1.1 equals() uitgewerkt in de class Rekening

Eerder is reeds gezegd dat de *equals()* method bepaalt of twee objecten inhoudelijk aan elkaar gelijk zijn. Voeg in de class *Rekening* volgende *equals()* method toe. Twee *Rekening* objecten zijn aan elkaar gelijk wanneer het rekeningnummer aan elkaar gelijk is.

```
@Override
public boolean equals(Object o) {
    if (o == null) {
        return false;
    }
    if (!(o instanceof Rekening)) {           (1)
        return false;
    }
    Rekening rek = (Rekening) o;             (2)
    return rekeningNummer.equals(rek.rekeningNummer); (3)
}
```

- (1) Er wordt nagegaan of het argument van de method type *Rekening* heeft. Indien niet en in geval van een *null*-argument, kunnen de objecten niet aan elkaar gelijk zijn en wordt er *false* gereturned.

- (2) Anders gebeurt er een typecast van Object o naar een Rekening object.
- (3) Vervolgens gebeurt de vergelijking op het rekeningnummer. Het rekeningnummer is een String object en daarom wordt de equals() van String gebruikt om te controleren of beide rekeningnummers aan elkaar gelijk zijn.

In de marge geeft NetBeans aan dat de method hashCode() ontbreekt. Dit is ook één van die methods die je automatisch overerft van Object. Negeer voorlopig deze melding. De method hashCode() hangt nauw samen met de equals() maar wordt verder in de cursus besproken bij het hoofdstuk van collections.

## 8.11 De for-each-lus

De while-lus van het main-programma BankBediende kan ook geschreven worden als een for-lus:

```
//for-lus ipv while-lus
for (int i=0; i<rekeningen.length;i++) {
    if (rekeningen[i] != null) {
        System.out.println("Saldo van de rekening: " + rekeningen[i].getSaldo());
    }
}
```

Het resultaat van deze lus is uiteraard hetzelfde.

Eerder is reeds aangehaald dat er een variant bestaat op de for-lus, met name de for-each-lus.

Bovenstaande for-lus kan herschreven worden naar een for-each-lus:

```
//for-each-lus
for (Rekening rekening: rekeningen) {
    if (rekening != null) {
        System.out.println("Saldo van de rekening: " + rekening.getSaldo());
    }
}
```

- (1) De lus itereert over de array `rekeningen`, een verzameling van referenties. De array wordt helemaal doorlopen van het begin tot het einde. Elk element wordt automatisch beschouwd als een Rekening referentie.  
Je kan de regel bijna vertalen als volgt: “Voor elke Rekening `rekening` uit de array `rekeningen` gebeurt er wat tussen de accolades `{ }` staat”.
- (2) Bij iedere cyclus van de lus wordt één element (uit de array) genomen. Dit element is een Rekening referentie. Er wordt gecontroleerd of het geen null-reference is.
- (3) Bij een geldige reference, wordt van de rekening het saldo getoond.

De for-each-lus is een veel gebruikte lus, vooral bij het itereren over arrays en collecties (zie later in de cursus). In het algemeen bij het itereren over verzamelingen.

De for-each-lus wordt ook een enhanced for loop genoemd. Een for-each-lus is korter, leesbaarder en foutvrijer.

## 8.12 Oefeningen



Zie takenbundel: maak de oefening die hoort bij hoofdstuk 8:  
- Voertuigen

## 9 Strings

---

### 9.1 Introductie

Een String is **geen** primitive datatype. Het is een class type. Strings zijn geïmplementeerd als objecten afgeleid van de class String, met membervariabelen en methods.

Nochtans het creëren van een stringobject met het keyword *new*, bijv.

```
String tekst = new String ("abc");
```

kan korter geschreven worden. De class String vormt hierop een uitzondering. Dit mag als volgt:

```
String tekst = "abc";
```

Tot nu toe zijn strings frequent gebruikt in de toString() method. Daar worden meerdere membervariabelen met een scheidingsteken (meestal een komma) aan elkaar gekoppeld door de + operator. Dit wordt stringconcatenatie genoemd.

```
String tekst = "abc";  
tekst += "def";
```

Wat gebeurt er in feite?

- Na de eerste regel bestaat er een referencevariabele *tekst* die verwijst naar een stringobject met de waarde "abc".
- Bij uitvoering van de tweede regel wordt er een nieuw stringobject gemaakt, dat voldoende groot is om alles te bevatten. De waarde wordt nu "abcdef". De bestaande referencevariabele *tekst* krijgt nu een nieuwe inhoud (een nieuw adres): deze referencevariabele verwijst nu naar het nieuwe grotere object. Het oude object (de oorspronkelijke string met inhoud "abc") wordt verwijderd omdat er geen references zijn naar dit object.

Dit betekent dat er bij stringconcatenaties steeds nieuwe objecten in het geheugen ontstaan. En dat heeft te maken met het feit dat een string **immutable** (onveranderbaar) is. Dit betekent dus dat als een String een bepaalde waarde heeft, deze waarde niet gewijzigd kan worden. Er wordt steeds een nieuw object gemaakt voor de nieuwe waarde. Gelukkig wordt de stringconcatenatie in de toString() door de compiler geoptimaliseerd, waardoor de opbouw van de string met de + operator behouden kan blijven.

Er bestaan nog andere classes zoals **StringBuilder** en **StringBuffer** die beter geschikt zijn voor manipulaties van strings (zie verder in dit hoofdstuk).

### 9.2 Speciale tekens in een string

Stringwaarden worden tussen dubbele aanhalingstekens geplaatst. Indien een string zelf aanhalingstekens dient te bevatten, maak je gebruik van een escape character. Voorbeeld:

```
String tekst = "En hier zitten de \"aanhalingstekens\" dan.";
```

De \ is het escape-teken. Dit wil zeggen dat de normale betekenis van het teken volgend op de \ ge-escaped wordt en vervangen wordt door een alternatieve betekenis.

De belangrijkste speciale tekens en hun escaped value zijn:

Escape character	Betekenis
\t	tab
\n	new line
\r	return
\\	backslash
\'	enkele quote
\"	dubbele quote

## 9.3 Bewerkingen met strings

### 9.3.1 Het vergelijken van strings

Beschouwen we volgende code:

```
String tekst1 = "abc";  
String tekst2 = "abc";
```

In bovenstaande code definiëren we twee strings met de waarde “abc”.

String variabele `tekst1` is een referentievariabele die na de bewerking verwijst naar een stringobject met inhoud “abc”. ‘Verwijzen naar’ betekent dat in `tekst1` het *adres* zit van de geheugenplaats waar de waarde van het stringobject (“abc”) is bewaard.

Voor String variabele `tekst2` geldt hetzelfde. `tekst2` heeft als inhoud een geheugenadres dat verwijst naar een (ander) stringobject waarvan de inhoud ook “abc” is.

Om beide strings te vergelijken op gelijkheid, wordt er geen gebruik gemaakt van de `==` operator. Deze operator gaat na of de referencevariabelen aan elkaar gelijk zijn en vergelijkt dus 2 adressen. Beide adressen zijn verschillend. Uiteraard is het niet de bedoeling om adressen te vergelijken, maar om de inhoud te vergelijken.

Voor stringobjecten zijn er in de String class een aantal methods die een goede vergelijking wel mogelijk maken.

Een vergelijking op gelijkheid:

- **`equals()`**: een hoofdlettergevoelige vergelijking; deze method geeft een boolean terug. Toegepast op bovenstaande 2 strings: `tekst1.equals(tekst2)` levert **true** op.
- **`equalsIgnoreCase()`**: zoals voorgaande, maar dan niet hoofdlettergevoelig.

Een vergelijking volgens alfabet:

- **compareTo():** veronderstel `string1.compareTo(String string2)`. De method geeft een integer terug: 0 bij gelijkheid, een negatief getal als de eerste string 'kleiner' is dan de tweede en een positief getal als de eerste string 'groter' is dan de tweede.
- **compareToIgnoreCase():** idem voorgaande, maar dan niet hoofdlettergevoelig.



Vergelijk strings altijd met **equals()**, nooit met de **==** operator!

### 9.3.2 Strings wijzigen

Zoals eerder reeds gezegd is, is een string immutable. Alle methods die beschreven worden, geven dus een nieuwe string terug! In de voorbeelden wordt de referentie van de nieuwe string toegekend aan de bestaande string.

#### 9.3.2.1 replace()

Met deze method kan in een string een karakter vervangen worden door een ander karakter:

```
public String replace(char oldChar, char newChar)
```

Voorbeeld:

```
String woord = "hallo";  
woord = woord.replace('a', 'e');
```

 (1)

- (1) De replace method heeft twee argumenten. Het eerste argument is het te vervangen karakter en het tweede argument is het nieuwe karakter.  
De stringvariabele `woord` bevat na het uitvoeren van de `replace()` method de waarde "hello".

#### 9.3.2.2 toLowerCase()

Deze method zet de inhoud van een string volledig om in kleine letters. Enkel de hoofdletters worden dus gewijzigd. Cijfers, leestekens, e.d. blijven ongewijzigd.

```
public String toLowerCase()
```

Voorbeeld:

```
String woord = "Hallo";  
woord = woord.toLowerCase();
```

 (1)



- (1) De stringvariabele `woord` bevat na het uitvoeren van de `toLowerCase()` method de waarde "hallo".

#### 9.3.2.3 `toUpperCase()`

```
public String toUpperCase()
```

Deze method is vergelijkbaar met de voorgaande maar er gebeurt een omzetting naar hoofdletters. Voor hetzelfde voorbeeld zou de stringvariabele `woord` na het uitvoeren van de `toUpperCase()` method de waarde "HALLO" bevatten.

#### 9.3.2.4 `trim()`

```
public String trim()
```

Deze method `trim()` verwijdert 'whitespaces' (witruimtes) voor en achter de tekst van de stringvariabele. Met witruimte wordt een spatie bedoeld.

Voorbeeld:

```
String woord = "    Hallo mevrouw    ";  
woord = woord.trim();
```

(1)

- (1) De stringvariabele `woord` bevat na het uitvoeren van de `trim()` method de waarde "Hallo mevrouw". De spaties voor en achter de tekst zijn verwijderd. De spaties tussen de woorden in de tekst worden niet verwijderd.

### 9.3.3 Strings onderzoeken

Met de volgende methods kan je String-objecten onderzoeken:

- **`length()`**: geeft het aantal tekens van een string.
- **`isEmpty()`**: geeft `true` enkel en alleen indien de lengte (`length()`) van een string 0 is. In het andere geval is het resultaat `false`.
- **`substring()`**: kopieert een deel van de string.

```
public String substring(int beginIndex, int endIndex)
```

Het deel dat gekopieerd wordt, begint bij de `beginindex` en eindigt bij de `eindex-1`. De index van een string begint altijd bij 0 (het eerste karakter van de string).

Deze method kan fouten (exceptions) genereren indien de `beginIndex < 0`, de `eindIndex > lengte` van de string of de `eindIndex < beginIndex`.

- **`charAt()`**: retourneert het 'character' dat op een bepaalde positie staat.

```
public char charAt(int index)
```

Ook hier bestaan er exceptions indien de index negatief is of groter is dan de lengte van de string.

- **`indexOf()`** en **`lastIndexOf()`**: beide methods onderzoeken of een bepaalde substring onderdeel is van de string. De eerste method zoekt van voor naar achter, de laatste van achter naar voor.

Beide methods geven de index terug waar de substring gevonden is, of **-1** indien de substring niet gevonden is.

```
public int indexOf(String str)
public int lastIndexOf(String str)
```



De meeste van bovenstaande stringmethods hebben één of meer overloaded varianten. Bovendien is deze opsomming onvolledig.

Voor de volledige lijst van stringmethods en voorbeelden: zie de API-documentatie.

### 9.3.4 Een voorbeeld

Op het gebruik van stringmethods volgt een voorbeeld met een controle van een e-mailadres op geldigheid. Bij wijze van oefening testen we op volgende voorwaarden:

- De lengte is minstens 4 tekens.
- Er moet (en er mag) maar één @ in het adres voorkomen en dit teken mag niet in de eerste en niet in de laatste positie staan.
- Na de @ moet er minstens één punt komen.

```
package jpfhfdst09;

public class EmailControle {

    public static void main(String args[]) {

        String email1 = "kamiel.kafka@praag.be";
        String email2 = "kamiel@kafka@praag.be";
        String email3 = "kamiel.kafka@";
        String email4 = "kamiel.kafka@praag";
    }
}
```

```

        System.out.println("\nControle van: " + email1);
        controleer(email1);
        System.out.println("\nControle van: " + email2);
        controleer(email2);
        System.out.println("\nControle van: " + email3);
        controleer(email3);
        System.out.println("\nControle van: " + email4);
        controleer(email4);
    }

    private static void controleer(String s) {
        String antw="";
        int plaats;
        int lengte = s.length();
        if (lengte < 4) antw += "e-mail adres is te kort\n";

        plaats = s.indexOf('@'); // @ mag maar 1 keer voorkomen
        if (plaats < 0)
            antw +=("Er moet een @ in het adres voorkomen.\n");
        if (plaats == 0 || plaats == (--lengte))
            antw +=("Een @ mag niet in de eerste of de laatste positie
                    staan.\n");
        if (plaats >= 0 && plaats != s.lastIndexOf('@'))
            antw +=("Er mag maar één @ voorkomen.\n");
        if (s.lastIndexOf('.') < s.lastIndexOf('@'))
            antw+=("Na de @ moet er nog minstens één punt volgen.\n");
        if (antw.length() == 0) {
            antw = "Alle controles zijn goed bevonden";
        }
        System.out.println(antw);
    }
}

```

## 9.3.5 Strings opsplitsen

### 9.3.5.1 split()

Aan de hand van de method `split()` kan je een String opsplitsen in verschillende stukjes. Deze method bestaat in twee varianten :

`String[] voorbeeld.split(String regex):` regex is hier een delimiter (symbool) die bepaalt waar de string gesplitst wordt.

`String[] voorbeeld.split(String regex, int limit):` de delimiter bepaalt opnieuw waar de string gesplitst wordt en een int-parameter zegt hoeveel keer de delimiter gebruikt wordt. De rest van de string komt in het laatste arrayelement te staan (zie voorbeeld).

Een voorbeeld:

```

String tekst = "Dit is een stukje tekst";
String[] stukjes = tekst.split(" ");
for (int i = 0; i < stukjes.length; i++) {
    System.out.println(stukjes[i]);
}

```

(1)

- (1) De tekst wordt geplitst op een spatie (= de delimiter) en elke (deel)string wordt gestockeerd in een array. Dit geeft volgend resultaat:

```
Dit
is
een
stukje
tekst
```

Nog een voorbeeld waarbij ook de int-parameter van de split method gebruikt wordt:

```
String tekst = "Dit is een stukje tekst";
String[] stukjes = tekst.split(" ", 3);
for (int i = 0; i < stukjes.length; i++) {
    System.out.println(stukjes[i]);
}
```

(1)

- (1) De tekst wordt geplitst op een spatie (= de delimiter), maar slechts in 3 delen. Het derde deel bevat de rest van de string. Elke (deel)string wordt gestockeerd in een array. Deze array bevat dus 3 elementen. Dit geeft volgend resultaat:

Het resultaat :

```
Dit
is
een stukje tekst
```

## 9.4 Strings: conversie van en naar primitive types

In de praktijk zal je dikwijls getallen, voorgesteld als een String, willen omzetten naar een ander gepast primitief type om er bijv. mee te kunnen rekenen. Omgekeerd kan je een getal, opgeslaan in een variabele van een primitive type, converteren naar een String.

Om de stringconversies toe te lichten, volgt eerst een korte uitleg over wrapperclasses.

### 9.4.1 Wrapperclasses

Elk primitive type in Java heeft een bijhorende wrapperclass die hetzelfde type voorstelt maar dan een type van een echte class is. Bijv. een int heeft als wrapperclass de class Integer. Dit maakt het mogelijk om waarden van een primitive type te gebruiken op plaatsen waar een objecttype vereist is. Dit proces heeft **autoboxing**. Wanneer een waarde van een primitive type gebruikt wordt in een context waarin een objecttype vereist wordt, gebruikt de compiler autoboxing om de waarde van het primitive type te 'verpakken' in een passend wrapperobject.

Het omgekeerde kan ook. Wanneer een object van een wrappertype gebruikt wordt in een situatie waarin de waarde van het bijhorende primitive type nodig is, zal de compiler automatisch **unboxing** gebruiken.

Tabel met de primitive types en hun bijhorende wrapperclass:

Primitive type	Wrappertype
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

### 9.4.2 Een String omzetten naar een primitive type

Hiervoor heeft elke wrapperclass een method `parse`: bijv. `parseInt`, `parseDouble`, enz... Het argument van deze method is een String variabele die omgezet moet worden naar een double, een int, of...

Een voorbeeld :

```
String tekst = "5.0";
double temperatuur = Double.parseDouble(tekst);
System.out.println(temperatuur);

tekst = "7";
int geluksgetal = Integer.parseInt(tekst);
System.out.println(geluksgetal);
```

### 9.4.3 Een primitive type omzetten naar een String

Voor de omgekeerde beweging, van een primitief type naar String gebruik je een method van de class String: `String.valueOf()`. Het argument van deze method bevat dan de variabele van het primitief type.

Een voorbeeld :

```
double temperatuur = 5.0;
String tekst = String.valueOf(temperatuur);
System.out.println(tekst);

int geluksgetal = 7;
tekst = String.valueOf(geluksgetal);
System.out.println(tekst);
```

## 9.5 De class StringBuilder

Zoals eerder reeds is aangehaald is een String op zich immutable. Strings kunnen dus niet gewijzigd worden. De class `StringBuilder` is geschikt om strings te manipuleren. `StringBuilder` heeft efficiënte

methods om strings samen te voegen (append), om strings tussen te voegen (insert), enz. en is bovendien mutable (wijzigbaar). Een StringBuilder kan dus wijzigen in inhoud en in lengte.

Een voorbeeld waarbij enkele methods en het gebruik van StringBuilder worden aangetoond:

```

StringBuilder naam = new StringBuilder ("Eddy");                                (1)
System.out.println(naam.length());

naam.append(' ');                                                                (2)
naam.append("Wally");
naam.append(" is de nr 1");
System.out.println(naam);
naam.insert(0, "De grote ");                                                    (3)
System.out.println(naam);
naam.delete(0, 9);                                                              (4)
System.out.println(naam);
System.out.println(naam.length());

naam = new StringBuilder ("De_Ronde_van_Vlaanderen");
for (int teller = 0; teller < naam.length(); teller++) {
    if (naam.charAt(teller) == '_') {
        naam.setCharAt(teller, ' ');                                           (5)
    }
}
System.out.println(naam);

```

Bovenstaande code geeft volgende output:

```

4
Eddy Wally is de nr 1
De grote Eddy Wally is de nr 1
Eddy Wally is de nr 1
21
De Ronde van Vlaanderen

```

Wat uitleg bij de code:

- (1) Een StringBuilder-object wordt geïnitieerd met de tekst 'Eddy'.
- (2) Vervolgens wordt mbv de append()-method achteraan de tekst een spatie toegevoegd.
- (3) Mbv de insert()-method wordt vanaf positie 0 een tekst tussengevoegd. Positie 0 slaat op het begin van de string, maw vooraan.
- (4) Mbv de delete()-method worden posities verwijderd. In dit statement worden 9 posities verwijderd te beginnen vanaf de 0<sup>de</sup> positie.
- (5) In de for-lus worden alles underscores in een tekst vervangen door spaties. Met de method charAt() wordt een individueel karakter op een positie in de StringBuilder onderzocht. Mbv de method setCharAt() wordt een individueel teken op een positie in de StringBuilder vervangen door een ander teken.

Ook Strings kunnen geconcateneerd worden met de + operator. De compiler kan intussen reeds vele bewerkingen optimaliseren, waardoor de aanmaak van nieuwe objecten beperkt blijft: de

stringconcatenatie `a+b+c` wordt door de compiler geoptimaliseerd als

```
new StringBuilder (a).append(b).append(c).toString()
```

## 9.6 De class StringBuffer

De mogelijkheden van de class `StringBuffer` zijn vergelijkbaar met deze van `StringBuilder`, doch er is één groot verschil. De class `StringBuffer` is performanter dan de class `StringBuffer` omdat `StringBuffer` thread safe is en `StringBuilder` niet. `StringBuffer` objecten kan je veilig gebruiken in een multi-threaded omgeving. Door niet te synchroniseren is de performantie van `StringBuilder` beter. (Meer uitleg over multi-threading en synchronisatie volgt verder in de cursus bij het hoofdstuk van Multi-threading).

Voorlopig kan je zowel `StringBuilder` als `StringBuffer` gebruiken. Onthou hierbij dat de class `StringBuffer` een multi-threaded alternatief is voor de class `StringBuffer`.

## 9.7 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 9:

- Klinkers
- Palindroom
- Rekenaar

## 10 Interfaces

---

Het begrip interface is eerder reeds aan bod gekomen bij de module Objectgeoriënteerde principes. Wat is een interface? Een interface is een abstracte class waarin:

- abstracte methods en/of
- publieke constanten (final membervariabelen)

gedeclareerd worden. De abstracte methods staan voor een bepaald **gedrag**. De methods zijn abstract omdat ze niet in de interface geïmplementeerd worden. Ze bevatten geen code. Het is enkel een declaratie van de method.

Een interface wordt aangemaakt met het keyword *interface* in plaats van het gebruikelijke keyword *class*. Een interface bevat ook geen constructor.

Voorbeeld:

```
public interface Gereedschap {  
    //statements  
}
```

De default access modifier van de members in een interface is public, iets anders is niet toegelaten. De interface zelf kan public zijn of package-access (package visibility) hebben.

Een class kan één of meerdere interfaces implementeren. Dit betekent dat alle (!) methods die in de interface(s) gedeclareerd worden, gedefinieerd moeten worden in de class (d.w.z. code dienen te bevatten). Een class ondertekent als het ware een contract waarmee de class zegt alle methods uit de interface(s) te zullen beschrijven.

Het implementeren van een interface in een class gebeurt door in de class-header het keyword *implements* gevolgd door de naam van de interface te vermelden. Indien meerdere interfaces geïmplementeerd worden, dienen de namen van de interfaces gescheiden te worden door een komma.

Voorbeeld:

```
public class Graafwerktuig implements Gereedschap {  
    // statements  
}  
  
public class Graafwerktuig implements Gereedschap, Serializable {  
    // Serializable is een bestaande interface van Java SE (zie verder)  
    // statements  
}
```

Inheritance kan ook toegepast worden bij interfaces. Een interface kan dus erven van een andere interface.

Met interfaces kan je classes met mekaar in verband brengen die geen gemeenschappelijke base class hebben, maar gemeenschappelijk **gedrag**. Je declareert het gemeenschappelijk gedrag in een interface. Je implementeert de interface in de class en je werkt de methods van de interface (het gedrag) uit in de class door code te schrijven.



Een voorbeeld: het bepalen van het volume. De interface kan bijv. de method *berekenVolume()* bevatten. De classes Bestelwagen en Diepvries kunnen de interface implementeren en de method *berekenVolume()* kan er uitgeschreven worden om respectievelijk het volume te berekenen voor een bestelwagen en voor een diepvries. Beide classes hebben met elkaar gemeen dat ze een bepaald volume hebben, maar ze hebben geen gemeenschappelijke base class. Een bestelwagen is een voertuig en een diepvries is een elektrisch apparaat. Beide classes hebben geen of een verschillende base class.



Opmerking:

Wanneer je om één of andere reden niet alle methods uit de interface kan beschrijven in een class, dan moet je die class abstract maken.

## 10.1 Declaratie en beschrijving

Als voorbeeld van een interface beschouw je een bedrijfskost. In een bedrijf zijn er o.a. personeelskosten, materiaalkosten, enz. De interface *Kost* heeft twee methods: *bedragKost()* en *personeelsKost()*.

Daarna zal je bij drie totaal verschillende objecten, nl. een werknemer, een vrachtwagen en een kopieermachine, de interface *Kost* implementeren.

De interface *Kost*:

```
package jpfhfdst10;
public interface Kost {
    public abstract double bedragKost();
    public abstract boolean personeelsKost();
}
```

(1)

(1) *public* mag weggelaten worden, want default is de access modifier van methods in een interface *public*.

Deze interface declareert dus twee methods *bedragKost()* en *personeelsKost()*. Het is enkel een declaratie. Een beschrijving van deze methods komt in de class, maar je kan wel afleiden dat de method *bedragKost()* een *double* zal returnen en de method *personeelsKost()* een *boolean* (is dit al dan niet een personeelskost).

## 10.2 Implementatie in een class

De class *Werknemer* zal deze *interface* *implementeren*. In de class *Werknemer* zullen dus beide methods uit de interface gedefinieerd moeten worden.

```
package jpfhfdst10;
public class Werknemer implements Kost {
    String naam;
    double wedde;

    public Werknemer(String naam, double wedde) {
        this.naam = naam;
        this.wedde = wedde ;
    }
}
```

```

    }
    public double getWedde() {
        return wedde;
    }
    public String getNaam() {
        return naam;
    }
    @Override
    public double bedragKost() {
        return wedde;
    }
    @Override
    public boolean personeelsKost() {
        return true;
    }
    @Override
    public String toString() {
        return naam + ";" + wedde;
    }
}

```

De interface *Kost* wordt geïmplementeerd. Dat houdt in dat de class code dient te bevatten voor beide methods van de interface. De code is hier in dit voorbeeld vrij eenvoudig gehouden. De method *bedragKost()* retourneert gewoon de waarde van de membervariabele *wedde* en de method *personeelsKost()* retourneert *true* aangezien het inderdaad om een personeelskost gaat.

Merk op dat voor deze methods de `@Override` geschreven wordt.

In het bedrijf zijn ook kopieermachines aanwezig en die implementeren eveneens de interface *Kost*:

```

package jpfhfdst10;
public class Kopieermachine implements Kost {
    private final String merk;
    private final double kostPerBlz;
    private final int aantalBlz;

    public Kopieermachine(String merk, double kostPerBlz, int aantalBlz) {
        this.merk=merk;
        this.kostPerBlz=kostPerBlz;
        this.aantalBlz=aantalBlz;
    }

    @Override
    public double bedragKost() {
        return kostPerBlz*aantalBlz;
    }

    @Override
    public boolean personeelsKost() {
        return false;
    }

    public String getMerk() {
        return merk;
    }

    public int getAantalBlz() {
        return aantalBlz;
    }
}

```

Ook hier opnieuw het sleutelwoord *implements* in de classheader, gevolgd door de naam van de interface die de class implementeert.

De class bevat drie private membervariabelen *merk*, *kostPerBlz* en *aantalBlz* en een constructor. Vervolgens worden de twee methods uit de interface gedefinieerd: het bedrag van de kost wordt berekend door het aantal pagina's te vermenigvuldigen met de kostprijs per pagina. Het betreft hier geen personeelskost dus retourneert de method *personeelsKost* *false*. Tenslotte nog enkele getters.

Het bedrijf heeft ook vrachtwagens. Er wordt de class *Vrachtwagen* voorzien, die uiteraard ook kosten heeft. Daarom wordt in de class *Vrachtwagen* de interface *Kost* geïmplementeerd. Naast de kosten is er ook een afschrijving voor de vrachtwagen. Hiervoor wordt eerst een interface *Afschrijving* geschreven die dan geïmplementeerd kan worden in de class *Vrachtwagen*.

```
package jpfhfdst10 ;
public interface Afschrijving {
    int termijn();           //is impliciet public abstract
    double jaarlijksBedrag(); //idem
}
```

De interface declareert twee methods: *termijn()* en *jaarlijksBedrag()* die aangeven over hoeveel termijnen iets wordt afgeschreven en hoeveel het jaarlijks af te schrijven bedrag is.

De implementatie van deze interface én van de interface *Kost* ziet er dan in de class *Vrachtwagen* als volgt uit:

```
package jpfhfdst10;
public class Vrachtwagen implements Kost, Afschrijving {
    private final String merk;
    private final double kostPerKm;
    private final int aantalKm;
    private final double aankoopPrijs;
    private final int voorzieneLevensduur;

    public Vrachtwagen(String merk, double kostPerKm, int aantalKm,
                       double aankoopPrijs, int voorzieneLevensduur) {
        this.merk = merk;
        this.kostPerKm = kostPerKm;
        this.aantalKm = aantalKm;
        this.aankoopPrijs = aankoopPrijs;
        this.voorzieneLevensduur = voorzieneLevensduur;
    }

    @Override
    public double bedragKost() {
        return kostPerKm * aantalKm;
    }

    @Override
    public boolean personeelsKost() {
        return false;
    }

    @Override
    public int termijn() {
        return voorzieneLevensduur;
    }
}
```

(1)

(1)

(2)

```

@Override
public double jaarlijksBedrag() {
    return aankoopPrijs / voorzieneLevensduur;
}

public String getMerk() {
    return merk;
}

public int getAantalKm() {
    return aantalKm;
}
}

```

(2)

(1) Dit is de implementatie van de twee methods van de interface Kost.

(2) Dit is de implementatie van de twee methods van de interface Afschrijving.

Volgend main()-programma berekent de kosten van het bedrijf:

```

package jpfhfdst10;
public class KostProg {
    public static void main(String[] args) {
        Werknemer eddy = new Werknemer ("Eddy", 2000.0);
        Werknemer elly = new Werknemer ("Elly", 2500.0);
        Vrachtwagen daf = new Vrachtwagen("DAF", 0.35, 25000, 150000.0, 8);
        Kopieermachine konica = new Kopieermachine("Konica", 0.02, 9000);

        double personeelsKosten=0.0;
        double andereKosten=0.0;

        //kosten van Eddy tellen
        if (eddy.personeelsKost())
            personeelsKosten += eddy.bedragKost();
        else
            andereKosten += eddy.bedragKost();

        //kosten van Elly tellen
        if (elly.personeelsKost())
            personeelsKosten += elly.bedragKost();
        else
            andereKosten += elly.bedragKost();

        //kosten van DAF tellen
        if (daf.personeelsKost())
            personeelsKosten += daf.bedragKost();
        else
            andereKosten += daf.bedragKost();

        //kosten van Konica tellen
        if (konica.personeelsKost())
            personeelsKosten = personeelsKosten + konica.bedragKost();
        else
            andereKosten = andereKosten + konica.bedragKost();

        System.out.println("Personeelskosten :" + personeelsKosten);
        System.out.println("Andere kosten : " + andereKosten);
    }
}

```

Dit geeft volgende output:

```
Personeelskosten :4500.0
Andere kosten : 8930.0
```

Er wordt een totaal berekend van de personeelskost en van de overige kosten. Er worden enkele objecten aangemaakt en per object wordt nagegaan of het om een personeelskost gaat of om een andere kost. Zo kan er een correct totaal van beide kosten berekend worden.

### 10.3 Interface als een data type

Net zoals je bij inheritance met polymorfisme een array kan maken van references naar objecten met eenzelfde base class, kan je een array maken van reference variabelen met als type een interface. Je laat dan elk van die reference variabelen verwijzen naar een object van een class die deze interface implementeert. Vervolgens kan voor elk element in de array een method opgeroepen worden die gedeclareerd is in die interface. Methods die niet in de interface voorkomen kunnen niet uitgevoerd worden. Voor een Werknemer-object kan dan bijvoorbeeld *getWedde()* niet uitgevoerd worden.

In onderstaand programma wordt dit toegepast:

```
package jpfhfdst10;
public class KostProg2 {
    public static void main(String[] args) {
        Kost[] kosten = new Kost[4];
        kosten[0] = new Werknemer ("Eddy", 2000.0);
        kosten[1] = new Werknemer ("Elly", 2500.0);
        kosten[2] = new Vrachtwagen("DAF", 0.35, 25000, 150000.0, 8);
        kosten[3] = new Kopieermachine("Konica", 0.02, 9000);

        double personeelsKosten=0.0;
        double andereKosten=0.0;

        for (Kost eenKost:kosten) {
            if (eenKost.personeelsKost())
                personeelsKosten += eenKost.bedragKost();
            else
                andereKosten += eenKost.bedragKost();
        }
        System.out.println("Personeelskosten :" + personeelsKosten);
        System.out.println("Andere kosten : " + andereKosten);
    }
}
```

- (1) Declaratie van de array met als data type de interface Kost.
- (2) Vervolgens verwijzen vier references naar objecten van een class die de interface Kost implementeert.
- (3) De array *kosten* wordt doorlopen en voor elk element (*eenKost*) wordt de body van de lus uitgevoerd. Elke reference wordt onderzocht. Op deze manier wordt het gepast bedrag verhoogd, m.a.w. de personeelskosten of andere kosten.

## 10.4 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 10:

- VoertuigenI
- Voorwerpen

## 11 Packages

---

### 11.1 Algemeen

Een package is een groep classes en interfaces die logisch bij elkaar horen.

Iedere package heeft een unieke naam.

Je ziet verder in dit hoofdstuk de naamgeving conventie voor packages.

Het gebruik van packages biedt volgende voordelen.

- Overzichtelijkheid.  
De standaard Java library bevat duizenden classes.  
Jij schrijft daarnaast in een enterprise applicatie ook veel classes.  
Packages helpen om structuur te brengen in dit groot aantal classes, zoals directories helpen om structuur te brengen in een groot aantal bestanden.
- Vermijden van naamconflicten.  
Als je met meerdere personen aan een enterprise applicatie werkt, of libraries gebruikt van verschillende firma's, bestaat het risico dat je applicatie meerdere classes bevat met dezelfde naam. Dit leidt tot compileerfouten, tenzij deze classes behoren tot een verschillende package.  
Het woord 'account' heeft bijvoorbeeld meerdere betekenissen.  
Account betekent een rekening waarop je spaart bij een bank.  
Account betekent ook een gebruiker met een gebruikersnaam en paswoord.  
Je kan beide voorstellen met een class Account, op voorwaarde dat de classes behoren tot een verschillende package.  
Je plaatst de class Account die een spaarrekening voorstelt in een package met de naam `be.eenbank.sparen`  
De volledige naam van deze class is: `be.eenbank.sparen.Account`  
Je plaatst de class Account die een gebruiker voorstelt in een package met de naam `be.eenbank.beveiliging`  
De volledige naam van deze class is: `be.eenbank.beveiliging.Account`
- Zichtbaarheid van classes.  
Als je voor een class het sleutelwoord `public` weglaat, is die class enkel zichtbaar voor andere classes binnen dezelfde package als die eerste class. Dit heet een class met package visibility.  
Je past package visibility toe op classes die enkel ten dienste staan van andere classes uit dezelfde package.

### 11.2 Naamgeving conventie voor packages

#### 11.2.1 Algemeen

Iedere package moet een unieke naam hebben.

Dit geldt voor packages die je zelf maakt, maar ook voor packages van libraries die je gebruikt in je applicatie en geschreven zijn door andere firma's.

Om deze uniciteit te bekomen, gebruiken alle Java ontwikkelaars dezelfde naamgeving conventie voor packages. De naam van een package begint met de internet domeinnaam van hun firma, in omgekeerde vorm (vdab.be wordt `be.vdab`).

Gezien internet domeinnamen uniek zijn, zijn op die manier ook package namen uniek, en zijn ook class namen (rekening houdend met hun package namen) uniek.

- VDAB Java ontwikkelaars beginnen de namen van hun packages dus met *be.vdab*.
- Ontwikkelaars van de open source organisatie Apache beginnen de namen van hun packages met *org.apache*.

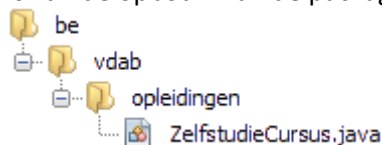
Je schrijft de naam van een package volledig in kleine letters.

Na deze omgekeerde internet domeinnaam, kan de naam van de package uit extra woorden bestaan, gescheiden door een punt. Voorbeeld: *be.vdab.opleidingen*

### 11.3 Voorwaarden voor classes in packages

Een class moet aan twee voorwaarden voldoen om tot een package te behoren:

- De source van de class moet zich bevinden in een directorystructuur die een weerspiegeling is van de opbouw van de package naam.

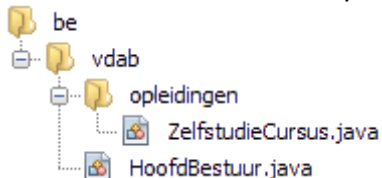


ZelfstudieCursus bevindt zich in de directory structuur *be/vdab/opleidingen*. Je geeft daarmee aan dat de class behoort tot de package *be.vdab.opleidingen*.

Directories hebben een hiërarchische structuur, maar packages niet.

De class ZelfstudieCursus behoort tot de package *be.vdab.opleidingen*, maar niet tot de package *be.vdab*, omdat de class zich niet rechtstreeks in de directory structuur *be/vdab* bevindt.

De class Hoofdbestuur behoort wel tot de package *be.vdab*, omdat ze een rechtstreeks onderdeel is van de directory structuur *be/vdab*.



- De eerste opdracht in de source file moet de opdracht `package` zijn. Je tikt na het keyword `package` de package waartoe de class behoort. ZelfstudieCursus.java begint met `package be.vdab.opleidingen;` Hoofdbestuur.java begint met `package be.vdab;`

Je IDE (zoals NetBeans) helpen je om aan deze voorwaarden te voldoen.

Je maakt een nieuw project met de naam `PackagesUitleg` om dit uit te proberen. Je verwijdert in de tweede stap van de wizard het vinkje bij `Create Main Class`. Je maakt zo een leeg project dat nog geen class en ook geen package bevat.

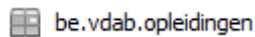
#### 11.3.1 Een package maken in NetBeans

Je klikt met de rechtermuisknop op het project en je kiest `New, Java Package`.

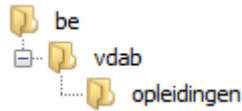
Je tikt `be.vdab.opleidingen` bij `Package Name` en je kiest `Finish`.



Je ziet in het tabblad Projects (links) deze package voorgesteld als



Je ziet in het tabblad Files (naast het tabblad Projects) dat NetBeans de directory structuur heeft aangemaakt die bij deze package hoort



### 11.3.2 Een nieuwe class toevoegen aan een bestaande package

Je selecteert het tabblad Projects.

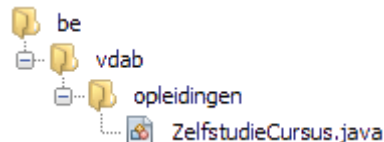
Je klikt met de rechtermuisknop op de package be.vdab.opleidingen.

Je kiest New, Java Class.

Je tikt ZelfstudieCursus bij Class Name en je kiest Finish.

Je ziet dat ZelfstudieCursus.java begint met `package be.vdab.opleidingen;`

Je ziet in het tabblad Files dat NetBeans de source file in de juiste directory plaatste



### 11.3.3 Een nieuwe class toevoegen aan een nieuwe package

Je selecteert het tabblad Projects.

Je klikt met de rechtermuisknop op het project en je kiest New, Java Class.

Je tikt Main bij Class Name.

Je tikt be.vdab.main bij Package en je kiest Finish.

### 11.3.4 Een class naar een andere package verplaatsen

Als voorbereidende stap voeg je een class Cursus toe aan de package be.vdab.main.

Nu verplaats je deze class naar de package be.vdab.opleidingen:

Je sleept in het tabblad Projects de source Cursus.java naar be.vdab.opleidingen. Je kiest Refactor.

NetBeans verplaatst Cursus.java naar de bijbehorende directory en wijzigt in de source de opdracht package naar `package be.vdab.opleidingen;`

## 11.4 Verwijzen naar interfaces en classes uit een package

### 11.4.1 Algemeen

Je kan in een class verwijzen naar een andere class (of interface)

- Een class kan erven van een andere class
- Een class kan een interface implementeren
- Een class kan variabelen bevatten waarvan het type een andere class is

- Een method kan parameters bevatten waarvan het type een andere class is
- Het returntype van een method kan een andere class zijn
- ...

Als deze classes (of interfaces) behoren tot dezelfde package, moet je geen nieuwe opdrachten tikken in je source.

Je ziet dit door de class ZelfstudieCursus te laten erven van de class Cursus.

Gezien beide classes behoren tot dezelfde package, hoeft je enkel volgende wijziging aan te brengen in ZelfstudieCursus.java: `public class ZelfstudieCursus extends Cursus`

Als de class (of interface) waarnaar je verwijst, zich bevindt in een andere package, moet je één van volgende methodes toepassen om compileerfouten te vermijden:

- Je verwijst naar de class met zijn volledige naam (inclusief de package naam).  
Je ziet dit door aan de class Main een private variabele toe te voegen van het type ZelfstudieCursus (die zich een andere package bevindt):  
`private be.vdab.opleidingen.ZelfstudieCursus zelfstudieCursus;`  
Het nadeel van deze method is dat je veel tikwerk hebt.
- Je importeert de volledige package die de class bevat waarnaar je verwijst. Je doet dit met een `import` statement. Je tikt na `import` de naam van de package, gevolgd door een punt en een sterretje.  
Je schrijft `import` statements voor een class, niet binnen een class, maar na het package statement. Als je daarna naar een class verwijst uit de geïmporteerde package, moet je enkel nog de naam van de class vermelden.  
Je wijzigt als voorbeeld de class Main

```
package be.vdab.main;
import be.vdab.opleidingen.*;
public class Main {
    private ZelfstudieCursus zelfstudieCursus; //enkel class naam
}
```

Deze methode wordt afgeraden, omdat je in het `import` statement niet ziet welke classes je wel gebruikt en welke classes je niet gebruikt uit de package.

- Je importeert de class waarnaar je verwijst. Je doet dit ook met een `import` statement. Je tikt na `import` de naam van de package, gevolgd door een punt en de class in die package. Als je daarna naar die class verwijst, vermeld je enkel de naam van de class.  
Je wijzigt als voorbeeld de class Main:

```
package be.vdab.main;
import be.vdab.opleidingen.ZelfstudieCursus;
public class Main {
    private ZelfstudieCursus zelfstudieCursus; //enkel class naam
}
```

Deze methode wordt aanzien als 'best practice'.

### 11.4.2 Ondersteuning van NetBeans

NetBeans helpt je om deze `import` statements toe te voegen, op twee manieren.

Eerste manier: de lamp in de marge (💡).

Je verwijdert in de class Main het import statement om dit uit te proberen.

Je voegt ook een extra private variabele toe: `private Cursus cursus;`

Je ziet in de marge voor elke private variabele een rood bolletje (fout indicator) en een lamp. Je klikt op de lamp. NetBeans stelt als eerste oplossing voor een import statement toe te voegen (Add import for ...). Je kiest deze oplossing.

NetBeans voegt boven in de source een import statement toe.

Tweede manier: de opdracht Fix Imports

Je verwijdert in de class Main de import statements om deze methode uit te proberen.

Je krijgt terug compileerfouten.

Je klikt met de rechtermuisknop ergens in de source en je kiest de opdracht *Fix Imports*. NetBeans voegt boven in de source de nodige import statements toe.

De tweede manier is productiever dan de eerste:

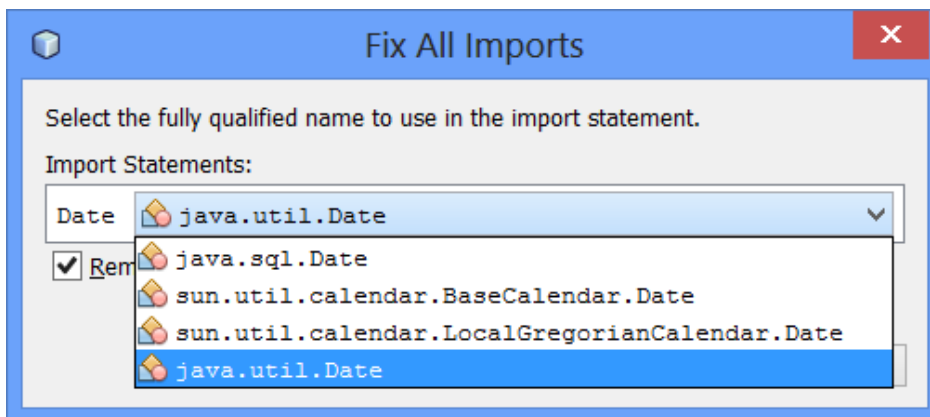
je voegt bij de tweede manier meerdere import statements in één keer toe.

Als een class voorkomt in meerdere packages, laat de tweede manier je kiezen welke package je wil importeren. Je probeert dit uit.

Je voegt aan de class Main een private variabele toe: `private Date date;`

Je klikt met de rechtermuisknop in de source en je kiest de opdracht Fix Imports.

NetBeans vraagt je uit welke package je de class Date wil importeren:



De class Date komt voor in vier packages. Je kiest `java.util.Date`.

Je gebruikt de class `java.sql.Date` enkel als je datums uit een database leest.

Packages die beginnen met `sun` zijn interne packages van de standaard Java Libraries.



Als je een class gebruikt uit de package **java.lang**, zoals de class `String`, moet je geen import statement schrijven.

## 11.5 Jar-bestand

Een grote applicatie bevat veel Java source files (met de extensie .java).

Iedere source wordt gecompileerd tot een bytecode file (met de extensie.class).

Een grote applicatie bevat dus ook veel bytecode files.

De applicatie kan ook tekstbestanden, afbeeldingen... bevatten die in de applicatie gebruikt worden.

Een Jar bestand is een gecomprimeerd bestand. Het bevat alle bytecode bestanden, tekstbestanden, afbeeldingen,... van de applicatie. Een Jar bestand bevat de source files van de applicatie niet: je hebt deze niet nodig om de applicatie uit te voeren.

De compressie techniek om een Jar bestand aan te maken is dezelfde als die bij het zip formaat. Je kan een Jar bestand dus openen met WinZip, WinRar, ....

Als de applicatie zich in een Jar bestand bevindt, neemt de applicatie niet veel plaats in beslag en kan ook performant over het netwerk (of internet) verspreid worden.

De JDK bevat een utility *jar*, waarmee je een Jar bestand aanmaakt in de console. Dit is een omslachtige manier om een Jar bestand aan te maken.

Je IDE (zoals NetBeans) kan eenvoudiger van een project een Jar bestand maken.

Je maakt een nieuw project met de naam JarUitleg om dit uit te proberen.

Je verwijdert in de tweede stap van de wizard het vinkje bij `Create Main Class`. Je maakt zo een leeg project dat nog geen class en ook geen package bevat.

Je maak in dit project een package `be.vdab`.

Je maakt in die package een class `Converter`:

```
package be.vdab;

public class Converter {
    private final static double CENTIMETERS_IN_ONE_INCH = 2.54;

    public double centimetersToInches(double centimeters) {
        return centimeters / CENTIMETERS_IN_ONE_INCH;
    }
}
```

Je maakt in dezelfde package een class `Main`:

```
package be.vdab;
import java.util.Scanner;

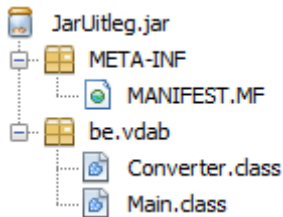
public class Main {
    public static void main(String[] args) {
        System.out.print("centimeters:");
        Scanner scanner = new Scanner(System.in);
        double centimeters = scanner.nextDouble();
        Converter converter = new Converter();
        System.out.println(converter.centimetersToInches(centimeters)
            + " inches");
    }
}
```

Vooraleer je een Jar bestand maakt van dit project, stel je best eerst de main class van het project in. Dit doe je als volgt: klik met de rechtermuisknop op het project en kies Properties. Kies Run bij Categories. Selecteer via de Browse knop bij Main Class het juiste uitvoerbare programma (in dit vb `be.vdab.Main`). Kies de knop Select Main Class en vervolgens kies je OK.

Je maakt dan een Jar bestand van dit project door met de rechtermuisknop te klikken op het project (in het tabblad Projects) en de opdracht Clean and Build te kiezen.

Je ziet in het tabblad Files dat je project nu een directory *dist* bevat.

Deze directory bevat het bestand `JarUitleg.jar`. Je kan dit bestand inzien met het plus teken voor het bestand:



Het onderdeel MANIFEST.MF bevat wat algemene informatie over het Jar bestand, zoals welke class de method `main` bevat (deze is ingesteld via de properties van het project) en een versienummer van de applicatie.

Je zal nu het Jar bestand (dus de applicatie) uitvoeren, los van NetBeans.

Je kopieert via de Windows File Explorer het bestand `JarUitleg.jar` naar een directory `vdab` in de root van `C:`, om straks in de console niet veel tikwerk te hebben.

Je klikt op de start knop van Windows en je tikt de opdracht `cmd` in het vak waar staat "Search programs and files". Je start zo de console.

Je tikt de opdracht `cd \vdab` in de console en je drukt Enter.

Je tikt `java -jar JarUitleg.jar` om het Jar bestand uit te voeren.

## 11.6 Class path

### 11.6.1 Algemeen

Het class path is de verzameling directories en JAR bestanden die Java doorzoekt naar classes, bij het compileren en bij het uitvoeren van een applicatie.

Standaard bestaat het class path uit

- De huidige directory, als je een applicatie uitvoert die nog niet in een Jar bestand verpakt is.
- Het huidige Jar bestand, als je een applicatie uitvoert die wel in een Jar bestand verpakt is.
- De Jar bestanden die de standaard Java libraries bevatten.

Je kan het class path uitbreiden met extra directories en JAR bestanden. Java doorzoekt deze daarna ook bij het compileren en uitvoeren van een applicatie.

### 11.6.2 De class path uitbreiden in NetBeans

Je maakt een nieuw project met de naam `ClassPathUitleg`.

Je verwijdert in de tweede stap van de wizard het vinkje bij `Create Main Class`. Je maakt zo een leeg project dat nog geen class en ook geen package bevat.

Je wil in dit project de class `Converter` oproepen uit het vorige project (`JarUitleg`). Je breidt daartoe de class path van het nieuw project uit met een verwijzing naar het Jar bestand van het vorige project.

Je doet dit op de volgende manier:

- Je klikt met de rechtermuisknop op het project en kiest `Properties`. Je kiest bij `Categories` voor `Libraries`.
- Je kiest `Add JAR/Folder...`
- Je zoekt het bestand `JarUitleg.jar` van het vorig project en je kiest `Open`. Vervolgens `OK`.

Je maakt een package `be.vdab`. Je maakt daarin een class `ConversieProg`, waarin je de class `Converter` (uit het andere project) oproept:

```
package be.vdab;
public class ConversieProgr {

    public static void main(String[] args) {
        Converter converter = new Converter();
        for (int centimeters = 1; centimeters <= 10; centimeters++) {
            System.out.println(centimeters + " cm = "
                               + converter.centimetersToInches(centimeters) + " inches");
        }
    }
}
```

Merk op dat er geen import statement nodig is.

## 11.7 Oefeningen



Zie takenbundel: maak de oefening die hoort bij hoofdstuk 11:  
- Voorwerpen (vervolg)

## 12 Exception Handling

---

In elk programma, hoe goed het ook geschreven is, kunnen zich fouten of onverwachte omstandigheden voordoen. Hiermee worden uitzonderlijke situaties, exceptions, bedoeld. Bijv. het ontbreken van een bestand dat moet ingelezen worden, geen connectie kunnen maken met de database, enz.

Dergelijke omstandigheden kunnen voorzien worden in het programma. Hiervoor schrijf je code, nl. code die de fout vaststelt (fout-veroorzakende code) en code die de fout opvangt (fout-afhandelende code). Dit wordt exception handling genoemd. Daar waar de fout zich voordoet wordt een exception gethrowed en deze exception wordt elders in de code netjes opgevangen. Dit gebeurt in een catch-block. Exceptions kunnen gethrowed worden door de JVM tijdens de uitvoer van het programma. Je kan echter ook zelf code schrijven die een exception throwt.

Exceptions signaleren dus ongewone omstandigheden die je tracht op te vangen. Naast exceptions zijn er ook errors. Errors zijn fouten die je niet opvangt. Ze zijn meestal vrij fataal.

Zowel exceptions als errors zijn objecten of instances van classes die erven van de bestaande class Throwable. Throwable heeft twee subclasses: *Exception* en *Error*.

Je zal in je code zelf echter bijna enkel exceptions voorzien. Zo kan je gebruik maken van bestaande classes. In de package *java.lang* is de class Exception beschreven. De class Exception heeft op zich ook weer afgeleide classes bijv. RuntimeException. Hiervan zijn dan weer NullPointerException en IndexOutOfBoundsException afgeleid.

In de package *java.io* zijn input-output Exceptions beschreven. Zo is er de class IOException. Hiervan afgeleid is er ook EOFException, FileNotFoundException, enz.

Er zijn vele bestaande Exception classes. Deze kan je dus gebruiken. Kies die Exception die je probleem het best omschrijft. Naast de bestaande Exception classes, kan je ook zelf een Exception class schrijven.

### 12.1 Exceptions afhandelen in een try-catch-blok

Veronderstel volgende code waarbij een deling door 2 getallen hard gecodeerd is. De getallen voor de deling kunnen ook ingegeven zijn of invoer zijn van een bestand.

```
package jpfhfdst12;
public class IntroInExceptions {
    public static void main(String[] args) {
        int result = 7 / 0;
        System.out.println(result);
    }
}
```

De deling door nul veroorzaakt tijdens de uitvoer van dit main()-programma een exception, nl. een java.lang.ArithmeticException. Deze exception wordt gethrowed door de JVM en de uitvoer van het programma stopt. Om te vermijden dat in dergelijke situaties het programma stopt, kan de exception opgevangen worden.

Werkwijze:

De code die een fout kan veroorzaken wordt geplaatst in een **try-blok**. Na het try-blok volgt een **catch-blok** met tussen ronde haken het type exception dat wordt opvangen en een naam voor het opgevangen exception object. Code die de fout afhandelt plaats je in dat catch-blok. Tot slot kan nog een **finally**-blok voorzien worden. De code in het finally-blok wordt ALTIJD uitgevoerd, ongeacht of tijdens de uitvoer van het try-blok een fout is opgetreden of niet.

```
package jpfhfdst12;
public class IntroInExceptions {
    public static void main(String[] args) {
        try {
            int result = 7 / 0;
            System.out.println("Deze code wordt alleen uitgevoerd" +
                               " wanneer er geen fout optreedt.");
            System.out.println(result);
        }
        catch (ArithmeticException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

- (1) Code die een exception kan veroorzaken wordt in een try-blok geplaatst. Wanneer de exception zich voordoet, stopt de verdere uitvoer van het try-blok. In dit voorbeeld zijn dat de twee statements met een System.out.println().
- (2) De exception wordt opgevangen in het catch-blok. Het betreft hier een catch-blok voor het opvangen van een ArithmeticException.  
Een exception is een object. Het heeft hier de naam ex. Van dit object wordt er informatie getoond. Deze info wordt opgevraagd met de method getMessage(). Deze method is vrijwel bij elke Exception class voorzien. Hier geeft deze method de reden van de exception (/ by zero).

Nog een voorbeeld waarbij een andere exception gethrowed wordt door de JVM. Voeg volgende code toe:

```
String tekst = "abc";
int tekstInGetalwaarde = Integer.parseInt(tekst);
int result = tekstInGetalwaarde / 4;
System.out.println(result);
```

De uitvoer van het programma stopt weer. Dit keer betreft het een NumberFormatException. De letters "abc" kunnen immers niet omgezet worden naar een getalwaarde.  
Wijzig de code zoals onderstaand om de exception op te vangen:

```
try {
    String tekst = "abc";
    int tekstInGetalwaarde = Integer.parseInt(tekst);
    int result = tekstInGetalwaarde / 4;
    System.out.println(result);
}
```



```

    }
    catch (NumberFormatException ex) {
        System.out.println(ex.getMessage());
    }
}

```

(1)

- (1) Het betreft hier een catch-blok voor het opvangen van een `NumberFormatException`.

De method `Integer.parseInt()`; veroorzaakt de fout. Deze reden van de exception wordt getoond (For input string: "abc").

De `NumberFormatException` is een afgeleide class van `IllegalArgumentException`. Daarom kan het catch statement ook het volgende zijn:

```
catch (IllegalArgumentException ex)
```

Een `NumberFormatException` object is immers ook een `IllegalArgumentException` object.

Nog een derde voorbeeld waarbij een andere exception gethrowed wordt door de JVM. Voeg volgende code weer toe:

```
int[] cijfers = { 12, 5, 28, 37};
System.out.println("Het 7e element is: " + cijfers[6]);
```

De uitvoer van het programma stopt weer. Dit keer betreft het een `ArrayIndexOutOfBoundsException`. De array bevat slechts 4 elementen waardoor je bij het opvragen van het 7<sup>e</sup> element buiten de array gaat. Wijzig de code zoals onderstaand om de exception op te vangen:

```

try {
    int[] cijfers = { 12, 5, 28, 37};
    System.out.println("Het 7e element is: " + cijfers[6]);
}
catch (ArrayIndexOutOfBoundsException ex) {
    System.out.println(ex.getMessage());
}

```

(1)

- (1) De `ArrayIndexOutOfBoundsException` is een afgeleide class van `IndexOutOfBoundsException`.

Daarom kan het catch statement ook het volgende zijn:

```
catch (IndexOutOfBoundsException ex)
```

Een `ArrayIndexOutOfBoundsException` object is immers ook een `IndexOutOfBoundsException` object.

Alle drie getoonde exceptions zijn voorbeelden van runtime exceptions. Het zijn exceptions die kunnen voorkomen tijdens de uitvoer van het programma. Uiteindelijk is de base class voor deze exceptions `RuntimeException`. Dat wil ook zeggen dat het catch statement in alle drie de gevallen vervangen kan worden door `catch (RuntimeException ex)`. Dit is nochtans geen goed idee. Beter is om de exception te specificeren zodat per specifieke exception er een specifieke actie uitgevoerd kan worden. Immers een fout die optreedt, kan meerdere oorzaken hebben. Zo kan je per oorzaak eventueel een andere actie uitvoeren in het betreffende catch-blok.

### 12.1.1 Het finally-blok

Het finally-blok is het laatste blok van een try-catch-blok en wordt dus na het catch-blok geplaatst. De code in het finally-blok wordt **altijd** uitgevoerd, ongeacht of de uitvoering van de code in het try-blok gelukt is of niet.

Even een vergelijking met de werkelijkheid: wanneer je iets bakt in de oven, dan zal de oven altijd uitgezet moeten worden, ongeacht of het bakken van het gerecht gelukt is of niet.

Dit finally-blok wordt vaak gebruikt voor het sluiten van resources, bijv. sluiten van bestanden, sluiten van de databaseconnectie, enz.

Het finally-blok kan ook een return-statement bevatten.

Aan bovenstaand voorbeeld wordt een zeer eenvoudig finally-blok toegevoegd:

```
try {
    int[] cijfers = { 12, 5, 28, 37};
    System.out.println("Het 7e element is: " + cijfers[6]);
}
catch (ArrayIndexOutOfBoundsException ex) {
    System.out.println(ex.getMessage());
}
finally {
    System.out.println("Dit is een poging voor het tonen van de"
        + " waarde van een element.");
}
```

## 12.2 Meerdere catch-blokken bij één try-blok.

Dit wordt aangetoond aan de hand van een voorbeeld:

```
String tekst = "2";
// String tekst = "6";
// String tekst = "abc";

try {
    int[] cijfers = { 12, 5, 28, 37};
    System.out.println("Element uit de array: " +
        cijfers[Integer.parseInt(tekst)]);
    System.out.println("Deze code wordt alleen uitgevoerd" +
        " wanneer er geen fout optreedt.");
}
catch (ArrayIndexOutOfBoundsException ex) {
    System.out.println("ArrayIndexOutOfBoundsException: " +
        ex.getMessage());
}
catch (NumberFormatException ex) {
    System.out.println("NumberFormatException: " + ex.getMessage());
}
catch (RuntimeException ex) {
    System.out.println("RuntimeException: " + ex.getMessage());
}
```

(1)

- (1) Op deze regel code staan twee statements met een risico: het parsen kan een fout veroorzaken of een onbestaand element van de tabel wordt aangesproken.  
Afhankelijk van de waarde van de string-variabele *tekst* treedt er al dan niet een fout op.

Om toch een gepaste foutboodschap weer te geven, kunnen er meerdere catch-blokken aan één try-blok gekoppeld worden. Op deze manier worden dus verschillende types exceptions opgevangen. Hierbij is de volgorde van de catch-blokken van belang. Hanteer steeds de volgorde van een \*specifieke exception naar een algemene exception. Van zodra de exception overeenkomt met de exception van het catch-blok wordt de code van dat catch-blok uitgevoerd.

In bovenstaand voorbeeld is het dus van belang dat het catch-blok voor een RuntimeException als laatste catch-blok staat, aangezien zowel een `ArrayIndexOutOfBoundsException` als een `NumberFormatException` RuntimeExceptions zijn. Wanneer het catch-blok voor een RuntimeException als eerste catch-blok zou staan, zou de code van de overige twee catch-blokken nooit uitgevoerd worden. De exception die optreedt is immers een RuntimeException, het eerste catch-blok komt overeen met de optredende fout, dus die code wordt uitgevoerd.

### 12.3 Multi catch

Multi catch in Java bestaat vanaf versie 7.

In bovenstaand voorbeeld wordt voor elke exception een apart catch-blok geschreven. De boodschap die getoond wordt verschilt lichtjes per exception (per fout). Stel dat in dit geval het weergeven van de message van de fout zonder bijkomende tekst voldoende was, dan kan je gebruik maken van het multi catch blok. Het multi catch blok voorkomt zo het schrijven van dubbele code.

```
String tekst = "2";  
// String tekst = "6";  
// String tekst = "abc";  
  
try {  
    int[] cijfers = { 12, 5, 28, 37};  
    System.out.println("Element uit de array: " +  
        cijfers[Integer.parseInt(tekst)]);  
    System.out.println("Deze code wordt alleen uitgevoerd" +  
        " wanneer er geen fout optreedt.");  
}  
catch (ArrayIndexOutOfBoundsException | NumberFormatException ex) { (1)  
    System.out.println(ex.getMessage());  
}
```

- (1) Je kan met één catch handler meerdere types exceptions opvangen. Dit catch-blok vangt exceptions op van het type `ArrayIndexOutOfBoundsException` en van het type `NumberFormatException`.  
De exceptions worden gescheiden d.m.v. de logische or operator |.

### 12.4 Eigen exceptions maken

Naast de bestaande exceptions, kan je ook zelf exceptions maken. Je kan dus voorzien dat een bepaalde method een gebeurtenis, een fout of exception kan veroorzaken. Als je bijvoorbeeld

reservaties verwerkt voor een voorstelling, dan kan het ingeven van een negatief aantal of te groot aantal tickets zorgen voor een Exception.

Eigen exceptions leid je af van de class RuntimeException of van de class Exception. Het verschil hier is dat een exception, afgeleid van RuntimeException **kan** opgevangen worden, terwijl een exception afgeleid van Exception, **moet** opgevangen worden.

#### 12.4.1 Eigen exception toegepast bij de class Rekening

Bij de bestaande class Rekening wordt een method voorzien om het rekeningnummer te controleren op geldigheid. Daarom eerst even een toelichting over de opbouw en controle van het rekeningnummer.

Het oude Belgische rekeningnummer (Belgian Bank Account Number of BBAN) of nationaal rekeningnummer bestaat uit 12 cijfers, verdeeld in drie groepen gescheiden door liggende streepjes: een groep van 3 cijfers, een groep van 7 cijfers en een groep van 2 cijfers, bijv. 091-0122401-16.

Het IBAN-nummer of International Bank Account Number is een internationale standaard voor het identificeren van bankrekeningnummers. Het IBAN-nummer vervangt het vroegere Belgische rekeningnummer van 12 cijfers. In België bestaat een IBAN-nummer uit 16 tekens: eerst de letters "BE" (code van het land waar de rekening gehouden wordt) gevolgd door 2 cijfers en daarna de 12 cijfers van het nationaal rekeningnummer. Per land kan het aantal tekens variëren. Een Belgisch IBANnummer telt dus altijd 16 tekens (de 2 letters "BE" + 14 cijfers). Het nummer wordt weergegeven in vier groepjes van vier elementen. De groepjes worden telkens door een spatie van elkaar gescheiden.

Een voorbeeld: het Belgische rekeningnummer 539 0075470 34 wordt BE68 5390 0754 7034, waarbij BE68 de bankcode is of ook wel het protocolnummer genoemd.

(Tip: je kan een nationaal rekeningnummer omzetten naar een IBAN-nummer via volgende convertor: <http://www.ibanbic.be/>.)

De controle van het rekeningnummer omvat dus het volgende:

- Eerste twee karakters moeten "BE" zijn
- Derde en vierde karakter moeten cijfers zijn

Dan volgt de controle zoals we die kennen van het nationaal rekeningnummer:

- Neem de volgende 10 cijfers en deel dit getal door 97
- De rest van deze deling moet gelijk zijn aan de laatste 2 cijfers van het rekeningnummer.

Om te kunnen testen volgen hier enkele rekeningnummers:

Correcte rekeningnummers	Foutieve rekeningnummers
BE68 1234 5678 9002	BE25 1112 2444 4891
BE68 1234 5678 9103	AB68 1234 5678 9007
BE68 1234 5678 9204	

Maak een nieuw project aan voor de classes van Rekening die je achteraf hierin zal kopiëren.

Voorzie eerst een eigen exception class. Maak hiervoor package *be.vdab.util*.

```
package be.vdab.util;
public class RekeningNummerException extends Exception {
```

(1)

```
    public RekeningNummerException() {}
    public RekeningNummerException(String omschrijving) {
        super(omschrijving);
    }
}
```

(2)

- (1) De class `RekeningNummerException` is afgeleid van de class `Exception`. Dat betekent dat deze exception moet opgevangen worden.
- (2) Zoals je al eerder in een vorige paragraaf zag, wordt er meestal bij het opvangen van een exception een stukje tekst weergegeven dat verduidelijkt welke soort fout er zich heeft voorgedaan. Dit is meestal de message-property van de exception. Daarom wordt hier de constructor voorzien die een stringparameter aanvaardt. Deze parameter wordt doorgegeven aan de constructor van de moederclass `Exception`.

Voorzie ook volgende packages:

- *be.vdab.rekening* voor de classes `Rekening`, `Zichtrekening` en `Spaarrekening`
- *be.vdab.main* voor het main-programma `Bankbediende`

In de class `Rekening` schrijf je volgende method om het rekeningnummer te controleren:

```
private boolean checkIBANnummer(String reknr) {
```

(1)

```
    //formaat van de string reknr: xxxx xxxx xxxx xxxx
```

```
    if (reknr == null || reknr.isEmpty() ||
        reknr.length() != 19 ||
        !reknr.substring(0,2).equals("BE")) {
        return false;
    }
```

(2)

```
    try {
        Integer.parseInt(reknr.substring(2,4));
        int d1 = Integer.parseInt(reknr.substring(5,9));
        int d2 = Integer.parseInt(reknr.substring(10,14));
        int d3 = Integer.parseInt(reknr.substring(15,17));
        int d4 = Integer.parseInt(reknr.substring(17,19));
```

(3)

```
        long tienCijfers = d1*1000000 + d2*100 + d3;
        int rest = (int)(tienCijfers % 97);
        return (rest == d4);
    }
```

```
    catch (NumberFormatException ex) {
        return false;
    }
}
```

- (1) De method is private d.w.z. dat deze method enkel binnen deze class gebruikt kan worden. Ze retournt een boolean om aan te geven of het rekeningnummer geldig of ongeldig is. Het formaat van het rekeningnummer is een string in de vorm van `xxxx xxxx xxxx xxxx` (met spaties tussen de groepjes).

- (2) De controles op null en een lege string die in de set-method staan, worden nu in deze method geplaatst zodat de volledige controle in deze method is geschreven.
- (3) De twee karakters na "BE" moeten cijfers zijn. Er is verder niets gegeven. Daarom wordt hier enkel geprobeerd om deze karakters om te zetten naar cijfers maar wordt het resultaat niet bewaard in een variabele. Indien dat niet lukt treedt er een NumberFormatException op.

Vervolgens pas je de method `setRekeningNummer` aan die voortaan gebruikt maakt van de method `checkIBANnummer`. In geval van een ongeldig rekeningnummer wordt een `RekeningNummerException` gethrowed:

```
public final void setRekeningNummer(String reknr) throws      (1)
    RekeningNummerException {
    if (checkIBANnummer(reknr)) {                             (2)
        rekeningNummer = reknr;
    }
    else {
        throw new RekeningNummerException("ongeldig IBANreknr"); (3)
    }
}
```

- (1) In de header van de method wordt met `throws` en de naam van de exception aangegeven dat deze method die exception kan throwen.
- (2) Hier wordt de controle method van rekeningnummer opgeroepen.
- (3) In geval van een ongeldig rekeningnummer wordt hier in de setter een exception gethrowed. Er mag dan geen rekeningobject worden aangemaakt omdat dit niet geldig zou zijn. De message van de exception krijgt de waarde van het string-argument.

Dit heeft gevolgen voor de constructors:

```
public Rekening(String rekeningNummer) throws RekeningNummerException { (1)
    setRekeningNummer(rekeningNummer);
}

public Rekening(String rekeningNummer, double saldo)
    throws RekeningNummerException { (1)
    setRekeningNummer(rekeningNummer);
    if (saldo >= 0) {
        this.saldo = saldo;
    }
}
```

- (1) Aangezien de constructors de setter voor rekeningnummer aanroepen, kunnen ook zij een exception veroorzaken. Daarom dient de header `throws` met de naam van de exception te bevatten. Er zal geen ongeldig object worden gemaakt.

Met volgend main-programma kan getest worden:

```
package be.vdab.main;

import be.vdab.util.RekeningNummerException;
import be.vdab.rekening.Rekening;
import be.vdab.rekening.Spaarrekening;
import be.vdab.rekening.Zichtrekening;
```

```
public class BankBediende {
    public static void main(String[] args) {

        Rekening[] rekeningen = new Rekening[7];
        try {
            rekeningen[0] = new Spaarrekening("BE68 1234 5678 9002", 1.5);
            rekeningen[0].storten(100.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        try {
            rekeningen[1] = new Zichtrekening(null, 1500);
            rekeningen[1].storten(200.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        try {
            rekeningen[2] = new Zichtrekening("BE68 1234 5678 9103", 2000);
            rekeningen[2].storten(300.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        try {
            rekeningen[3] = new Zichtrekening("", 1500);
            rekeningen[3].storten(400.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        try {
            rekeningen[4] = new Zichtrekening("AB68 1234", 1500);
            rekeningen[6] = new Zichtrekening(null, 1500);
            rekeningen[4].storten(500.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        try {
            rekeningen[5] = new Zichtrekening("BEZZ 1112 2444 4891", 1500);
            rekeningen[5].storten(600.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        try {
            rekeningen[6] = new Spaarrekening("BE68 1234 5678 9204", 1.5);
            rekeningen[6].storten(400.0);
        }
        catch (RekeningNummerException ex) {
            System.out.println(ex.getMessage() );
        }

        for (Rekening rekening: rekeningen) {
            if (rekening != null) {
                System.out.println("\n" + rekening + "\nSaldo van de " +
                    "rekening: " + rekening.getSaldo());
            }
        }
    }
}
```

```

    }
  }
}

```

Bij de uitvoer kan je zien dat er 4 ongeldige rekeningnummers zijn en dat er 3 geldige rekeningen aangemaakt zijn:

```

ongeldig reknr
ongeldig reknr
ongeldig reknr
ongeldig reknr

```

```

BE68 1234 5678 9002, 100.0, 1.5
Saldo van de rekening: 100.0

```

```

BE68 1234 5678 9103, 300.0, 2000
Saldo van de rekening: 300.0

```

```

BE68 1234 5678 9204, 400.0, 1.5
Saldo van de rekening: 400.0

```

Voor de gebruiker is het nu duidelijk dat er zich een fout heeft voorgedaan bij het rekeningnummer. Hiermee weet de gebruiker echter nog niet over welk rekeningnummer het ging. Aangezien het Rekening-object in het catch-block niet meer bestaat, kan het foutieve rekeningnummer ook opgenomen worden in de RekeningNummerException class. Deze ziet er dan als volgt uit:

```

package be.vdab.util;

public class RekeningNummerException extends Exception {

    private String foutRekeningNummer;                                (1)

    public RekeningNummerException() {}
    public RekeningNummerException(String omschrijving) {
        super(omschrijving);
    }
    public RekeningNummerException(String omschrijving, String      (2)
        foutRekeningNummer) {
        super(omschrijving);
        this.foutRekeningNummer = foutRekeningNummer;
    }

    public String getFoutRekeningNummer() {
        return foutRekeningNummer;
    }
}

```

- (1) Een private membervariabele voor het foutieve rekeningnummer. Hiervoor is ook een getter geschreven.
- (2) De constructor waarbij het foutieve rekeningnummer opgeslaan wordt in de membervariabele.

In de class Rekening wordt het foutieve rekeningnummer ook bewaard in het exception-object. De method `setRekeningNummer` wordt:



```
public final void setRekeningNummer(String reknr) throws
RekeningNummerException {
    if (checkIBANnummer(reknr) ) {
        rekeningNummer = reknr;
    }
    else {
        throw new RekeningNummerException("ongeldig reknr", reknr);
    }
}
```

Dit stelt ons in staat om in het hoofdprogramma *BankBediende* in de catch-blokken niet alleen de message van de exception weer te geven maar ook het rekeningnummer dat de fout veroorzaakte:

```
try {
    rekeningen[4] = new Zichtrekening("AB68 1234", 1500);
    rekeningen[6] = new Zichtrekening(null, 1500);
    rekeningen[4].storten(500.0);
}
catch (RekeningNummerException ex) {
    System.out.println(ex.getMessage() + ": " +
        ex.getFoutRekeningNummer() );
}
```

## 12.5 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 12:

- TestExceptions
- Controle ISBN13nummer

## 13 BigDecimal

### 13.1 Probleemstelling

Ongetwijfeld heb je ooit versted gestaan van het resultaat van een berekening in Java met het gebruik van een float of double.

In het onderstaande voorbeeld wordt 1000 keer 0,01 bij elkaar opgeteld, zowel met floats als met doubles. Je verwacht voor de som als resultaat 10, doch helaas...

```
package jpfhfdst13;
public class JPFhfdst13 {

    public static void main(String[] args) {
        float totaalF = 0.0F;
        for (int i=0 ; i<1000 ; i++) {
            totaalF += 0.01F;
        }
        System.out.println("som met floats: " + totaalF);

        double totaalD = 0.0;
        for (int i=0 ; i<1000 ; i++) {
            totaalD += 0.01;
        }
        System.out.println("som met doubles: " + totaalD);
    }
}
```

Dit bovenstaande programma levert volgende output :

```
som met floats: 10.0001335
som met doubles: 9.999999999999831
```

Het resultaat komt in de buurt van 10, maar heeft niet als resultaat exact 10.0.

Het probleem is dat berekeningen met floating-point getallen snel gebeuren, het zijn primitieve gegevenstypes. Maar het resultaat heeft niet (altijd) de volledige juiste precisie. Er treden afrondingsfouten op, zelfs bij een eenvoudige berekening.

De class `BigDecimal` biedt een oplossing voor dit probleem: deze class kan decimale getallen exact weergeven en kan er exacte berekeningen mee maken. Dit komt omdat de class werkt met meer dan 16 beduidende cijfers. Het betreft dus een decimaal getal zonder afrondingsproblemen.

Gooien we de float en de double dan maar beter overboord? Helemaal niet, deze types zijn ideaal wanneer performance belangrijker is dan precisie. Denk bijvoorbeeld aan een grafische toepassing al dan niet bewegend. Voor financiële of wetenschappelijke berekeningen, waar elk cijfer voor of na de komma belangrijk is, gebruik je beter `BigDecimal`.

## 13.2 De BigDecimal class

### 13.2.1 Creatie van een object

Om de class `BigDecimal` te kunnen gebruiken, dien je volgende import te doen:

```
import java.math.BigDecimal;
```

Een `BigDecimal` object kan je aanmaken via een constructor.

Doch om van een `long` of `double` een object van de class `BigDecimal` te maken, gebruik je beter de static method `valueOf()`. Ondanks er hiervoor constructors bestaan, is dit de aangewezen manier. Enkele voorbeelden:

```
BigDecimal geluksGetal = BigDecimal.valueOf(7);  
BigDecimal grootGetal = BigDecimal.valueOf(7123459L);  
BigDecimal increment = BigDecimal.valueOf(0.01);
```

Om vanuit een string met een decimale waarde een `BigDecimal` object te creëren, gebruik je wel een constructor:

```
String piString = "3.141592653";  
BigDecimal pi = new BigDecimal(piString);
```

Een aantal waarden kan je rechtstreeks gebruiken:

```
BigDecimal.ZERO is de waarde 0  
BigDecimal.ONE is de waarde 1  
BigDecimal.TEN is de waarde 10
```

### 13.2.2 Methods

Eerst en vooral beschikt `BigDecimal` over de methods `add()`, `subtract()`, `divide()` en `multiply()` om de basisbewerkingen uit te voeren. Daarnaast is er een `compareTo()`-method om twee `BigDecimal`s met elkaar te vergelijken. Verder zijn er nog methods zoals `abs()`, `negate()` en `toString()`.

Een woordje uitleg is nodig voor de method `setScale(scale [rounding-mode])`. Hiermee stel je in met hoeveel cijfers na de komma de `BigDecimal` moet werken en de manier waarop er afgerond wordt.

De volgende instructie stelt een `BigDecimal` in met een scale van twee cijfers na de komma en een afronding "HALF\_UP" :

```
BigDecimal eenBigDecimal = BigDecimal.valueOf(2.365);  
eenBigDecimal = eenBigDecimal.setScale(2, RoundingMode.HALF_UP);
```

Bovenstaande `BigDecimal` wordt afgerond naar de dichtstbijzijnde `BigDecimal` die aan de scale voldoet. Wanneer de "afstand" zoals hier naar boven (2.37) of naar beneden (2.36) gelijk is dan kan je met `RoundingMode.HALF_UP` aangeven dat er naar boven moet afgerond worden en met `RoundingMode.HALF_DOWN` dat er naar beneden moet afgerond worden.

`RoundingMode.HALF_UP` levert in dit geval 2.37 op,  
`RoundingMode.HALF_DOWN` levert in dit geval 2.36 op.

Een getal als 2.362 wordt in beide gevallen naar 2.36 afgerond. Immers hier stelt zich het probleem van deze afronding niet. Het getal bevindt zich niet op het “midden” van de afronding naar boven of beneden.

Naast deze twee afrondingsmethodes zijn er nog verschillende andere die je kan terug vinden in de API-documentatie.

### 13.3 Het voorbeeld van de probleemstelling opgelost met BigDecimal

Als we nu het voorbeeld aanpassen door gebruik te maken van de `BigDecimal` class dan krijg je wel het goede resultaat:

```
BigDecimal totaalBD = BigDecimal.ZERO;
BigDecimal incremBD = BigDecimal.valueOf(0.01);

for (int i=0 ; i<1000 ; i++) {
    totaalBD = totaalBD.add(incremBD);
}
System.out.println(totaalBD);
```

De output is exact:

10.00



Naar analogie bestaat er ook een class **BigInteger**.

## 14 Collections

Collections betekent letterlijk verzamelingen. Er bestaan sinds Java 5 een aantal classes die zich gedragen als verzamelingen. Het zijn classes om een aantal objecten in te bewaren. Er bestaan meerdere collections waarbij elke collectie andere kenmerken of eigenschappen heeft, andere methods en een andere manier om de objecten op te slaan, te raadplegen en te manipuleren. Men spreekt over het Collections framework in Java.

Het is lastig om te werken met een verzameling van objecten van een verschillend type. Bijv. één verzameling met hierin objecten van het type Persoon, Voertuig, Cursus en String. Trouwens ieder object heeft andere methods die uitgevoerd kunnen worden. Wanneer je iets met de objecten uit de verzameling wenst te doen, zal je steeds met de instanceof operator dienen na te gaan welk object het is. Want het betreft een verzameling van Objecten. In een verzameling wordt een specifiek object (Persoon, Voertuig, Cursus, String) opgeslaan en wanneer het object wordt opgevraagd is het een algemeen object van type Object. Typecasting is dan nodig om specifieke methods van het object uit te voeren. Daarom wordt er vooral gewerkt met generic collections. Dit zijn collections die enkel één specifiek type van objecten kan bewaren.

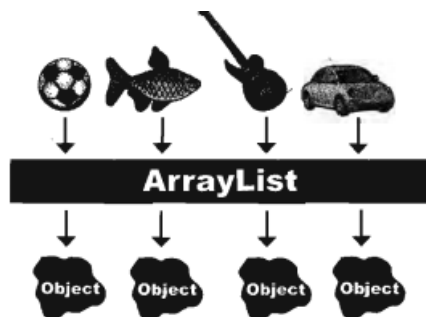
### 14.1 Generics

Omdat het lastig is om te werken met een collection van Objecten, wordt er vooral gewerkt met generic collections. Dit zijn collections die enkel één specifiek type van objecten kunnen bewaren.

Even een voorstelling hoe objecten bewaard worden in een collection:

(afbeelding is gecopieerd uit een bron)

#### Zonder generics:



Objecten worden bewaard in de collectie (ArrayList) als een referentie naar Football, Fish, Guitar en Car. Wanneer ze uit de collectie gehaald worden zijn het zonder meer objecten van type Object. Typecasting naar het juiste type is nodig om te beschikken over de volledige functionaliteit van het object.

#### Met generics:



Enkel objecten van type Fish worden bewaard in de collection (ArrayList<Fish>). Wanneer ze uit de collectie gehaald worden zijn het objecten van type Fish. Typecasting is niet langer nodig.

Het gebruik van generics kenmerkt drie belangrijke zaken voor de programmeur:

- het maken van objecten van generieke classes
- het declareren en definiëren van variabelen met een generiek type
- het schrijven en oproepen van methods met als argument een generiek type

### 14.1.1 Generic classes

De JavaSE bevat reeds generic classes, waaronder het collections framework.

Je kan ook zelf generic classes schrijven, alhoewel je dat in de praktijk niet veel zal doen.

Je zal in dit hoofdstuk kennis maken met het collections framework en generic collections leren kennen. Trouwens de meeste code die generics bevat is collections-gerelateerde code.

## 14.2 Inleiding tot collections

### 14.2.1 Wat is een collection?

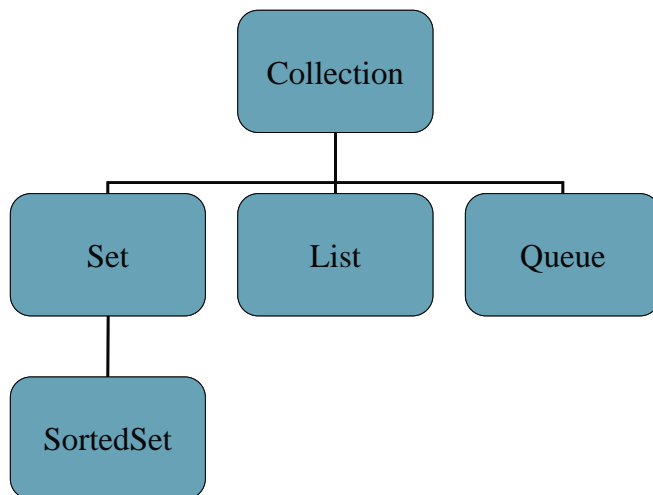
In het hoofdstuk van **Error! Reference source not found.** is het concept besproken. In een array kunnen we een aantal elementen van een primitive type of een aantal objecten van een bepaald type bewaren. Een array is een klassieke en krachtige datastructuur die heel veel wordt gebruikt, maar een array heeft een aantal belangrijke nadelen:

- De grootte van de array moet je van tevoren weten en vastleggen en die grootte kan je daarna niet meer veranderen.
- Een array is geen volwaardig object omdat het geen methods heeft. Daardoor moet je allerlei bewerkingen zoals het toevoegen of verwijderen van een element, of het bijhouden van het aantal elementen dat daadwerkelijk in de array zit, zelf programmeren.

Een alternatief voor een array kan een collection zijn: een verzameling. **Een collection is een object dat een reeks van andere objecten groepeerst.** De collections vormen een framework: het **Collections Framework**. Dit is een framework dat bestaat uit interfaces en bijhorende implementaties zodat er gewerkt kan worden met een reeks (verzameling) van objecten. De ruggengraat van het collections framework wordt gevormd door de **Collection interface**. Deze bevat een reeks van methods die bewerkingen op een collection mogelijk maken, zoals:

- Het **toevoegen** van een object aan de collection.
- Het **verwijderen** van een object uit de collection.
- Het **opvragen** van een element uit de collection.
- Het **itereren** over alle objecten in de collection.

Een overzicht van de **Collection interface**:



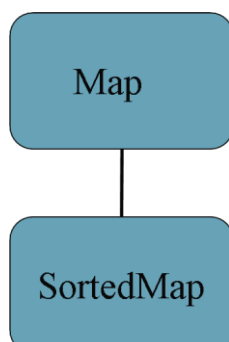
In bovenstaande afbeelding kan je zien dat van de Collection interface nog andere interfaces zijn afgeleid. Een korte toelichting bij deze afgeleide interfaces:

<b>Set</b>	Kan geen dubbele elementen bevatten.
<b>SortedSet</b>	Is een afgeleide interface van Set. Kan geen dubbele elementen bevatten. Is altijd gesorteerd in stijgende volgorde.
<b>List</b>	Is een sequentiële verzameling, dus de volgorde van toevoegen van de elementen blijft behouden. Kan dubbele elementen bevatten. Elk element heeft een integer-positie in de verzameling.
<b>Queue</b>	Is een rij. Elementen worden meestal via het FIFO-principe (First In First Out) verwerkt.

De java-bibliotheek die nodig is voor het gebruik van collections is **java.util.Collection**.

Naast de hiërarchie van Collection bestaat er ook een hiërarchie van Map. Maps zijn niet afgeleid van java.util.Collection maar worden meestal beschouwd als een collection en worden daarom ook bij collections behandeld. De benodigde bibliotheek is **java.util**.

Overzicht van deze **Map interface**:



Ook hier een korte toelichting bij deze verschillende interfaces:

Map	Bevat key-value paren. Kan geen dubbele keys bevatten.
SortedMap	De key-value paren zijn gesorteerd volgens de key. Kan geen dubbele keys bevatten.

Verder in de cursus komen we uiteraard terug op deze interfaces. Ze komen allen aan bod, behalve Queue.

### 14.2.2 Concepten van collections

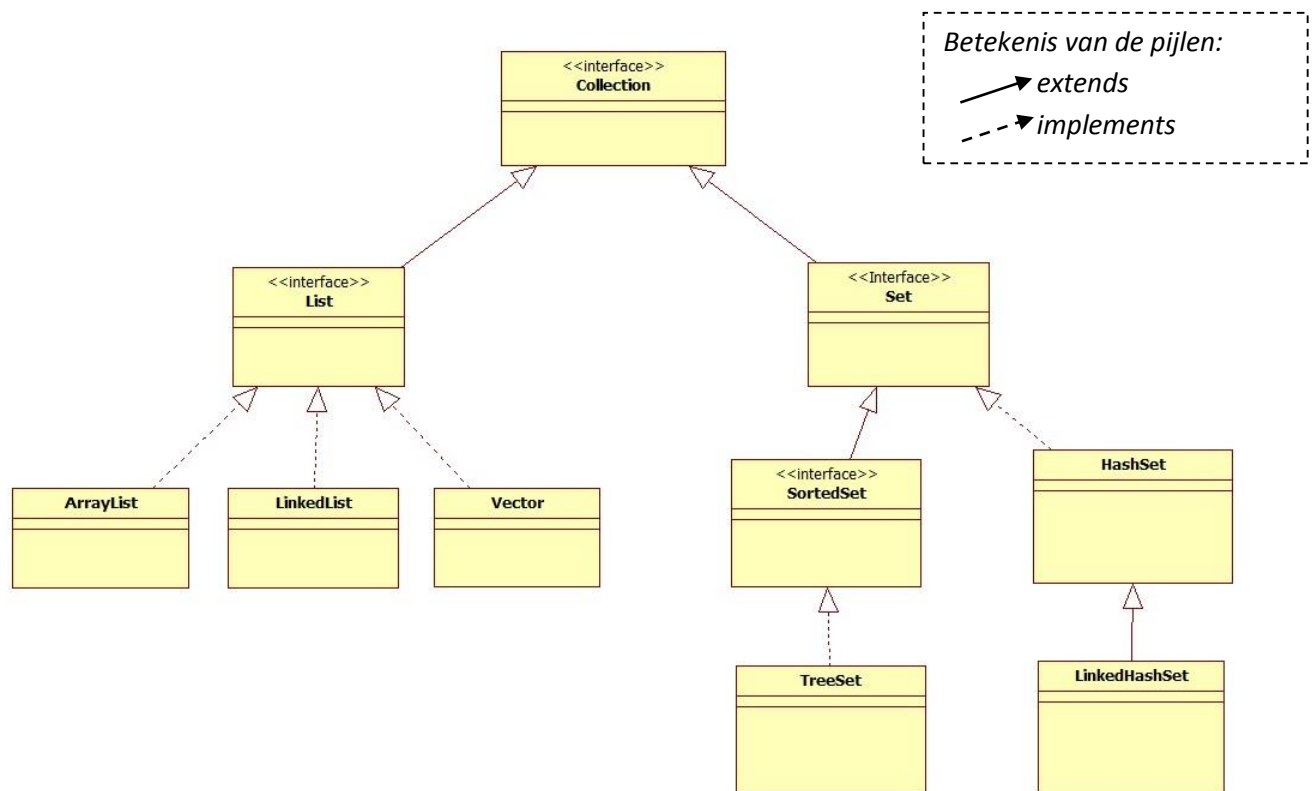
Er worden kort enkele principes van collections opgesomd. Later zullen deze principes bij de uitleg van de interfaces en de voorbeelden toegelicht worden.

- In een collection kunnen enkel **objecten** bewaard worden. Indien we dus een verzameling willen van primitive types (bijv. integers), moeten we deze types verpakken in **wrapper classes**.
- Elke collection heeft haar eigen opslagstructuur. De organisatie van deze opslagstructuur is **intern** aan de collectie.
- Een collection moet doorlopen kunnen worden. Dit kan makkelijk met het gebruik van een for-each-lus. Het is de aangewezen manier.  
Het kan ook via een speciale class, nl. een Iterator. Iterators worden binnen in de collection gedefinieerd, omdat zij moeten vertrouwd zijn met de interne structuur van de collection.

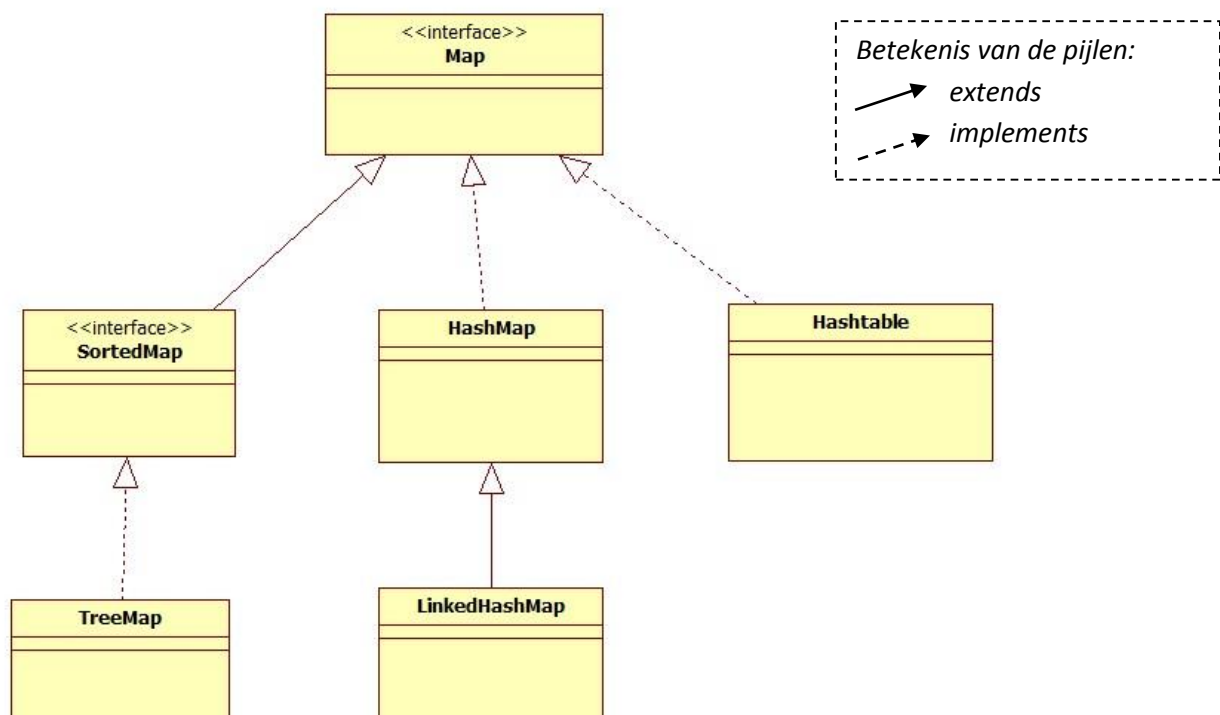
## 14.3 De collection interface

Hieronder volgt het class diagram van de Collection interface:





En van de Map interface:



Het Collections framework biedt, zoals eerder gezegd, vele mogelijkheden. De interface **Collection** beschikt over verschillende methods, o.a. voor het toevoegen van een object aan de collection, het verwijderen van een object uit de collection, het opvragen van een object uit de collection, nagaan of een object aanwezig is in de collection, itereren over alle objecten in de collection, enz.

We sommen er enkele op:

Method	Returnwaarde	Toelichting
<code>add(E e)</code>	boolean	Toevoegen van een element aan de collectie.
<code>remove(Object o)</code>	boolean	Verwijderen van een element uit de collectie, als het aanwezig is.
<code>contains(Object o)</code>	boolean	Opvragen of de collectie het element bevat.
<code>iterator()</code>	<code>Iterator&lt;E&gt;</code>	Opvragen van de iterator om de collectie te kunnen doorlopen.
<code>size()</code>	int	Opvragen van het aantal elementen in de collectie.
<code>clear()</code>	void	Leegmaken van de collectie.
<code>isEmpty()</code>	boolean	Opvragen of de collectie leeg is.

E staat voor het type element dat bewaard wordt in de collectie.

Over de collection kan je itereren met een **iterator**. Die vraag je op met de method `iterator()`. De interface *iterator* bevat 3 methods:

Method	Returnwaarde	Toelichting
<code>hasNext()</code>	boolean	Is er nog een volgend element?
<code>next()</code>	E	Opvragen van het volgende element.
<code>remove()</code>	void	Verwijderen van het element dat het laatst is teruggegeven door <code>next()</code> .



De collectie kan echter beter doorlopen worden met een for-each-lus. Het gebruik van een for-each-lus is veel handiger en zal daarom in de praktijk veel meer gebruikt worden!

Naast de besproken methods bevat de interface Collection nog een reeks andere methods, o.a. methods die een Collection als parameter aanvaarden. Een greep hieruit:

Method	Returnwaarde	Toelichting
<code>addAll(Collection c)</code>	boolean	Voegt alle elementen uit de collection c toe aan de huidige collection.
<code>removeAll(Collection c)</code>	boolean	Verwijdert uit de huidige collection alle elementen die in de collection c zitten.
<code>retainAll(Collection c)</code>	boolean	Verwijdert uit de huidige collection alle elementen die niet in de collection c zitten.

<code>containsAll(Collection c)</code>	boolean	Nagaan of alle elementen uit de collection c in de huidige collection zitten.
<code>toArray()</code>	Object[ ]	Omzetten van de collection naar een array.



Raadpleeg de API-documentatie van de Collection interface voor een volledig overzicht van alle beschikbare methods!

## 14.4 De diamond operator <>

Alvorens in de volgende paragrafen de verschillende interfaces besproken worden, wordt eerst de diamond operator toegelicht.

Het declareren van een generic variabele doe je als volgt:

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

`arrayList` is de generic variabele. Het type is een Collection (nl. `List`) en dat is een class. Daarom gebruik je een constructor om het object te creëren. `List<String>` wil zeggen dat er enkel String objecten in de List collection opgeslaan kunnen worden. `List` is een interface en de concrete collection is `ArrayList`. `ArrayList` heeft bepaalde kenmerken en eigenschappen die verderop besproken worden.

Het is lastig om het generic type bij de constructor te herhalen:

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

Dit is niet nodig door het gebruik van de diamond operator <>. Dat maakt het typen van code minder lastig, is korter en voorkomt vergissingen.



Indien je een `ArrayList` van primitive types wenst, dien je het type aan te geven m.b.v. de wrapper-class.

```
Bijv. List<Integer> arrayList = new ArrayList<>();
```

## 14.5 De List interface

Eerder is reeds gezegd, een List :

- is een sequentiële verzameling, dus de volgorde van toevoegen van de elementen blijft behouden,
- kan dubbele elementen bevatten,
- elk element heeft een integer-positie in de verzameling.

Onder List verstaan we een collectie met een **ordering**: de elementen zitten in een bepaalde volgorde in de lijst. In een geordende collectie hoeven de elementen niet gesorteerd te zijn.

Gevolg hiervan is dat een lijst een eerste en een laatste element heeft. Daarbovenop heeft elk element een nummer. Dat nummer is de index van het element.

In een List is tevens het *null*-element toegestaan.

De List interface voegt een aantal extra methods toe aan die van de Collection interface. Hieronder worden er slechts enkele opgesomd:

Method	Return-waarde	Toelichting
<code>add(int index, E element)</code>	void	Voegt het element in de collection tussen op de positie aangegeven door <i>index</i> .
<code>addAll(int index, Collection c)</code>	boolean	Voegt in de huidige collectie alle elementen van de collectie <i>c</i> tussen, op de positie aangegeven door <i>index</i> .
<code>remove (int index)</code>	E	Verwijdert uit de huidige collection het element op de positie aangegeven door <i>index</i> en dit element wordt teruggegeven. Alle volgende elementen schuiven dus één positie op naar voren.
<code>indexOf(Object o)</code>	int	Geeft de index van het eerste voorkomen van het object <i>o</i> in de collectie, of -1 wanneer de collectie het object niet bevat.
<code>lastIndexOf(Object o)</code>	int	Geeft de index van het laatste voorkomen van het object <i>o</i> in de collectie, of -1 wanneer de collectie het object niet bevat.
<code>get(int index)</code>	E	Geeft het element terug van de opgegeven index.
<code>set(int index, E element)</code>	E	Vervangt het element op de positie opgegeven door <i>index</i> door het opgegeven element. Het oorspronkelijke element van de positie <i>index</i> wordt teruggegeven.
<code>toArray()</code>	Object[ ]	Returnt een array die alle elementen bevat van deze List in de juiste volgorde.



Ook hier geldt: Raadpleeg de API-documentatie van de List interface voor een volledig overzicht van alle beschikbare methods!

### 14.5.1 ArrayList

Een ArrayList is een concrete implementatie van een List. Het is eigenlijk een resizable-array implementatie van de List-interface. Een ArrayList is een class die intern gebruik maakt van een gewone array om de elementen te bewaren. De class ArrayList heeft methods om bewerkingen op de array te doen, zoals het toevoegen of verwijderen van een element. Als de array vol is, maakt

ArrayList automatisch een nieuwe grotere array, en kopieert de elementen uit de oude array naar de nieuwe.

Door deze eigenschappen heeft een ArrayList niet de nadelen van een traditionele array en daarom is een ArrayList in het algemeen makkelijker in het gebruik dan een array.

Verder kan je in een ArrayList dubbels bewaren en tevens het null-element.

#### 14.5.1.1 ArrayList: een voorbeeld

```
package jpfhfdst14;
import java.util.List;
import java.util.ArrayList;
public class VbArrayList {
    public static void main(String[] args) {
        List<String> al = new ArrayList<>();
        al.add("fiets");
        al.add(null);           //null-element toegestaan
        al.add("even");
        al.add("dak");
        al.add("citroen");
        al.add("citroen");     //dubbels toegestaan
        al.add("boom");
        al.add("aap");

        System.out.println("4e element is: " + al.get(3));

        System.out.println("Voorbeeld van een ArrayList:");
        for (String woord : al ) {
            System.out.println(woord);
        }
    }
}
```

- (1) Er wordt een collectie gemaakt van het type ArrayList die enkel String-objecten kan bevatten.
- (2) Er worden verschillende string-objecten toegevoegd aan de arraylist. Ook het null-element en een dubbel wordt toegevoegd.
- (3) Met de get-method wordt een object op een bepaalde positie uit de collectie gehaald. Vergeet niet dat de objecten in een collectie vanaf 0 worden geteld.
- (4) Er wordt geïtereerd over de collectie en alle elementen worden weergegeven. Elk element is een String, dus de for-each-lus itereert over de verzameling strings, vertrekkende van het eerste element tot en met het laatste element.

De uitvoer van deze code is:

```
4e element is: dak
Voorbeeld van een ArrayList:
fiets
null
even
dak
citroen
citroen
boom
aap
```

Merk op dat de volgorde behouden blijft: zoals de elementen worden toegevoegd aan de ArrayList, worden ze er ook weer uitgehaald.

#### 14.5.1.2 Itereren met de iterator

Eén keer wordt er aangetoond hoe de collectie doorlopen kan worden met een iterator. Voeg volgende code toe:

```
System.out.println("\nWeergave m.b.v. iterator");  
for ( Iterator<String> i = al.iterator(); i.hasNext(); ) {           (5)  
    String woord = i.next() ;                                       (6)  
    System.out.println(woord) ;
```

- (5) M.b.v. de method `iterator()` wordt de iterator van de ArrayList opgevraagd. Hiermee kan je itereren over de collection. Je start met het eerste element van de collection en je itereert tot het einde van de collectie. Zolang er elementen in de collection aanwezig zijn (`hasNext()` levert dan true op), wordt de lus doorlopen. Het derde argument van de for-lus blijft leeg. Een ‘verhoging’ van de iterator is niet nodig vermits in de body van de lus de method `next()` wordt uitgevoerd en hiermee wordt de iterator een element verder verplaatst.
- (6) Met de method `next()` wordt het eerstvolgend element uit de collectie opgehaald. Je itereert steeds van het begin tot het einde in de collectie. Vermits je enkel Strings kon opslaan in de collectie, itereer je over strings en retournt de method `next()` een String object.

De uitvoer is natuurlijk hetzelfde als de vorige.

Er bestaan diverse methods om te arraylist te manipuleren. Eentje wordt nog besproken:

```
al.add(3, "test");
```

In de ArrayList wordt op positie 3 het element “test” tussengevoegd. Vermits de elementen in een collection vanaf 0 geteld worden, zal dit nieuwe element “test” het vierde element uit de collection zijn.



In het voorbeeld zijn String-objecten gebruikt voor het vullen van een collectie. Een collectie is een verzameling van objecten, d.w.z. dat er eender welke objecten bewaard kunnen worden in een collectie.

### 14.5.2 LinkedList

Een linkedlist of gelinkte lijst is een lijst die bestaat uit objecten die aan elkaar gekoppeld zijn door middel van referenties. Dit kan zijn in één richting d.m.v. één verwijzing naar een volgend element of in 2 richtingen d.m.v. 2 verwijzingen nl. naar een vorig én een volgend element.

Een linked list heeft als voordeel boven een array dat hij heel eenvoudig op verschillende manieren kan groeien. Aanvankelijk is de lijst leeg. De lijst groeit door er telkens een object aan toe te voegen. Een voorbeeld:



Deze lijst bestaat uit 3 objecten waarin strings zijn bewaard. Elk van deze objecten heet een *entry*. De gegevens in de lijst, in dit geval de strings, zijn de elementen van de lijst. De verbindingen tussen de entry's worden gevormd door referenties met de naam *next*. Het begin van de lijst is aan de linkerkant, en daar wijst een referentie met de naam *header* naar de eerste entry. De referentie *next* van de laatste entry wijst naar *null*, de nulwaarde voor referenties, die (in dit geval) aangeeft dat hier het einde van de lijst is.

Elk object in deze lijst heeft twee attributen: een referentie naar het element (het data-gegeven) dat in de lijst is opgeslagen en een referentie naar de volgende entry (of naar *null*).

Meer uitleg hierover bij de implementatie van LinkedList. Eerst een voorbeeld.

#### 14.5.2.1 LinkedList: een voorbeeld

Het voorbeeld voor de ArrayList werkt ook voor de LinkedList:

```

package jpfhfdst14;
import java.util.LinkedList;
import java.util.List;
import java.util.Iterator;
public class VbLinkedList {

    public static void main(String[] args) {
        List<String> ll = new LinkedList<>();
        ll.add("fiets");
        ll.add(null);           //null-element toegestaan
        ll.add("even");
        ll.add("dak");
        ll.add("citroen");
        ll.add("citroen");     //dubbels toegestaan
        ll.add("boom");
        ll.add("aap");

        System.out.println("4e element is: " + ll.get(3));

        ll.add(3, "test");

        System.out.println("Voorbeeld van een LinkedList:");
        for (String woord : ll) {
            System.out.println(woord);
        }
        System.out.println("\nWeergave m.b.v. iterator");
    }
}
  
```

```

        for ( Iterator<String> i = ll.iterator(); i.hasNext(); ) {
            String woord = i.next() ;
            System.out.println(woord) ;
        }
    }
}

```

Er wordt een collectie gemaakt van het type `LinkedList`. Verder zijn het dezelfde statements als bij de `ArrayList`.

De uitvoer van deze code is:

```

4e element is: dak
Voorbeeld van een LinkedList:
fiets
null
even
test
dak
citroen
citroen
boom
aap

```

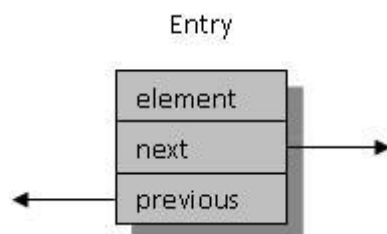
```

Weergave m.b.v. iterator
fiets
null
even
test
dak
citroen
citroen
boom
aap

```

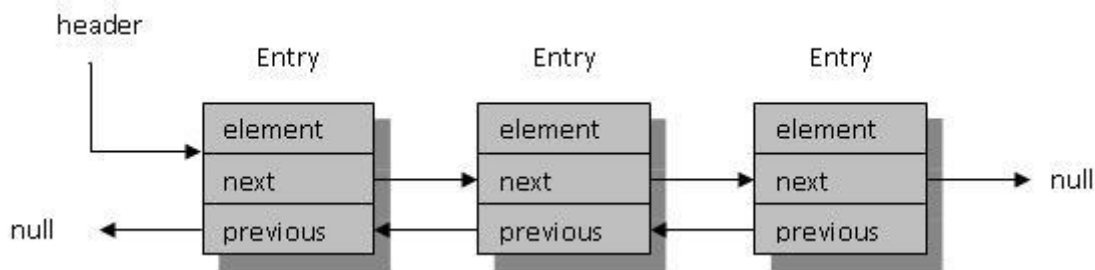
#### 14.5.2.2 Implementatie van `LinkedList`

De implementatie van `LinkedList` bevat een class `Entry` die er uit ziet als volgt:



De entry heeft een referentie *element* die naar het opgeslagen gegeven wijst. En verder niet alleen een referentie *next* naar de volgende entry, maar ook één naar de vorige met de naam *previous*. Een lijst die met dergelijke entry's is opgebouwd heet een doubly linked list of dubbel gelinkte lijst. Een voorbeeld:





Deze implementatie zorgt ervoor dat een LinkedList andere methods heeft dan een ArrayList.

#### 14.5.2.3 Enkele extra methods

Method	Return-waarde	Toelichting
<code>addFirst(E e)</code>	<code>void</code>	Voegt element E tussen, aan het begin van de lijst.
<code>addLast(E e)</code>	<code>void</code>	Voegt element E achteraan toe, aan het einde van de lijst.
<code>getFirst()</code>	<code>E</code>	Returnt het eerste element van de lijst.
<code>getLast()</code>	<code>E</code>	Returnt het laatste element van de lijst.
<code>removeFirst()</code>	<code>E</code>	Returnt en verwijdert het eerste element van de lijst.
<code>removeLast()</code>	<code>E</code>	Returnt en verwijdert het laatste element van de lijst.

Breid het voorbeeld uit met volgende code:

```

System.out.println("\nExtra methods");
LinkedList llijst = (LinkedList) l1;                                (1)
System.out.println(llijst.getFirst());                             (2)
System.out.println(llijst.getLast());                               (3)

llijst.addFirst("eerste");                                          (4)
llijst.addLast("laatste");                                          (5)

System.out.println(llijst.getFirst());                              (6)
System.out.println(llijst.getLast());

System.out.println(llijst.removeFirst());                           (7)
System.out.println(llijst.removeLast());                            (8)

System.out.println(llijst.getFirst());                              (9)
System.out.println(llijst.getLast());

```

- (1) Er gebeurt een typecast van de bestaande list, specifiek naar `LinkedList` omdat methods van de `LinkedList` worden gebruikt.

Het is belangrijk dat de variabele `l1` een referentievareabele is van de interface `List`. Met typecasting kan de variabele `l1` ook verwijzen naar een ander type zolang dit type de interface `List` implementeert.

- (2) Het resultaat van de method `getFirst()` wordt getoond op scherm. De method `getFirst()` geeft het eerste object uit de collectie terug.
- (3) Het resultaat van de method `getLast()` wordt getoond op scherm. De method `getLast()` geeft het laatste object uit de collectie terug.

- (4) De method `addFirst()` voegt een nieuw eerste element toe aan de collectie.
- (5) De method `addLast()` voegt een nieuw laatste element toe aan de collectie.
- (6) Opnieuw wordt het eerste en laatste element uit de collectie getoond. Dit zijn de zopas toegevoegde elementen “eerste” en “laatste”.
- (7) De method `removeFirst()` verwijdert het eerste element uit de collectie. Dit verwijderde element wordt ook teruggegeven door deze method en hier via de `System.out.println()` getoond.
- (8) De method `removeLast()` verwijdert het laatste element uit de collectie. Dit verwijderde element wordt ook teruggegeven door deze method en hier via de `System.out.println()` getoond.
- (9) Opnieuw wordt het eerste en laatste element uit de collectie getoond. Dit zijn opnieuw de oorspronkelijke elementen, vermits de nieuw toegevoegde elementen weer uit de collectie verwijderd zijn.

De uitvoer van deze code is:

```
...
Extra methods
fiets
aap
eerste
laatste
eerste
laatste
fiets
aap
```

### 14.5.3 Verschil tussen ArrayList en LinkedList

Hoewel een ArrayList en een LinkedList allebei een List zijn en dus veel overeenkomsten hebben in het gebruik, zijn er grote verschillen in de implementatie. Een ArrayList maakt gebruik van een array om de referenties naar de gegevens op te slaan. Een LinkedList daarentegen koppelt entry-objecten aan elkaar, en elke entry bevat een referentie naar één van de gegevens.

Uit deze implementatie volgt dat een LinkedList geen random access kent zoals een ArrayList. Bij een LinkedList moet je immers de koppelingen van entry naar entry volgen voor je bij een element met een willekeurige index bent aangekomen. Hoe langer de lijst is, hoe langer dit gemiddeld duurt.

Daar staat tegenover dat, ook dankzij de implementatie, het in een LinkedList erg gemakkelijk is een element ergens in het midden in te voegen of te verwijderen. Het enige dat er moet gebeuren is het aanbrengen en verwijderen van een paar koppelingen tussen entry's.

Voor een ArrayList geldt dat bij het invoegen van een element in het midden de overige elementen een positie moeten opschuiven om ruimte te maken. Bij het verwijderen moeten ze een positie terugschuiven om te voorkomen dat er een gat ontstaat. Naarmate de lijst langer wordt, is dit meer werk en zal dit langer duren.



Het is belangrijk dat je programmeert met interfaces en niet met implementaties!

List lijst = new ArrayList();                    en niet                    ArrayList lijst = new ArrayList();  
List lijst = new LinkedList();                en niet                    LinkedList lijst = new LinkedList();

Op deze manier kan de variabele *lijst* via typecasting ook verwijzen naar een ander type zolang dit type de interface List implementeert

## 14.6 Oefeningen



Zie takenbundel: maak de oefening die hoort bij hoofdstuk 14:  
- Land

## 14.7 De Set interface

Eerder is reeds het volgende gezegd:

- Een Set kan geen dubbele elementen bevatten.
- Een SortedSet is altijd gesorteerd in stijgende volgorde. De SortedSet is een afgeleide interface van Set en kan dus ook geen dubbele elementen bevatten.

Een Set is een datastructuur waarin de elementen uniek zijn. Je kunt dus niet twee dezelfde elementen in een Set opbergen, iets wat in een List wel kan.

Wanneer zijn twee elementen hetzelfde? Het gaat hier om objecten die in een collectie bewaard kunnen worden en dubbels van objecten worden bepaald door de method `equals()`. Wanneer deze `true` teruggeeft, zijn de 2 objecten gelijk.

De twee meest gebruikte implementaties van een Set zijn de HashSet en de LinkedHashSet. Een veel gebruikte implementatie van de SortedSet is de TreeSet.

Vooraleer deze implementaties besproken worden, volgt eerst een korte uitleg over de hashcode en de hashtable.

### 14.7.1 De hashcode

De hashcode is een waarde, een hashwaarde. Dit is een numeriek getal. Om de hashwaarde van een object te bepalen bestaat er een method `hashCode()`. De moeder van alle objecten, class Object, beschikt over deze method en vervolgens beschikken ook alle classes over deze method. Zo heeft de class String deze method geërfd en zo kan je zelf die method overriden in je eigen classes.

Je kan niet rechtstreeks met de hashcode omgaan, ze wordt intern gegenereerd. De hashcode wordt intern gebruikt bij bijv. hashtables, collections zoals de HashSet, enz..

Een voorbeeld:

```
String tekst = "auto";
System.out.println(tekst + ", hashCode: " + tekst.hashCode()); (1)

tekst = "huis";
System.out.println(tekst + ", hashCode: " + tekst.hashCode());
```

(1) De bestaande method `hashCode()` (van de class `String`) wordt gebruikt en berekent de hashcode van de betreffende string. Deze method retournt een `int`.

De uitvoer van deze code is:

```
auto, hashCode: 3005871
huis, hashCode: 3214071
```

De method `hashCode()` is nodig om objecten te kunnen opslaan in verzamelingen die intern werken met een hashcode, o.a. een `Hashtable`, een `HashSet` en een `HashMap`. Wanneer objecten dus opgeslagen dienen te worden in een dergelijke collection, dient de method `hashCode()` voorzien te zijn in de class.

Wanneer je zelf een eigen class schrijft is het aangewezen dat je, net zoals de `equals()` method, ook altijd een `hashCode()` method schrijft. Standaard geeft de `hashCode()` method van class `Object` een `int`-waarde terug.

Zoals je deze methods in je eigen classes erft van `Object`, werkt het mogelijk niet perfect wanneer je de objecten wenst te verzamelen in een `HashSet`. Je kan dit niet volledig vertrouwen. Best bepaal je zelf de berekening van de hashcode, net zoals je zelf bepaalt wanneer 2 objecten aan elkaar gelijk zijn. Telkens wanneer de method uitgevoerd wordt op hetzelfde object, moet steeds dezelfde `int`-waarde gereturnd worden.

Een **vuistregel** om de hashcode te berekenen:

- Gebruik de sleutel van het object, dus de membervariabele die je object uniek identificeert (voor een klant is dat bijv. het klantnummer). Dit kunnen eventueel ook meerdere membervariabelen zijn.
- Is er geen sleutel voor dat object, zet dan de data members die gebruikt worden in de `equals()` method om te testen op gelijkheid, om naar een `String`. Voer op deze string de `hashCode()` method uit.

Je kan dus stellen dat de `equals()` method en de `hashCode()` method samen horen. Een opsomming van de regels in verband hiermee:

Een **`equals()` method moet:**

- **reflexief** zijn: d.w.z. een object is altijd gelijk aan zichzelf.
- **symmetrisch** zijn: `object1 = object2`, dus dan is `object2 = object1`.
- **transitief** zijn: `object1 = object2` en `object2 = object3`, dan is `object1 = object3`.
- **consistent** zijn: bij herhaaldelijk opvragen of `object1 = object2` moet steeds hetzelfde resultaat teruggegeven worden, ervan uitgaande dat er geen waarden van membervariabelen gewijzigd zijn die gebruikt worden bij de vergelijking met `equals()`.

- `object.equals(null)`: moet altijd `false` opleveren.  
Dit argument is toegestaan en geeft geen `nullPointerException`.

#### Over de `hashCode()` method het volgende:

- wanneer `object1 = object2`, dan is de hashcode van de objecten aan elkaar gelijk, maar andersom is dat niet zo, d.w.z. dat wanneer de hashcode van 2 objecten aan elkaar gelijk is, daarom niet de objecten aan elkaar gelijk zijn (maar het kan wel).



Voorzie in je class steeds de methods:

- `equals()`
- `hashCode()`

Vuistregel: baseer deze methods op membervariabelen die niet wijzigen.

### 14.7.2 De hashtable

Een hashtable is een datastructuur waarbij sleutels worden geassocieerd met waarden (hashwaarden) die dan vervolgens gebruikt worden om het element in de tabel te plaatsen of op te zoeken.

Hashing is het proces om een sleutel om te zetten in een hashcode volgens een bepaalde techniek.

De onderliggende werking berust er dus op dat de sleutels worden omgezet in een semi-willekeurig getal, de hashwaarde, in een bepaald bereik!

Als een element in de hashtable moet worden opgezocht, wordt deze hashwaarde gebruikt als index voor een tabel en gekeken of het element op deze plek van de index inderdaad overeenkomt met het element dat opgezocht werd:

- is dit het geval, dan kan de waarde worden teruggegeven.
- is dit niet het geval, dan moet worden nagegaan:
  - ♦ of niet toevallig meerdere sleutels bestaan met de huidige waarde (hashwaarde) waardoor de sleutel niet op de index te vinden is, maar op een alternatieve plek bijv. de volgende index of in een gelinkte lijst (of linkedlist) achter de eerst gevonden index
  - ♦ of dat de gezochte sleutel in zijn geheel niet aanwezig is in de hashtable.

### 14.7.3 Hoe werken de `HashSet` en de `LinkedHashSet`?

Vooraleer de `HashSet` en de `LinkedHashSet` verder bekeken worden, wordt even stil gestaan bij de werking van de `Set`. Dit wordt toegelicht aan de hand van een voorbeeld.

Je beschikt over een class `Coördinaat` met 2 membervariabelen nl. de x- en y-coördinaat, beiden integers. Stel dat je een collection (een `Set`) wil aanleggen van meerdere coördinaten. Belangrijk is de method `hashCode()` die herschreven moet zijn in de class `Coördinaat`. Er is bepaald dat de hashcode wordt berekend als volgt:

- Maak de som van de x- en y-coördinaat.
- Bereken de modulus (%) van de deling (= rest van de deling) van deze som door 7 (7 als deler ligt niet vast, dat kan ook een ander getal zijn, maar het is meestal een priemgetal).

De rest van de deling levert dus een waarde tussen 0 en 6 (grenzen inbegrepen), dus 7 mogelijkheden. Deze rest is de hashcode. De hashcode wordt gebruikt als index van een tussentabel (met dus 7 elementen) voor het bewaren van objecten van de class `Coördinaat` in de collection of het opzoeken ervan.

De collection van het type `Set` wordt `set` genoemd en achtereenvolgens worden een aantal objecten toevoegen aan de collection:

```
1) set.add(new Coördinaat(11,20));
```

Hashcode wordt berekend:  $11 + 20 = 31$   
 $31 \% 7 = 3$

Deze hashcode wordt gebruikt voor de index van een tussentabel, waarin de plaats bewaard wordt van het object in de set:

<i>Tussentabel</i>		<i>Collection set</i>	
0		0	Coördinaat (11,20)      -1
1			
2			
3	0		
4			
5			
6			

Het eerste element in de collection krijgt plaats 0. Dus in de tussentabel wordt bij index 3 (hashcode) de plaats bewaard van dit element. In de collection zelf wordt op plaats 0 het eigenlijke element, het object van `Coördinaat`, geplaatst. Achteraan dit element staat -1. Dit duidt op het einde van de ketting (voorlopig is er slechts één element met hashcode 3).

```
2) Het tweede element wordt toegevoegd aan de set:
```

```
set.add(new Coördinaat(13,20));
```

Hashcode wordt berekend:  $13 + 20 = 33$   
 $33 \% 7 = 5$

Deze hashcode wordt gebruikt voor de index van een tussentabel, waarin de plaats bewaard wordt van het object in de set:

*Tussentabel*

0	
1	
2	
3	0
4	
5	1
6	

*Collection set*

0	Coordinaat(11,20)	-1
1	Coordinaat(13,20)	-1

Het tweede element in de set krijgt plaats 1. Dus in de tussentabel wordt bij index 5 (hashcode) de plaats bewaard van dit element. In de collection zelf wordt op plaats 1 het eigenlijke element, het object van Coordinaat, geplaatst. Achteraan dit element staat opnieuw -1. Dit duidt op het einde van de ketting (voorlopig is er slechts één element met hashcode 5).

- 3) Het derde element wordt toegevoegd aan de set:

```
set.add(new Coordinaat(10,21));
```

Hashcode wordt berekend:  $10 + 21 = 31$

$$31 \% 7 = 3$$

Deze hashcode wordt gebruikt voor de index van een tussentabel, waarin de plaats bewaard wordt van het object in de set: maar de plaats met index 3 in de tussentabel is reeds bezet! Indien het nieuwe element niet gelijk is aan het reeds bewaarde element (wordt bepaald door de `equals()`), wordt de ketting langer gemaakt. Het derde element in de set krijgt plaats 2. Dit wordt aangegeven bij het element op plaats 0 (het element met ook hashcode 3) door de -1 te vervangen door 2, nl. de plaats in de collection en op deze plaats 2 het nieuwe element in de collection te plaatsen:

*Tussentabel*

0	
1	
2	
3	0
4	
5	1
6	

*Collection coll*

0	Coordinaat(11,20)	2
1	Coordinaat(13,20)	-1
2	Coordinaat(10,21)	-1

De ketting wordt dus langer gemaakt indien er meerdere **verschillende** elementen met dezelfde hashcode zijn. Want een Set kan geen dubbele elementen bevatten.

Wanneer er opnieuw een element toegevoegd wordt met dezelfde hashcode (3), dan wordt deze ketting weer verlengd: bij element op plaats 2 wordt achteraan de -1 vervangen door de volgende plaats enz...

Het is dus wenselijk dat de hashcode niet te veel dubbels geeft.

Eerder is reeds gezegd dat wanneer 2 objecten aan elkaar gelijk zijn, ook de hashcode gelijk is, maar andersom geldt dit niet: wanneer de hashcode van 2 objecten gelijk is, zijn daarom de objecten nog niet aan elkaar gelijk.

Dit is zeer duidelijk in dit voorbeeld.

4) Vervolgens wordt een element opgezocht:

```
set.contains(Coordinaat(13, 20));
```

Hashcode wordt berekend:  $13 + 20 = 33$

$33 \% 7 = 5$

Via de tussentabel wordt nu de plaats opgezocht in de Set. Vervolgens wordt m.b.v. de equals() bepaald of de objecten gelijk zijn. Wanneer ze niet gelijk zijn, wordt de ketting gevolgd en wordt vervolgens het object op de volgende plaats vergeleken met het object dat opgezocht wordt in de Set. Zo wordt de ketting afgewerkt. Er wordt gestopt wanneer het op te zoeken object gevonden is of wanneer het einde van de ketting (-1) bereikt is.

Je merkt dus dat het bepalen van de hashcode zijn belang heeft. Eerder is reeds gezegd dat je best de method hashCode() override. Dit kan door een simpel return statement, bijv. `return 1`, maar dat betekent dat de hashcode van elk object gelijk is, dus dat je een zeer lange ketting gaat vormen en dat het opzoeken dus lang gaat duren. Zo verlies je het voordeel van de Set. M.a.w. een return van een vaste waarde is zeker niet aan te bevelen!

**Denk aan de opgesomde vuistregels voor het bepalen van de hashcode!**

#### 14.7.4 De HashSet

Een belangrijke implementatie van de Set is de HashSet:

- Gebruikt intern een hashtable om de elementen op te slaan.
- De volgorde van de elementen is niet bepaald.
- Het *null* element is toegestaan.
- Heeft enkel de methods van de Collection interface. Voegt hier dus geen extra methods aan toe.

In de HashSet gebeurt het toevoegen, verwijderen en opvragen van elementen allemaal even snel. Ongeacht of dit het eerste, middelste of laatste element van de set is. Dit komt omdat de plaats in de collection berekend wordt via een formule die de hashcode gebruikt om de plaats te berekenen.

Een voorbeeld:

```
package jpfhfdst14;
import java.util.HashSet;
import java.util.Set;
```



```

public class VbSet {
    public static void main(String[] args) {
        Set<String> hs = new HashSet<>();
        hs.add("fiets");
        hs.add("even");
        hs.add("dak");
        hs.add("citroen");
        hs.add("boom");
        hs.add("aap");

        System.out.println("Voorbeeld van een HashSet:");
        for (String woord : hs) {
            System.out.println(woord + "\t" + woord.hashCode());
        }
    }
}

```

- (1) Er wordt een collectie gemaakt van het type HashSet.
- (2) Er worden verschillende string-objecten toegevoegd aan de hashset.
- (3) Je itereert over de set. Je itereert steeds van het begin tot het einde over de set. Je start vooraan de set, bij het eerste element en je itereert tot het einde van de set.  
Er zijn geen methods ter beschikking om van achter naar voren te itereren over de set of om je ergens midden in de collectie te positioneren.
- (4) Het element wordt getoond. Daarnaast wordt, gewoon ter info, ook het resultaat van de hashCode() getoond.

De uitvoer van deze code is:

```

Voorbeeld van een HashSet
fiets      97427969
aap        96336
dak        99214
boom       3029739
even       3125530
citroen    785246580

```

Je merkt op dat de volgorde willekeurig is. De volgorde wordt bepaald door de hashcode.

Je probeert nu om het null-element en een dubbel toe te voegen:

```

hs.add(null);
hs.add("dak");

System.out.println("Voorbeeld van een HashSet:");
for (String woord : hs) {
    //System.out.println(woord + "\t" + woord.hashCode());
    System.out.println(woord);
}

```

- (1) Je kan geen hashCode() meer tonen, omdat dat problemen geeft bij het null-element. Vandaar dat de vorige regel tussen commentaar staat.

De uitvoer van deze code is:

```
Set op basis van HashSet
null
fiets
aap
dak
boom
even
citroen
```

Dubbels toevoegen heeft dus geen zin. Het programma loopt niet fout, maar het dubbele element wordt niet bewaard. Zie in dit voorbeeld het string-object *dak*. Deze twee strings zijn gelijk en dus is ook hun hashcode gelijk. Twee objecten met dezelfde hashcode kunnen wel bewaard worden in een hashset, maar niet wanneer deze twee objecten aan elkaar gelijk zijn. En dat is hier het geval.

### 14.7.5 De LinkedHashSet

Nog een implementatie van de Set is de LinkedHashSet:

- Gebruikt intern een combinatie van een hashtable en een linked list (gelinkte lijst) om de elementen op te slaan.
- Behoudt de volgorde waarin de elementen toegevoegd zijn.
- Het *null* element is toegestaan.
- Heeft dezelfde methods als de HashSet: dit zijn dus enkel de methods van de Collection interface. Er zijn geen extra methods toegevoegd aan deze class.

In de LinkedHashSet wordt op gelijke wijze als bij de HashSet de plaats berekend. Maar bij het toevoegen en verwijderen moet de linkedlist worden aangepast en dat kost extra tijd. Het itereren over de lijst gaat sneller omdat dan de linkedlist kan worden gebruikt. Dit gaat sneller dan het berekenen van de plaats via een formule.

Een voorbeeld:

```
package jpfhfdst14;

import java.util.LinkedHashSet;
import java.util.Set;

public class VbLinkedHashSet {

    public static void main(String[] args) {
        Set<String> lhs = new LinkedHashSet<>();
        lhs.add("fiets");
        lhs.add("even");
        lhs.add("dak");
        lhs.add("citroen");
        lhs.add("boom");
        lhs.add("aap");
        lhs.add(null);
        lhs.add("dak");

        System.out.println("Voorbeeld van een LinkedHashSet:");
    }
}
```

(1)

```
for (String woord : lhs) {  
    //System.out.println(woord + "\t" + woord.hashCode());  
    System.out.println(woord);  
}  
}
```

- (1) Er wordt een collectie gemaakt van het type `LinkedHashSet`.  
Verder gebeurt in deze code hetzelfde als er reeds besproken is bij het voorbeeld van de `HashSet`.

De uitvoer van deze code is:

```
Voorbeeld van een LinkedHashSet  
fiets  
even  
dak  
citroen  
boom  
aap  
null
```

Merk op dat de volgorde behouden blijft! De elementen worden weer uit de set gehaald in de volgorde zoals ze zijn toegevoegd. Dit komt door het principe van de linkedlist die gebruikt wordt om de elementen in de set op te slaan.

Ook hier heeft het toevoegen van dubbels geen zin. Het programma loopt niet fout, maar het dubbele element wordt niet bewaard. Zie in dit voorbeeld het string-object *dak*. Het null-element wordt wel bewaard.

### 14.7.6 De TreeSet

Ook in een `TreeSet` bewaar je objecten. Een belangrijke eigenschap van een `TreeSet` is dat de elementen gesorteerd worden opgeslagen. De `TreeSet`:

- Is een implementatie van de `SortedSet`. De elementen uit de set zijn dus op een bepaalde manier gesorteerd.
- Volgorde van de sortering wordt bepaald door de `compareTo()` method van het type object dat bewaard wordt in de collectie. Deze method is een method van de interface `Comparable`. Ze bepaalt de **natural ordering** van de gegevens in de `TreeSet`!
- Het *null* element is **niet** toegestaan.
- Dubbels van objecten worden niet bewaard.

De class `TreeSet` implementeert behalve de interface `Set` ook de interface `SortedSet`. Naast de methods van `Set`, zijn er nog extra methods van `SortedSet` beschikbaar voor een `TreeSet`. Daarbovenop heeft de `TreeSet` ook nog zijn eigen constructors en methods.

Een opsomming van enkele van deze extra methods:

Method	Returnwaarde	Toelichting
<code>first()</code>	E	Geeft het eerste, dus laagste, element terug van de gesorteerde set.
<code>last()</code>	E	Geeft het laatste, dus hoogste, element terug van de gesorteerde set.
<code>headSet(E toElement)</code>	<code>SortedSet&lt;E&gt;</code>	Geeft een view van deze set van alle elementen kleiner dan <code>toElement</code> .
<code>subSet(E fromElement, E toElement)</code>	<code>SortedSet&lt;E&gt;</code>	Geeft een view van deze set van alle elementen groter of gelijk aan <code>fromElement</code> en kleiner dan <code>toElement</code> .
<code>tailSet(E fromElement)</code>	<code>SortedSet&lt;E&gt;</code>	Geeft een view van deze set van alle elementen groter of gelijk aan <code>fromElement</code> .

E staat voor het type element dat bewaard wordt in de collectie.

Voor een volledig overzicht wordt verwezen naar de documentatie.

#### 14.7.6.1 Natural ordening

In de `TreeSet` wordt de volgorde bepaald door de `compareTo()` method, van de interface *Comparable*.

Eerst een voorbeeld:

```
package jpfhfdst14;
import java.util.Set;
import java.util.TreeSet;

public class VbTreeSet {
    public static void main(String[] args) {
        Set<String> ts = new TreeSet<>();
        ts.add("fiets");
        ts.add("even");
        ts.add("dak");
        ts.add("citroen");
        ts.add("boom");
        ts.add("aap");

        System.out.println("Voorbeeld van een TreeSet:");
        for (String woord : ts) {
            //System.out.println(woord + "\t" + woord.hashCode());
            System.out.println(woord);
        }
    }
}
```

(1)

- (1) Er wordt een collectie gemaakt van het type `TreeSet`. Verder gebeurt in deze code hetzelfde zoals er reeds besproken is bij de vorige voorbeelden.

De uitvoer van deze code is:

Voorbeeld van een TreeSet

```
aap
boom
citroen
dak
even
fiets
```

Merk op dat de volgorde alfabetisch is, bovendien in stijgende volgorde. Deze volgorde is beschreven in de `compareTo()` method van de class `String`. De class `String` implementeert de interface `Comparable` en heeft de `compareTo()` method dusdanig overriden dat de strings in alfabetische volgorde en oplopend gesorteerd worden. Dit is de **natural ordering**. Deze wordt dus steeds bepaald in de class zelf door de `compareTo()` method.

Een poging om het null-element en een dubbel toe te voegen:

```
ts.add(null);
ts.add("dak");
```

De uitvoer van deze code geeft een fout:

```
Exception in thread "main" java.lang.NullPointerException
```

Er kan geen null-element toegevoegd worden aan de `TreeSet` omdat dit problemen geeft bij de `compareTo()` method.

Wanneer je de regel `ts.add(null);` in commentaar zet en het programma opnieuw uitvoert stel je vast dat het dubbele element “dak” daarentegen geen problemen geeft, maar ook geen effect heeft: dit wordt namelijk niet bewaard. De string “dak” wordt slechts één keer getoond en is dus ook slechts één keer bewaard in de collectie. Het element wordt enkel aan de set toegevoegd wanneer het nog niet aanwezig is.

Breid de code uit met enkele extra statements:

```
//Extra methods
SortedSet<String> ss = (SortedSet) ts;                                (1)
System.out.println("\nExtra methods:");
System.out.println("Eerste element: " + ss.first());                (2)
System.out.println("Laatste element: " + ss.last());                (3)

SortedSet<String> ssSubSet = ss.subSet("boom", "even");             (4)
System.out.println("\nSubSet van de TreeSet vanaf 'boom' tot
'even':");
for (String woord : ssSubSet) {
    System.out.println(woord);
}
```

- (1) Er gebeurt een typecast van de bestaande set, specifiek naar `SortedSet` omdat methods van de `SortedSet` worden gebruikt.  
Het is belangrijk dat de variabele `ts` een referentievariabele is van de interface `Set`. Met typecasting kan de variabele `ts` ook verwijzen naar een ander type zolang dit type de interface `Set` implementeert.

- (2) De method `first()` geeft het eerste of laagste element van de `treeSet`.
- (3) De method `last()` geeft het laatste of hoogste element van de `treeSet`.
- (4) Vervolgens wordt een nieuwe `sortedset` via de method `subSet( elementVanaf, elementTot )` gecreëerd: deze haalt een gedeelte uit de `TreeSet`, nl. alle elementen die groter of gelijk zijn aan `elementVanaf` en kleiner zijn dan `elementTot`.  
In dit voorbeeld zal een nieuwe `sortedset` gecreëerd worden met alle elementen vanaf 'boom' tot 'even'.

De uitvoer van dit extra stuk code is :

Extra methods:

Eerste element: aap

Laatste element: fiets

SubSet van de `TreeSet` vanaf 'boom' tot 'even':

boom

citroen

dak

#### 14.7.6.2 Natural ordering – `compareTo()`

De natural ordering wordt bepaald door de `compareTo()` method. In het vorige voorbeeld is deze sortering beschreven in de class `String`, nl. oplopend alfabetisch. Dat is een gegeven.

Aan de hand van volgend voorbeeld wordt aangetoond hoe je zelf de volgorde kan bepalen van objecten van je eigen class die bewaard worden in een `TreeSet`.

Maak een package `be.vdab.cursus`. Voorzie hierin de class `Cursus`:

```
package be.vdab.cursus;
```

```
public class Cursus implements Comparable<Cursus> {                                (1)
    private int cursusNr;
    private String cursusNaam;
    private int prijs;

    public Cursus(int nr, String naam, int prijs) {
        setCursusNr(nr);
        setCursusNaam(naam);
        setPrijs(prijs);
    }

    public int getCursusNr() {
        return cursusNr;
    }

    public final void setCursusNr (int nr) {
        if (nr > 0)
            this.cursusNr = nr;
    }

    public String getCursusNaam() {
        return cursusNaam;
    }

    public final void setCursusNaam (String naam) {
        if ((naam!=null) && !(naam.isEmpty()))
            this.cursusNaam = naam;
    }
}
```

```

    }

    public int getPrijs() {
        return prijs;
    }
    public final void setPrijs (int prijs) {
        if (prijs > 0)
            this.prijs = prijs;
    }

    @Override
    public String toString() {
        return (cursusNr + " ; " + cursusNaam + " ; " + prijs);
    }

    @Override
    public boolean equals (Object o) {
        if (o == null) {
            return false;
        }
        if (!(o instanceof Cursus)) {
            return false;
        }
        Cursus c = (Cursus) o;
        return cursusNr == c.getCursusNr();
    }

    @Override
    public int hashCode() {
        return cursusNr;
    }

    @Override
    public int compareTo(Cursus c) {
        // sorteren op cursusnr
        if (c == null) {
            throw new NullPointerException();
        }
        else {
            return cursusNr - c.getCursusNr();
        }
    }
}

```

(2)

(3)

(4)

- (1) De class Cursus implementeert de generic interface *Comparable<Cursus>*. Dat betekent dat de *compareTo()* method overriden moet worden. Het betreft de generic method van *compareTo()*, d.w.z. dat de *compareTo()* method een object van type Cursus als argument heeft: *compareTo(Cursus c)*.
- (2) De *equals()* method beschrijft dat 2 cursusobjecten aan elkaar gelijk zijn wanneer hun cursusnummers aan elkaar gelijk zijn.
- (3) Het is aanbevolen om naast de *equals()* method ook altijd de *hashCode()* method te voorzien. Het is hier eenvoudig gehouden. Vermits het cursusnummer uniek is en ook gebruikt wordt bij de *equals()* method, is dit de returnwaarde van de *hashCode()* method.
- (4) Dit is de method die de sortering bepaalt. Hier wordt de sortering of vergelijking geschreven van 2 cursus-objecten. Indien het vergelijkend object een null-object is, wordt er een *NullPointerException* gethrowed.

De method geeft een integer terug: 0 bij gelijkheid, een negatief getal als het eerste object 'kleiner' is dan het tweede en een positief getal als het eerste object 'groter' is dan het tweede.

Wanneer is een cursusobject gelijk, kleiner of groter aan een ander cursusobject?

Hier worden de cursusnummers met elkaar vergeleken door het verschil van de cursusnummers terug te geven: een cursus-object is kleiner dan een ander cursusobject wanneer zijn cursusr kleiner is (het verschil is dan negatief). Het is groter dan een ander cursusobject wanneer zijn cursusr groter is (het verschil is dan positief) en de twee cursus-objecten zijn gelijk wanneer de cursusnummers aan elkaar gelijk zijn (het verschil is dan 0).

Deze vergelijking moet **consistent** zijn met de `equals()` method. Dat is belangrijk! We komen daar nog op terug.

De regel code van het return statement kan eventueel ook als volgt geschreven worden, maar dat is wel langer:

```
if (cursusNr < c.getCursusNr() )
    return -1;
else
    if (cursusNr > c.getCursusNr() )
        return 1;
    else
        return 0;
```

Hier wordt steeds -1, 0 of 1 gereturned.

Schrijf hierbij nu het volgende main-programma:

```
package be.vdab;
import be.vdab.cursus.Cursus;
import java.util.Set;
import java.util.TreeSet;

public class CursusMain {
    public static void main(String[] args) {

        Set<Cursus> cursussen = new TreeSet<>();
        cursussen.add(new Cursus(5, "Word", 100) );
        cursussen.add(new Cursus(3, "Excel", 110) );
        cursussen.add(new Cursus(1, "Windows", 90) );
        cursussen.add(new Cursus(4, "Access", 120) );
        cursussen.add(new Cursus(2, "Powerpoint", 80) );

        for (Cursus cursus : cursussen) {
            System.out.println(cursus);
        }
    }
}
```

- (1) Er wordt een collectie gemaakt van een TreeSet.
- (2) Er worden 5 objecten van de class Cursus toegevoegd aan de set.



- (3) De set wordt overlopen en elk cursusobject wordt weergegeven.

De uitvoer van deze code is:

```
1 ; Windows ; 90
2 ; Powerpoint ; 80
3 ; Excel ; 110
4 ; Access ; 120
5 ; Word ; 100
```

Je zal vaststellen dat de volgorde anders is dan bij het opvullen van de collectie. De volgorde is hier bepaald door het cursusnr. De cursussen worden getoond in volgorde van hun cursusnr, zoals beschreven in de `compareTo()` method.

Indien je de cursussen in dalende volgorde wenst te ordenen, dien je het return statement in de `compareTo()` method aan te passen als volgt:

```
return c.getCursusNr() - cursusNr;
```

Normaal gezien geeft de method een integer terug: 0 bij gelijkheid, een negatief getal als het eerste object 'kleiner' is dan het tweede en een positief getal als het eerste object 'groter' is dan het tweede. Dit wordt dan omgekeerd: er wordt een positief getal teruggegeven als het eerste object 'kleiner' is dan het tweede en een negatief getal als het eerste object 'groter' is dan het tweede.

Zo kan je de volgorde ook aanpassen op cursusnaam of op prijs, beide stijgend of dalend. Doch hou er steeds rekening mee dat de natural ordening (`compareTo()` method) consistent is met `equals()`!

#### 14.7.6.3 Consistentie van `equals()` en `compareTo()`

De documentatie van de JDK schrijft voor dat volgende vergelijking best opgaat:

```
(x.compareTo(y) == 0) == (x.equals(y))
```

Het komt er dus op neer dat sterk wordt aanbevolen om:

- óf het resultaat van `compareTo()` in overeenstemming te brengen met dat van `equals()`;
- óf duidelijk aan te geven dat die overeenstemming er niet is.

Wanneer die overeenstemming er niet is, hoeft dit geen probleem op te leveren, maar het is op zijn minst onduidelijk als er in een programma 2 definities van "is gelijk aan" in omloop zijn. Als je wilt dat een `SortedSet` zijn werk goed doet, ben je verplicht ervoor te zorgen dat de werking van `compareTo()` voor gelijke objecten dezelfde is als de werking van `equals()`.

Dit wordt aangetoond met de class `Cursus`. De natural ordening van cursussen wordt gewijzigd naar een oplopende volgorde van cursusprijs. Wijzig in de class `Cursus` de `compareTo()` method als volgt (plaats eventueel de huidige code in commentaar):

```
@Override
public int compareTo(Cursus c) {
    // sorteren op cursusPrijs: consistent met equals()
    if (c == null)
```

```

        throw new NullPointerException();
    else {
        if ( this.equals(c) )
            return 0;
        else
            return prijs == c.getPrijs()? -1 : prijs - c.getPrijs();
    }
}

```

- (1) In geval van gelijke cursusobjecten wordt 0 teruggegeven. In een set kunnen immers geen dubbels zitten! Dit is de consistentie met equals().
- (2) Wanneer het om twee verschillende cursusobjecten gaat, wordt het verschil van de cursusprijs teruggegeven. Een cursusobject is kleiner dan een ander cursusobject wanneer zijn prijs kleiner is (het verschil is dan negatief). Het is groter dan een ander cursusobject wanneer zijn prijs groter is (het verschil is dan positief). Wanneer de cursusprijs aan elkaar gelijk is (het verschil is dan 0) moet de cursus toch bewaard kunnen worden in de set. Daarom mag er geen 0 teruggegeven worden, maar wel een positief of negatief getal. Hier wordt in geval van gelijke prijs -1 teruggegeven zodat deze cursus op de juiste plaats wordt tussengevoegd.

Breid het main()-programma uit met volgende regel:

```
cursussen.add(new Cursus(6, "Photoshop", 100) );
```

Bedoeling is om een cursus toe te voegen met dezelfde prijs als een andere cursus maar met een verschillend cursusrnr.

De uitvoer van het main-programma is:

```

2 ; Powerpoint ; 80
1 ; Windows ; 90
6 ; Photoshop ; 100
5 ; Word ; 100
3 ; Excel ; 110
4 ; Access ; 120

```

Je ziet dat de volgorde opnieuw anders is dan bij het opvullen van de collectie. De volgorde wordt hier bepaald door de prijs. De cursussen worden getoond in oplopende volgorde van hun prijs, zoals beschreven in de `compareTo()` method.

#### 14.7.6.4 Meerdere sorteringen

Je kan in je class slechts één volgorde bepalen in de `compareTo()` method. Hier is de `compareTo()` method steeds gewijzigd. De volgorde die je hier bepaalt, is de natural ordening van de objecten van deze class, die gebruikt wordt bij de `TreeSet`. Vaak zal deze ordening zeer geschikt zijn, doch af en toe kan het voorkomen dat je de objecten volgens een ander criterium wenst te ordenen. Dat is mogelijk met de **Comparator**. Je kan dus de elementen in een collection sorteren volgens een aparte comparator class (zie API-documentatie). Het gebruik van een comparator verhoogt daardoor het aantal sorteermethoden gevoelig.

## 14.8 Oefeningen



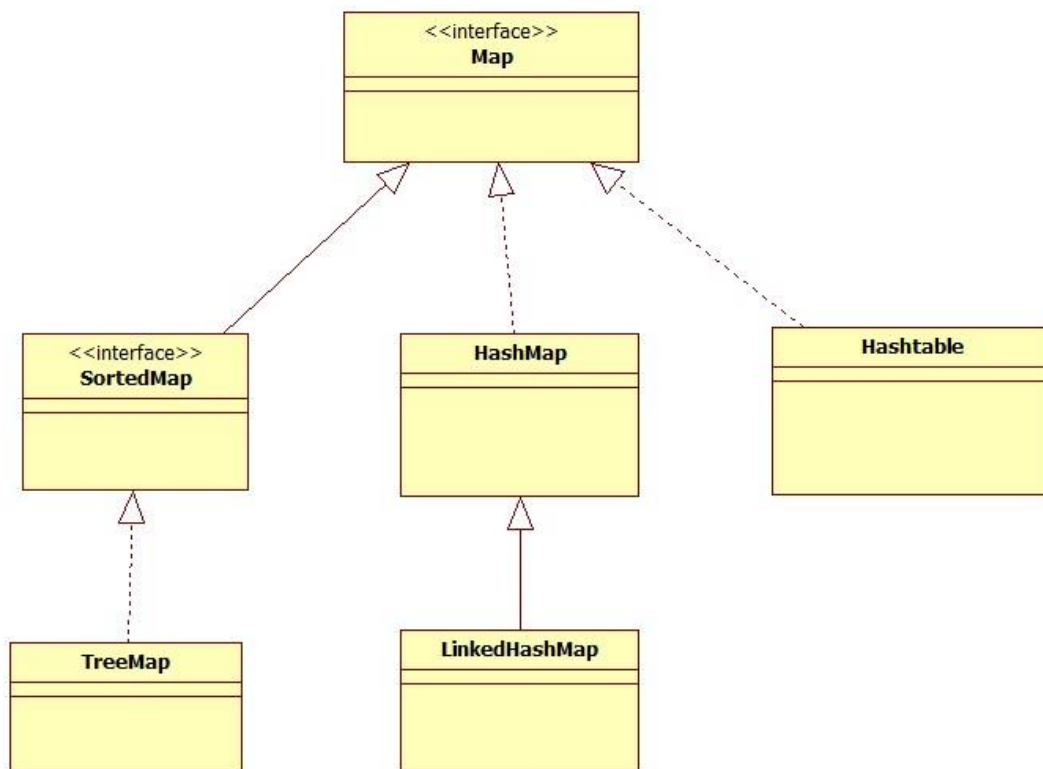
Zie takenbundel: maak oefeningen die horen bij hoofdstuk 14:

- VoertuigenC
- Tienkamper

## 14.9 De Map Interface

Naast de Collection Interface bestaat er ook de Map Interface. Maps zijn niet afgeleid van *java.util.Collection* maar worden meestal beschouwd als een collection en worden daarom ook bij collections behandeld.

Het class-diagram van de Map interface:



De classes HashMap en TreeMap implementeren beide de interface Map. De laatste class implementeert bovendien de interface SortedMap (die een uitbreiding is van Map). In een TreeMap worden de elementen gesorteerd opgeslaan.

Het volgende is reeds vermeld :

Map...	...bevat key-value paren. ...kan geen dubbele keys bevatten.
--------	---

Bij de `ArrayList` kan je een element opzoeken aan de hand van zijn index (m.b.v. de method `get(int index)`). Nadeel hiervan is dat je alleen via een geheel getal (de index) het element kan opzoeken. Het zou mooi zijn als je ook zou kunnen zoeken m.b.v. andere zaken dan een geheel getal. Dat is mogelijk bij de `Map`.

### 14.9.1 Key-value paren

Een `Map` is een verzameling van key-value paren. Wat zijn dat dan?

Een key-value paar is een paar dat bestaat uit 2 gegevens die aangeduid worden met de woorden **key (sleutel)** en **value (waarde)**.

Een voorbeeld van key-value paren die bepaalde cursussen beschrijven:

```
(5, "Word")
(3, "Excel")
(1, "Windows")
(4, "Access")
(2, "Powerpoint")
```

De key, het eerste deel van het paar, is het nummer van de cursus en de value, het tweede deel van het paar, is de cursusnaam. Een cursusnummer is een uniek nummer.

In de praktijk komt het vaak voor dat de key bij voorkeur niet een geheel getal is, maar veel liever een string of nog iets anders. Zo kan je de cursusnummers vervangen door een cursuscode en dat kan dan bijvoorbeeld het volgende zijn:

```
("DOC", "Word")
("XLS", "Excel")
("WIN", "Windows")
("MDB", "Access")
("PPT", "Powerpoint")
```

De keys zijn nu strings, en bij elke key hoort precies één waarde.

M.b.v. de diamond operator die je gebruikt voor generics geef je het type aan van de key en de value. Bijv. `Map<Integer,String>` stelt een map voor waarbij de key een integer is en de value een `String`. `Map<String,String>` stelt een map voor waarbij zowel de key als de value van het type `String` zijn.

### 14.9.2 Interface Map

Eigenlijk wordt er niet gesproken over een interface `Map` maar over een interface `Map<K, V>`, waarbij

$K$  – het type is van de keys  
 $V$  – het type is van de values

Het is een object dat keys mapt naar waardes. Een map kan geen dubbele keys bevatten en een key verwijst slechts naar één waarde.

Hieronder volgen enkele methods, doch raadpleeg de documentatie voor een volledig overzicht van alle methods:

Method	Returnwaarde	Toelichting
<code>put (K, V)</code>	V	Voegt het key-value paar aan de map toe. Deze method zorgt er voor dat alle sleutels in de map uniek zijn. Indien de key reeds bestaat, zal de bijbehorende value in de map vervangen worden door de nieuwe value. De oude value is dan de returnwaarde van de method. Indien de key niet in de map voorkomt, wordt de waarde <i>null</i> teruggegeven.
<code>get (Object K)</code>	V	Geeft de waarde terug die overeenkomt met de key of <i>null</i> indien de key niet aanwezig is in de map.
<code>remove (Object K)</code>	V	Verwijdert de mapping voor de key indien deze aanwezig is in de map. De value is de returnwaarde van de method. Indien de key niet in de map voorkomt, wordt de waarde <i>null</i> teruggegeven.
<code>containsKey (Object key)</code>	boolean	Geeft <i>true</i> terug wanneer de map een mapping bevat van de gevraagde sleutel.
<code>containsValue (Object value)</code>	boolean	Geeft <i>true</i> terug wanneer de map één of meerdere keys mapt naar de gevraagde value.
<code>size ()</code>	int	Geeft het aantal mappings van de map terug.
<code>keySet ()</code>	Set<K>	Geeft een set terug van alle keys in de map.
<code>values ()</code>	Collection<V>	Geeft een collectie terug van alle waarden in de map.
<code>entrySet ()</code>	Set<Map<K,V>>	Geeft een set terug van de mappings in de map.
<code>clear ()</code>	void	Alle mappings worden verwijderd uit de map.

De belangrijkste implementaties van de Map zijn de HashMap, de LinkedHashMap en de TreeMap.

### 14.9.3 De HashMap

De HashMap heeft naast de methods waarover het beschikt omdat het de interface Map implementeert, vier constructors en enkele extra methods. Met de default constructor kan je een lege map maken, maar er bestaat ook een constructor die een map maakt gebaseerd op een reeds bestaande map. Voor meer details aangaande de constructors en de extra methods verwijzen we naar de documentatie.

Hoe werkt een HashMap eigenlijk? De HashMap bergt elk key-value paar op in een object van de class Entry, een inwendige class van HashMap. De Entry-objecten komen in de HashMap in een array, een zogenaamde hashtable. De plaats van deze Entry-objecten in de hashtable wordt berekend op grond van de informatie in de key: nl. aan de hand van de hashcode ervan. Vandaar dat de volgorde van toevoegen niet behouden blijft. Dit principe is reeds eerder besproken bij de HashSet.

#### 14.9.3.1 Collection views van de HashMap

Een HashMap heeft geen iterator, maar via 3 zogeheten collection views kun je toch iterator-achtige bewerkingen op de elementen van een HashMap uitvoeren.

Volgende 3 views zijn mogelijk :

- Een view van de keys van de map: deze krijg je via de method `keySet()`
- Een view van de values van de map: deze krijg je via de method `values()`;
- Een view van de key-value-paren van de map: deze krijg je via de method `entrySet()`;

De terugkeerwaardes van deze methods zijn een Set of een Collection waarover vervolgens geïtereerd kan worden via de for-each-loop of via een iterator.

Het woord view doet vermoeden dat je de elementen alleen kunt bekijken, maar dat is niet het geval. Via een view kun je ook in beperkte mate wijzigingen aanbrengen in de onderliggende collectie. Je bent dan beperkt tot de methods van de iterator om wijzigingen aan te brengen. Een `remove()` is o.m. mogelijk.

Voorbeeld van een HashMap:

```
package jpfhfdst14;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
public class VbHashMap {
    public static void main(String[] args) {
        Map<String, String> landen = new HashMap<>();           (1)
        landen.put("B", "Belgie");                             (2)
        landen.put("NL", "Nederland");
        landen.put("F", "Frankrijk");
        landen.put("D", "Duitsland");
        landen.put("L", "Luxemburg");

        String eenLand = landen.get("F");                      (3)
        System.out.println("Land met code F: " + eenLand);

        String vorigLand = landen.put("F", "Finland");         (4)
```

```

System.out.println("Vorig land met code F: " + vorigLand);
eenLand = landen.get("F");
System.out.println("Land met code F: " + eenLand);

System.out.println("\n*** View van de Keys ***");
for (String eenLandcode : landen.keySet()) {
    System.out.println(eenLandcode);
}

System.out.println("\n*** View van de Keys met bijhorende value-
waarde ***");
for (String eenLandcode : landen.keySet()) {
    System.out.println(eenLandcode + " heeft als landnaam: " +
landen.get(eenLandcode));
}

System.out.println("\n*** View van de Values ***");
for (String eenLandnaam : landen.values()) {
    System.out.println(eenLandnaam);
}

System.out.println("\n*** View van de Key-Value-paren ***");
for (Entry eenLandEntry : landen.entrySet()) {
    System.out.println(eenLandEntry);
}
}
}

```

- (1) Er wordt een map gemaakt van het type HashMap. Zowel de key als de value zijn van het type String. Bedoeling is om een aantal landen te bewaren in de map met als key een landcode en voor de value een landnaam.
- (2) Vervolgens worden er een aantal mappings bewaard in de HashMap. De terugkeerwaarde van de method `put()` wordt niet bewaard. Vermits het hier nieuwe mappings zijn, is de key nog niet aanwezig in de map en zal de method steeds *null* teruggeven.
- (3) Met de method `get(Object key)` wordt een key opgezocht in de map en wordt de overeenkomstige value teruggegeven. Hier in het voorbeeld wordt de landnaam opgezocht van de landcode F.
- (4) Er wordt opnieuw een mapping toegevoegd met key "F", maar nu met value "Finland". Vermits key "F" reeds aanwezig is in de map, wordt de vorige value "Frankrijk" overschreven door "Finland". De oude value "Frankrijk" is de terugkeerwaarde van de `put()`.  
Het is de method `put()` die er voor zorgt dat de keys uniek blijven in de map!
- (5) Een eerste view wordt gemaakt. Via de method `keySet()` bekom je een set van alle keys aanwezig in de map. Dit is een set van Strings. Met de for-each-lus wordt hierover geïtereerd en worden alle keys (zijnde landcodes) weergegeven.

- (6) Per aanwezige key in de set wordt met de method `get()` de bijbehorende value opgezocht zodat je de volledige inhoud van de map kan tonen.
- (7) Een tweede view wordt gemaakt. Via de method `values()` bekom je een Collection van alle aanwezige values in de map. Met de for-each-lus wordt hierover geïtereerd en worden alle values (zijnde landnamen) weergegeven.
- (8) Tot slot wordt ook de derde view gemaakt. Via de method `entrySet()` bekom je een set van alle entry-objecten in de map. Met de for-each-lus wordt hierover geïtereerd en worden alle entries (key-value paren) weergegeven.

De uitvoer van deze code is:

```
Land met code F: Frankrijk
Vorig land met code F: Frankrijk
Land met code F: Finland

*** View van de Keys ***
D
F
NL
B
L

*** View van de Keys met bijhorende value-waarde ***
D heeft als landnaam: Duitsland
F heeft als landnaam: Finland
NL heeft als landnaam: Nederland
B heeft als landnaam: België
L heeft als landnaam: Luxemburg

*** View van de Values ***
Duitsland
Finland
Nederland
Belgie
Luxemburg

*** View van de Key-Value-paren ***
D=Duitsland
F=Finland
NL=Nederland
B=Belgie
L=Luxemburg
```

Je merkt wellicht op dat de volgorde anders is dan deze van het toevoegen aan de map. Zoals reeds eerder is vermeld, wordt de volgorde bepaald door de hashcode van de key: het is immers een hashmap.



Ook *null* is toegestaan, zowel voor de key als voor de value, doch dit is weinig zinvol.



Indien je een `HashMap` wenst waarvan de key en/of value een primitieve type is, dien je het type aan te geven m.b.v. de wrapper-class.

Bijv. `Map<Integer, String> hashMap = new HashMap<>();`

#### 14.9.4 De LinkedHashMap

De class `LinkedHashMap` is een subclass van `HashMap`. Daardoor beschikt `LinkedHashMap` over alle functionaliteit en over de efficiëntie van `HashMap`.

Het grote verschil met `HashMap` is dat de entries die in de tabel worden opgeborgen onderling ook nog verbonden zijn door 2 referenties, één naar de vorige en één naar de volgende entry. Ze vormen dus een dubbel gelinkte lijst.

Deze gelinkte lijst wordt default opgebouwd in de volgorde waarin je de entries aan de `HashMap` toevoegt. Dit wordt **insertion-order** genoemd.

De `LinkedHashMap` heeft een extra constructor en enkele extra methods.

Voorbeeld van een `LinkedHashMap`: wanneer je in het vorige voorbeeld geen `HashMap` creëert, maar een `LinkedHashMap`, zal je merken dat dit zonder meer werkt:

```
package jpfhfdst14;
import java.util.LinkedHashMap;
import java.util.Map;
public class VbLinkedHashMap {
    public static void main(String[] args) {
        Map<String, String> landen = new LinkedHashMap<>();
        landen.put("B", "Belgie");
        ...
    }
}
```

(1)

- (1) Er wordt een map gemaakt van het type `LinkedHashMap`. Zowel de key als de value zijn van het type `String`.

Bij de uitvoer stel je vast dat de volgorde van toevoegen behouden is:

```
Land met code F: Frankrijk
Vorig land met code F: Frankrijk
Land met code F: Finland
```

```
*** View van de Keys ***
B
NL
F
D
L
```

```
*** View van de Keys met bijhorende value-waarde ***
```

```
B heeft als landnaam: Belgie  
NL heeft als landnaam: Nederland  
F heeft als landnaam: Finland  
D heeft als landnaam: Duitsland  
L heeft als landnaam: Luxemburg
```

```
*** View van de Values ***
```

```
Belgie  
Nederland  
Finland  
Duitsland  
Luxemburg
```

```
*** View van de Key-Value-paren ***
```

```
B=Belgie  
NL=Nederland  
F=Finland  
D=Duitsland  
L=Luxemburg
```

### 14.9.5 De TreeMap

Ook in een TreeMap sla je key-value-paren op. Een belangrijke eigenschap van een TreeMap is dat de elementen **gesorteerd op key** worden opgeslagen.

- De volgorde van de sortering wordt bepaald door de `compareTo()` method van de Comparable interface. Dit is dus een voorwaarde voor het gebruik van de TreeMap: het type van de key (van het key-value-paar) moet een type zijn dat de interface Comparable implementeert en dus beschikt over een method `compareTo()`.
- Het *null* element is **niet** toegestaan.

De class TreeMap implementeert behalve de interface Map ook de interface SortedMap. Naast de methods van Map, zijn er nog extra methods van SortedMap beschikbaar voor een TreeMap. Daarbovenop heeft de TreeMap ook nog zijn eigen constructors en methods.

We bespeken enkele van deze extra methods:

Method	Returnwaarde	Toelichting
<code>firstKey()</code>	K	Geeft de eerste, dus laagste, key terug van deze gesorteerde map.
<code>lastKey()</code>	K	Geeft de laatste, dus hoogste, key terug van deze gesorteerde map.

<code>headMap(K toKey)</code>	<code>SortedMap&lt;K,V&gt;</code>	Geeft een view van deze map van alle key-value-paren waarvan de key kleiner is dan <code>toKey</code> .
<code>subMap(K fromKey, K toKey)</code>	<code>SortedMap&lt;K,V&gt;</code>	Geeft een view van deze map van alle key-value-paren waarvan de key groter of gelijk is aan <code>fromKey</code> en kleiner is dan <code>toKey</code> .
<code>tailMap(K fromKey)</code>	<code>SortedMap&lt;K,V&gt;</code>	Geeft een view van deze map van alle key-value-paren waarvan de key groter of gelijk is aan <code>fromKey</code> .

Raadpleeg de documentatie voor een volledig overzicht.

Vervang in het vorige voorbeeld de `LinkedHashMap` door een `TreeMap`:

```
package jpfhfdst14;
import java.util.LinkedHashMap;
import java.util.Map;
public class VbTreeMap {
    public static void main(String[] args) {
        Map<String, String> landen = new TreeMap<>();
        landen.put("B", "Belgie");

        ...

    }
}
```

- (1) Er wordt een map gemaakt van het type `TreeMap`. Zowel de key als de value zijn van het type `String`. De sortering van de key-value-paren gebeurt volgens de natural ordening van de key, d.w.z. zoals beschreven is in de `compareTo()` van de class `String`.

Bij de uitvoer zal je merken dat de landen in de treemap geordend zijn in oplopende, alfabetische volgorde van de key, zijnde de landcode.

Een deel van de uitvoer

...

```
Land met code F: Frankrijk
Vorig land met code F: Frankrijk
Land met code F: Finland
```

```
*** View van de Keys ***
```

```
B
D
F
L
NL
```

\*\*\* View van de Keys met bijhorende value-waarde \*\*\*

B heeft als landnaam: Belgie  
D heeft als landnaam: Duitsland  
F heeft als landnaam: Finland  
L heeft als landnaam: Luxemburg  
NL heeft als landnaam: Nederland

\*\*\* View van de Values \*\*\*

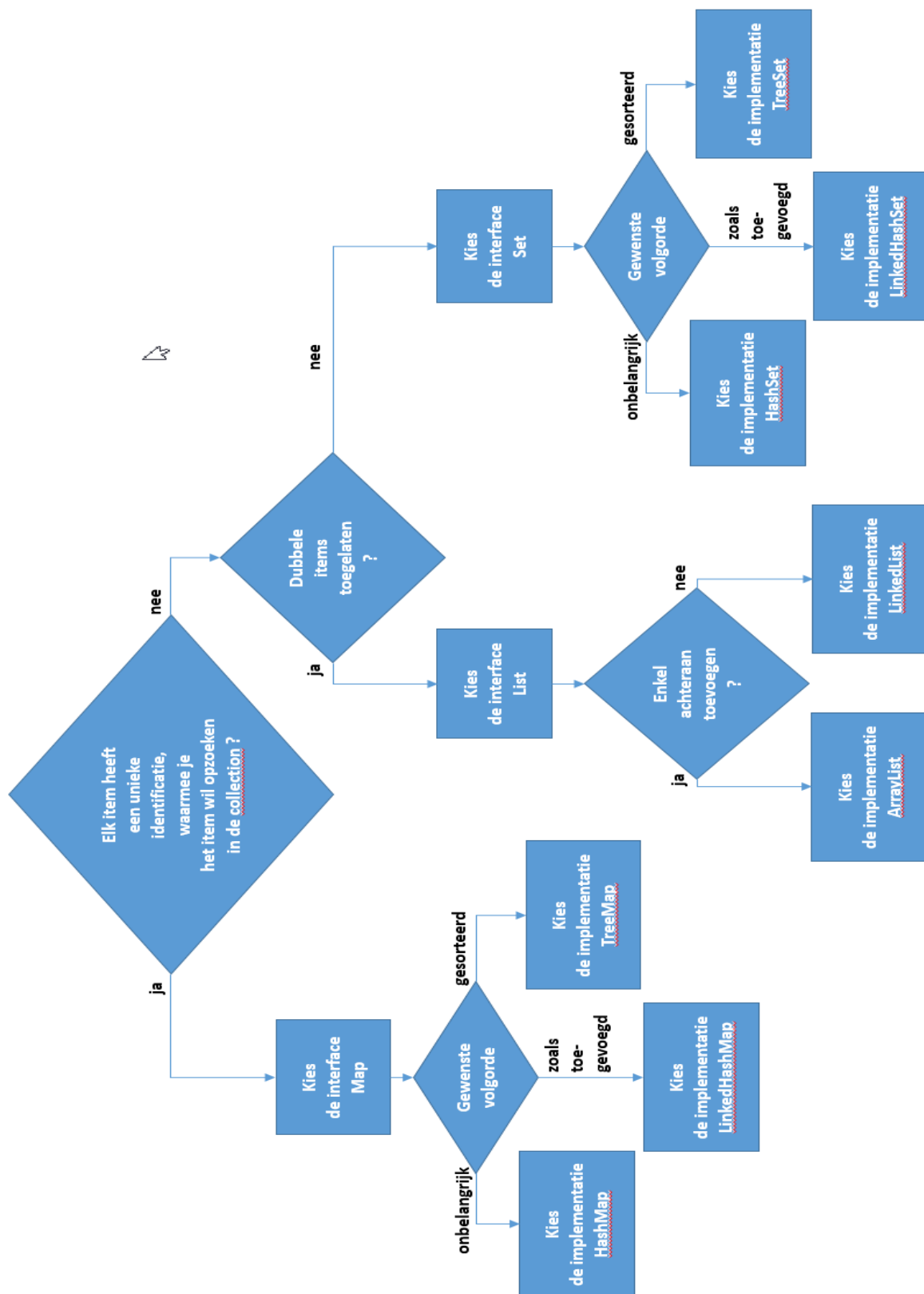
Belgie  
Duitsland  
Finland  
Luxemburg  
Nederland

\*\*\* View van de Key-Value-paren \*\*\*

B=Belgie  
D=Duitsland  
F=Finland  
L=Luxemburg  
NL=Nederland

## 14.10 Schema

Vaak is het moeilijk om te bepalen welke collectie je best gebruikt. Eén en ander hangt af van de gewenste volgorde van de objecten, of er dubbels bewaard mogen worden, enz. Onderstaand schema kan je helpen om de juiste interface en implementatieclass te kiezen uit de Collections.



## 14.11 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 14:

- Beginletter
- Winkel

## 15 Streams - I/O en Serialization

---

Een stream is een *gegevensstroom* waar uit gelezen of waar naar geschreven kan worden. In- en uitvoer van gegevens is zeer divers. Invoer kan invoer zijn via het toetsenbord, gegevens lezen uit een bestand, gegevens ontvangen via het netwerk (in de brede zin van het woord)... Bij uitvoer kan je denken aan uitvoer naar de printer, gegevens schrijven naar een bestand, gegevens versturen via het netwerk... Algemeen spreek je over input/output of kortweg I/O.

De package voor in- en uitvoer-gerelateerde classes is `java.io`. Het zou te ver gaan om alle verschillende classes van deze package te bespreken. Enkele elementaire classes komen zeker aan bod. Dit levert voldoende basisinformatie op om er mee te starten. Het aanspreken van een database valt buiten deze cursus.

Je zal drie soorten I/O streams leren kennen:

- **Byte Streams:** voor input en output van 8-bit bytes
- **Character Streams:** voor input en output van tekstbestanden
- **Object Streams:** voor input en output van objecten (binaire bestanden)

### 15.1 Byte streams

#### 15.1.1 De classes `InputStream` en `OutputStream`

Byte Streams worden gebruikt voor de in- en uitvoer van 8-bit bytes. Er zijn veel byte stream classes en ze zijn afgeleid van de abstracte classes `InputStream` en `OutputStream`.

Er volgt een voorbeeld met het gebruik van de classes **`FileInputStream`** en **`FileOutputStream`**.

In het voorbeeld wordt een txt-file (de input-file) byte per byte gelezen en geschreven naar een andere txt-file (de output-file). Het bestand wordt als het ware gecopieerd.

```
package jpfhfdst15;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = null;
        FileOutputStream fos = null;

        try {
            fis = new FileInputStream("D:/JPF/alfabetIn.txt");           (1)
            fos = new FileOutputStream("D:/JPF/alfabetOut.txt");         (2)
            int c;
            while ((c = fis.read()) != -1) {                             (3)
                fos.write(c);                                           (4)
            }
        }
        catch (IOException e) {                                         (5)
            System.out.println("file niet gevonden");
        }
    }
}
```

```

        finally {
            if (fis != null) {
                fis.close();
            }
            if (fos != null) {
                fos.close();
            }
        }
    }
}

```

(6)

- (1) Er wordt een referentie gemaakt naar de input-file, zijnde *alfabetIn.txt* (zie theoriemateriaal.zip). Dit tekstbestand bevat de letters van het alfabet, de cijfers en enkele speciale karakters.  
In dit voorbeeld staat *alfabetIn.txt* op de D-schijf in de folder JPF (D:\JPF). Het pad kan ook aangeduid worden als volgt: "D:\\JPF\\alfabetIn.txt" waarbij de dubbele slash het escape character is voor de enkele slash.
- (2) Er wordt een output-file gedeclareerd, zijnde *alfabetOut.txt*. Indien deze file reeds bestaat, wordt de file zondermeer overschreven.
- (3) Met de read() method wordt een byte gelezen van de inputstream (inputfile). De returnwaarde van deze method is een int.  
Indien het einde van de file (EOF – end of file) bereikt wordt, geeft deze read() method -1 terug. Zo kan in de lus de input-file van begin tot einde byte per byte gelezen worden.
- (4) Met de write() method wordt de ingelezen byte geschreven naar de outputstream. Deze method bestaat (via overloading) in verschillende varianten.
- (5) De class IOException behoort ook tot de package java.io. Indien de opgegeven inputfile niet bestaat of er is een probleem met de outputfile treedt er een exception op, m.n. een IOException. Om te voorkomen dat het programma afgebroken wordt, wordt de exception opgevangen.
- (6) Het is belangrijk dat de inputstream en outputstream objecten gesloten worden. Dit moet ten allen tijde gebeuren. Daarom wordt dit in het finally block geplaatst.

Om te vermijden dat de outputfile overschreven wordt, kan je gebruik maken van een andere constructor. De constructor die naast de naam van de file ook een boolean-parameter bevat, waarmee je aangeeft of de file al dan niet moet overschreven worden. Met *false* geef je aan dat de file overschreven mag worden, met *true* geef je aan dat gegevens achteraan de file toegevoegd mogen worden.

Indien je de regel code aangeduid met (2) als volgt aanpast:

```
fos = new FileOutputStream("D:/JPF/alfabetOut.txt", true);
```

zal je vaststellen dat na uitvoering van het programma de gegevens van de inputfile zijn toegevoegd aan de outputfile.

### 15.1.2 Try with resources

Zoals je in het vorig voorbeeld zag, dien je steeds de file-objecten te sluiten. M.b.v. het try with resources statement kan het voorbeeld verkort worden. Je kan dan een te sluiten object aanmaken



binnen het try statement. Op het einde van het try statement wordt dit object automatisch gesloten. Het voorbeeld wordt dan:

```
package jpfhfdst15;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytesTwr {
    public static void main(String[] args) throws IOException {

        try (FileInputStream fis=new FileInputStream("D:/JPF/alfabetIn.txt");
            FileOutputStream fos=new FileOutputStream("D:/JPF/alfabetOut.txt");) {
            int c;
            while ((c = fis.read()) != -1) {
                fos.write(c);
            }
        }
        catch (IOException e) {
            System.out.println("file niet gevonden");
        }
    }
}
```

In het try statement worden de te sluiten objecten aangemaakt. De statements hiervoor plaats je tussen ronde haken achter *try*. Daarna volgt de code tussen accolades. Die is hier onveranderd gebleven. Het finally blok is komen te vervallen, aangezien de objecten automatisch gesloten worden.

De te sluiten objecten die je binnen het try statement kan aanmaken zijn allen objecten van classes die de interface *java.lang.AutoCloseable* implementeren.

Vanzelfsprekend wordt deze werkwijze aanbevolen waar mogelijk.

### 15.1.3 BufferedInputStream en BufferedOutputStream

Tot nu toe gaat het in het voorbeeld om ongebufferde in- en uitvoer. Dit betekent dat elke lees- en schrijfo opdracht meteen afgehandeld wordt. Dat maakt je programma minder efficiënt want elke opdracht vraagt toegang tot je schijf (kan ook toegang zijn tot het netwerk, of een andere bron die tijdrovend is).

Om dit terug te dringen bestaan er gebufferde I/O streams. Gebufferde input streams lezen gegevens uit een buffer (een soort geheugengebied). Wanneer de buffer leeg is, wordt er pas opnieuw een leesopdracht van de schijf uitgevoerd. Hetzelfde gebeurt bij een schrijfo opdracht. Gegevens worden geschreven naar een buffer en pas wanneer de buffer vol is gebeurt een schrijfo opdracht naar de eigenlijke schijf. Hoeveel gegevens in de buffer kunnen handelt de class voor ons af. Alleszins maak je je programma met gebufferde in- en uitvoer efficiënter.

Het voorbeeld wordt dan:

```
package jpfhfdst15;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```

public class CopyBytesBuffered {
    public static void main(String[] args) throws IOException {

        try (BufferedInputStream bis= new BufferedInputStream(
            new FileInputStream("D:/JPF/alfabetIn.txt"));           (1)
            BufferedOutputStream bos= new BufferedOutputStream(
                new FileOutputStream("D:/JPF/alfabetOut.txt")); ) { (2)
            int c;
            while ((c = bis.read()) != -1) {                       (3)
                bos.write(c);
            }
        }
        catch (IOException e) {
            System.out.println("file niet gevonden");
        }
    }
}

```

- (1) Het te lezen bestand wordt hier aangegeven via de constructor van een `FileInputStream`. Dit object wordt gebruikt als argument bij de constructor van een `BufferedInputStream`. Op deze manier worden beide objecten gelinkt.
- (2) Hetzelfde gebeurt voor de `BufferedOutputStream`.
- (3) Verder lees en schrijf je naar de buffer in plaats van naar de in- of outputstream.

Door gebruik te maken van `try with resources` wordt de `BufferedInputStream` en `BufferedOutputStream` automatisch gesloten. Bij een `BufferedOutputStream` wordt de laatste inhoud van de buffer geschreven naar het bestand op het moment dat de buffer gesloten wordt! Indien je geen gebruik maakt van `try with resources` is het dus belangrijk om de `BufferedOutputStream` zelf te sluiten.

`BufferedInputStream` en `BufferedOutputStream` zijn afgeleid van respectievelijk `FilterInputStream` en `FilterOutputStream`, kortweg filterstreams genoemd. Er bestaan nog andere filterstreams zoals o.a. `datastreams`, nl. `DataInputStream` en `DataOutputStream`. `DataInputStream` heeft o.a. volgende methods `readBoolean()`, `readByte()`, `readChar()`, `readDouble()`, `readFloat()`, `readInt()`, enz. Deze read methods geven echter niet -1 terug in geval van EOF. Wanneer het einde van de file bereikt is, zal er een `EOFException` gethrowed worden. `DataOutputStream` heeft de write-versie van dergelijke methods.

Raadpleeg de documentatie voor een volledige beschrijving van de classes en bijhorende methods.

## 15.2 Character streams

### 15.2.1 Unicode

Voor het bewaren van character waardes gebruikt Java de Unicode conventions. Elk karakter heeft een unicode. De unicode bestaat uit 16 bits per karakter. Het vormt een uniek getal voor elk teken, ongeacht het gebruikte platform, ongeacht het gebruikte programma, ongeacht de gebruikte taal. Men spreekt van een character set, de Unicode Character Set.

Zie <http://www.unicode.org/standard/translations/dutch.html> voor meer informatie.

### 15.2.2 De classes Reader en Writer

Character Streams lezen karakter per karakter in. Ze vertalen automatisch elk karakter van en naar de lokale character set (de Unicode Character set).

De character stream classes zijn afgeleid van de abstracte classes Reader en Writer. Er volgt een voorbeeld met het gebruik van de classes **FileReader** en **FileWriter**, respectievelijk afgeleid van **InputStreamReader** en **OutputStreamWriter**.

```
package jpfhfdst15;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        try (FileReader fr = new FileReader("D:/JPF/alfabetIn.txt");
            FileWriter fw = new FileWriter("D:/JPF/alfabetOutChar.txt");) {
            int c;
            while ((c = fr.read()) != -1) {
                fw.write(c);
            }
        }
        catch (IOException e) {
            System.out.println("file niet gevonden");
        }
    }
}
```

Je zal vaststellen dat de karakters als het ware gecopieerd zijn naar de ouputfile.

De read() method van de betreffende classes verschillen uiteraard. De read() method van de byte streams leest een byte terwijl de read() method van character streams een teken leest.

### 15.2.3 De classes BufferedReader en BufferedWriter

Ook hier bestaat een gebufferde versie van de classes met dezelfde gekende voordelen:

```
package jpfhfdst15;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharactersBuffered {
    public static void main(String[] args) throws IOException {
        try (BufferedReader br= new BufferedReader(new
            FileReader("D:/JPF/alfabetIn.txt"));

            BufferedWriter bw= new BufferedWriter(new
            FileWriter("D:/JPF/alfabetOutChar.txt"));) {
            int c;
            while ((c = br.read()) != -1) {
                bw.write(c);
            }
        }
    }
}
```

```

        catch (IOException e) {
            System.out.println("file niet gevonden");
        }
    }
}

```

### 15.3 Andere file operaties

Het lezen en schrijven van files is een deel van de mogelijke bewerkingen op bestanden. De package *java.io* bevat ook classes voor het uitvoeren van andere bewerkingen op bestanden, bijv. hernoemen van files, verwijderen van files, aanmaken van directory's, checken op het bestaan van bestanden, enz. Kortom het beheeren van het file system. De class hiervoor is de class **File**.

Een File object creëren: bijv.

```
File inputFile = new File("D:/JPF/alfabetIn.txt");
```

Zo kan je in een bovenstaand voorbeeld de regel code

```
BufferedReader br= new BufferedReader(new
    FileReader("D:/JPF/alfabetIn.txt"));
```

vervangen door:

```
BufferedReader br= new BufferedReader(new
    FileReader(inputFile));
```

### 15.4 Object streams

Object stream ondersteunen input en output van objecten, m.a.w. objecten naar een stream wegschrijven en objecten van een stream inlezen. Objecten kunnen eenvoudige objecten zijn, bijv. een voertuigobject. Maar ook arrays van objecten of collections van objecten, meervoudige objecten dus, kunnen met een object stream geschreven en gelezen worden. Een collection op zich is trouwens ook een object.

#### 15.4.1 Serialization

Opdat objecten van een class naar een file geschreven kunnen worden, moet de class hiervoor voorbereid worden, m.a.w. *Serializable* gemaakt worden. Eigenlijk worden alle bits van het object op een rijtje gezet en worden ze geserialiseerd weggeschreven. Bij het inlezen worden de objecten terug gedeserialiseerd. Het geheel is een bepaald proces, het serialisatieproces.

Een class serializable maken doe je door bij de class de interface *Serializable* te implementeren. Je voegt zonder meer `implements Serializable` toe in de class header en je hoeft verder niets te doen. Je dient zelf geen method te definiëren in je class. Het serialisatieproces wordt door de JVM beheerd. Dit wordt een marker interface genoemd. Het geeft een indicatie aan de JVM dat de betreffende class serialiseerbaar is zodat objecten ervan kunnen weggeschreven worden naar schijf.

Je maakt gebruik van de class *Cursus* uit het vorig hoofdstuk en past de header als volgt aan:

```
package jpfhfdst15ser;
import java.io.Serializable;
```

```
public class Cursus implements Comparable<Cursus>, Serializable {
    private int cursusNr;
    private String cursusNaam;
    private int prijs;
    ...
}
```

De class Cursus is nu klaar opdat objecten ervan weggeschreven kunnen worden. De class Cursus bevat een referentie naar de String class voor één van de membervariabelen. Zo kan een class meerdere referenties bevatten naar andere classes. Opdat objecten ervan volledig kunnen worden weggeschreven is het belangrijk dat ook deze classes serialiseerbaar zijn. De class String is serialiseerbaar, dus de class Cursus is in orde.

Een class kan dus referenties hebben naar een andere class, maar ook deze class kan op zich weer referenties hebben naar nog een andere class, enz. Zo ontstaat een soort boomstructuur van objecten. Met de writeObject() method worden dergelijke objecten weggeschreven naar een stream, en de readObject() method leest ze weer correct in. Ondanks deze methods gemakkelijk te gebruiken zijn, bevatten zij toch gesofisticeerde logica om de boomstructuur van objecten correct weg te schrijven en bij het inlezen terug correct op te bouwen. Zo kan een writeObject() method een groot aantal objecten wegschrijven naar de stream en de readObject() method een groot aantal objecten terug inlezen van de stream.

#### 15.4.2 De classes ObjectOutputStream en ObjectInputStream

Object streams kunnen dus objecten schrijven naar en lezen van een stream. De classes ObjectOutputStream en ObjectInputStream zijn respectievelijk afgeleid van OutputStream en InputStream.

Volgend main-programma toont het gebruik ervan door een aantal cursusobjecten weg te schrijven en nadien terug in te lezen:

```
package jpfhfdst15ser;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class CursusMain {
    public static void main(String[] args) {

        List<Cursus> cursussenIn = new ArrayList<>();
        cursussenIn.add(new Cursus(5, "Word", 100) );
        cursussenIn.add(new Cursus(3, "Excel", 110) );
        cursussenIn.add(new Cursus(1, "Windows", 90) );
        cursussenIn.add(new Cursus(4, "Access", 120) );
        cursussenIn.add(new Cursus(2, "Powerpoint", 80) );
        cursussenIn.add(new Cursus(6, "Photoshop", 100) );

        //List met cursussen wegschrijven naar een object stream
        try (FileOutputStream fos =
            new FileOutputStream("D:/JPF/cursussen.obj");
```

(1)

```

        ObjectOutputStream oos = new ObjectOutputStream(fos);) { (2)
        for (Cursus cursus : cursussenIn) {
            oos.writeObject(cursus); (3)
        }
    }
    catch (IOException e) {
        System.out.println("file niet gevonden");
    }

    //De weggeschreven cursussen terug inlezen van de object stream
    List<Cursus> cursussenOut = new ArrayList<>();
    Cursus cursus;

    try (FileInputStream fis = (4)
        new FileInputStream("D:/JPF/cursussen.obj");
        ObjectInputStream ois = new ObjectInputStream(fis);) { (5)
        for (int i = 0 ; i<= 5 ; i++) {
            cursus = (Cursus) ois.readObject(); (6)
            cursussenOut.add(cursus);
        }
    }
    catch (ClassNotFoundException | IOException e) { (7)
        System.out.println("probleem met de file");
    }
    System.out.println(cursussenOut);
}

```

- (1) Via een `FileOutputStream` wordt het bestand aangegeven waarnaar de cursusobjecten worden weggeschreven. De extensie van de file is best geen `.txt` vermits het hier niet om platte tekst gaat, maar om binaire gegevens.
- (2) De `ObjectOutputStream` wordt aangemaakt en de `fileoutputstream` wordt gebruikt bij de constructor. Op deze manier worden beide objecten gelinkt.
- (3) Een cursusobject wordt weggeschreven naar de `ObjectOutputStream` met de `writeObject()` method.
- (4) Via een `FileInputStream` wordt het bestand aangegeven waaruit de cursusobjecten worden ingelezen.
- (5) De `ObjectInputStream` wordt aangemaakt en de `FileInputStream` wordt gebruikt bij de constructor. Zoals bij de `ObjectOutputStream` worden beide objecten gelinkt.
- (6) Met de `readObject()` method wordt een object van type `Object` ingelezen. Daarom is typecasting nodig. In de lus worden exact zes cursusobjecten ingelezen. Je kan dat zo doen omdat je weet dat er zoveel zijn weggeschreven. Indien je niet weet om hoeveel objecten het gaat, dien je in de lus te blijven lezen totdat een `IOException` gethrowed wordt.
- (7) Hier wordt een `IOException` en `ClassNotFoundException` opgevangen. Aangezien er bij de `readObject()` method typecasting nodig is, kan er een `ClassNotFoundException` optreden in het geval er geen class `Cursus` in de package zou zijn.

### 15.4.3 Transient variabelen

Indien je wenst om niet alle membervariabelen van het object weg te schrijven naar een bestand, dien je deze membervariabelen transient te maken. Je doet dit door vlak voor het type van de

membervariabele het woord *transient* te plaatsen. Zo markeer je de membervariabele als “niet weg te schrijven”. Het serialisatieproces houdt hiermee geen rekening.

Als voorbeeld markeer je bij de class *Cursus* de prijs transient:

```
private transient int prijs;
```

Dit dien je ook te doen voor referenties van membervariabelen naar andere type classes die niet serializable zijn.

Wanneer je nu cursusobjecten wegschrijft en terug inleest, zal de prijs geïntialiseerd worden op zijn default-waarde, d.w.z. 0 voor integer.

Transiente membervariabelen worden bij het inlezen geïntialiseerd op hun default-waarde: 0 voor integers, 0.0 voor floats, *false* voor boolean, *null* in geval het gaat om een referentie naar een ander object.

Wanneer je hetzelfde main-programma opnieuw uitvoert, zal je dus zien dat de prijs bij het inlezen steeds op 0 wordt gezet.

#### 15.4.4 serialVersionUID

In de praktijk kan tussen het wegschrijven en het inlezen van objecten (bijv. cursusobjecten) een lange tijdspanne zitten. Indien in deze tussentijd de class gewijzigd is, wens je toch nog de oudere objecten in te lezen. Dit kan een conflict geven met de typecasting, aangezien de class gewijzigd is t.o.v. het moment van wegschrijven. Er kunnen membervariabelen gewijzigd of toegevoegd zijn, methods gewijzigd of toegevoegd zijn, de header van de class kan gewijzigd zijn, enz. Het probleem is dus dat het weggeschreven object niet meer overeenkomt met de huidige versie van de class.

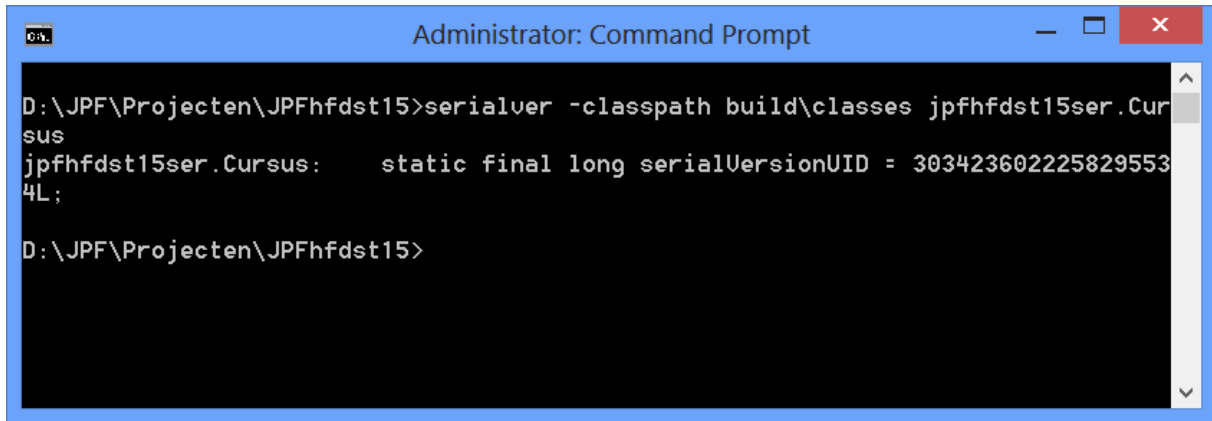
Een oplossing voor dit probleem is het *serialVersionUID*. Dit is een uniek versienummer dat de compiler automatisch toevoegt aan de class als een soort membervariabele. Iedere keer wanneer de class gecompileerd wordt, wijzigt dit nummer. Het is een static final membervariabele van het type *long*.

Met de tool *serialver* kan je in het cmd-venster het huidige *serialVersionUID* van de class opvragen.

Positioneer je in de directory waar het project bewaard is. In het voorbeeld is dat

*D:\JPF\Projecten\JPFhfdst15*. Als parameter van *serialver* geef je met *-classpath* de volledige naam van de class mee. Hier is dat *build\classes\jpfhfdst15ser.Cursus*. De gecompileerde classes staan per definitie in de folder *build\classes* en de volledige class naam is de naam van de class inclusief de package naam, vandaar *build\classes\jpfhfdst15ser.Cursus*

De class naam is ook hier hoofdlettergevoelig!



```

Administrator: Command Prompt

D:\JPF\Projecten\JPFhfdst15>serialver -classpath build\classes jpfhfdst15ser.Cursus
jpfhfdst15ser.Cursus: static final long serialVersionUID = 3034236022258295534L;

D:\JPF\Projecten\JPFhfdst15>

```

In bovenstaand voorbeeld heeft het serialVersionUID de waarde 3034236022258295534L. Wanneer je de class Cursus na een kleine aanpassing opnieuw compileert, zal je merken dat het serialVersionUID verschillend is van de vorige versie.

Normaal worden static membervariabelen niet mee geserialiseerd, doch serialVersionUID vormt hierop een uitzondering. Op het moment dat weggeschreven data ingelezen wordt van een object stream in een object, zal het serialVersionUID van het ingelezen object vergeleken worden met het serialVersionUID van de actuele versie van de class. Zijn de serialVersionUID's verschillend dan wordt een exception gethrowed. Op die manier wordt er bijvoorbeeld voorkomen dat je membervariabelen of methods gaat gebruiken die niet meer in de class voorzien zijn.

Om ervoor te zorgen dat je de weggeschreven objecten opnieuw kan inlezen, mag het serialVersionUID dus niet wijzigen. Daarom kan je dit best zelf opnemen in je class. Voeg bij de class Cursus het serialVersionUID toe als volgt:

```

public class Cursus implements Comparable<Cursus>, Serializable {
    public static final long serialVersionUID = 1L;

    private int cursusNr;
    private String cursusNaam;
    private int prijs;
}

```

Zo kunnen objecten te allen tijde terug ingelezen worden. Extra membervariabelen worden geïnitieerd op hun default-waarde omdat ze destijds niet zijn weggeschreven. Verwijderde membervariabelen worden op *null* geïnitieerd.

## 15.5 Oefeningen



Zie takenbundel: maak oefeningen die horen bij hoofdstuk 15:  
- Gastenboek



## 16 Multithreading

---

### 16.1 Processen en threads

#### 16.1.1 Proces

Een proces is een programma in uitvoering.

Een tekstverwerker en een rekenbladprogramma die je startte, zijn dus 2 processen.

Elk proces heeft zijn eigen interne geheugenruimte.

Een proces kan niet lezen of schrijven in de geheugenruimte van een ander proces.

#### 16.1.2 Thread

Een thread is het uitvoeren van code binnen een proces.

Ieder proces heeft minstens één thread.

Een proces kan meerdere threads hebben. Dit heet multithreading: het tegelijk uitvoeren van verschillende code binnen een proces. Een browser is bijvoorbeeld een multithreaded programma: je kan een groot bestand downloaden en tegelijk surfen naar andere pagina's. De browser voert het downloaden uit met een thread, en het surfen met een andere thread. Je kan zelfs meerdere bestanden tegelijk downloaden. De browser voert deze extra taken uit met extra threads.

Alle threads binnen een proces delen de geheugenruimte van dat proces.

Naast multithreading bestaat ook het woord multiprocessing.

Dit betekent het gelijktijdig uitvoeren van meerdere processen (applicaties).

Multithreading en multiprocessing vormen één groot geheel:

de computer voert meerdere gelijktijdige processen uit, die zelf één of meerdere gelijktijdige threads uitvoeren.

### 16.2 Het verdelen van threads over processoren

Als een computer minstens evenveel processoren bevat als het aantal uit te voeren threads, verdeelt de computer de threads over de processoren. Iedere processor voert een thread uit. Gelijktijdig voeren de andere processoren een andere thread uit.

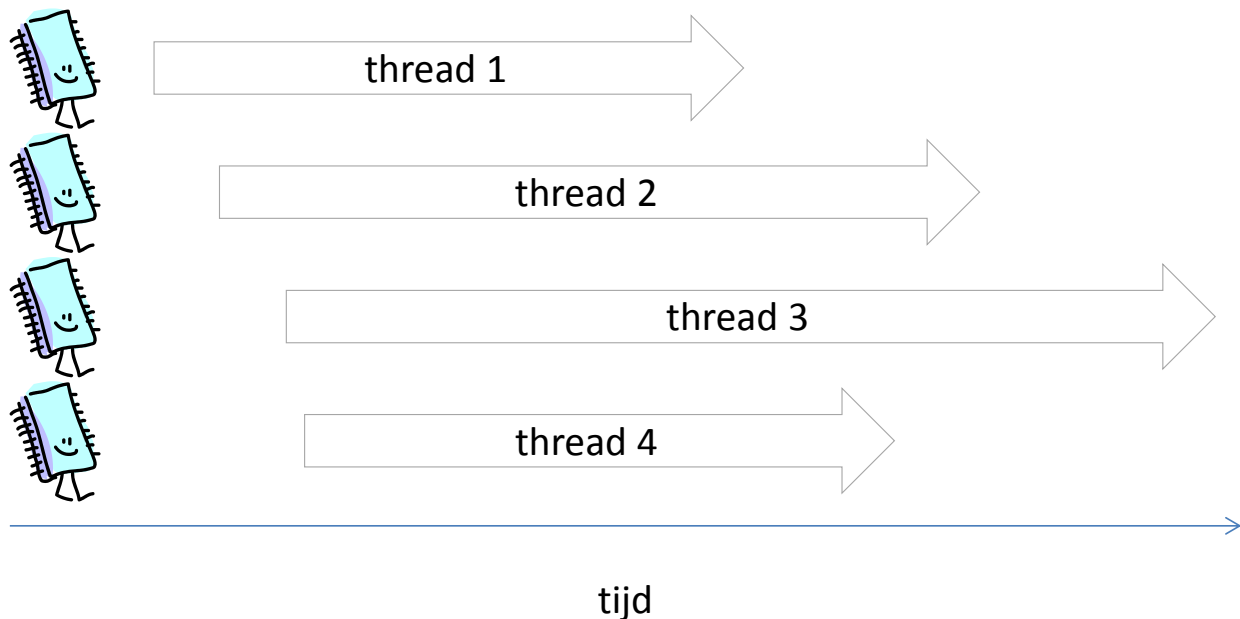
Dit leidt tot een optimale performantie.

Voorbeeld: een computer met vier processoren moet twee processen uitvoeren, die elk twee threads uitvoeren. De computer moet dus in totaal vier threads uitvoeren. Iedere processor voert één van deze threads uit.

Opmerking:

deze threads hoeven niet op hetzelfde moment te starten, en de uitvoeringstijd hoeft niet even lang te zijn.

In de volgende afbeelding staan links de vier processoren, en daarnaast de thread die ze gelijktijdig uitvoeren.



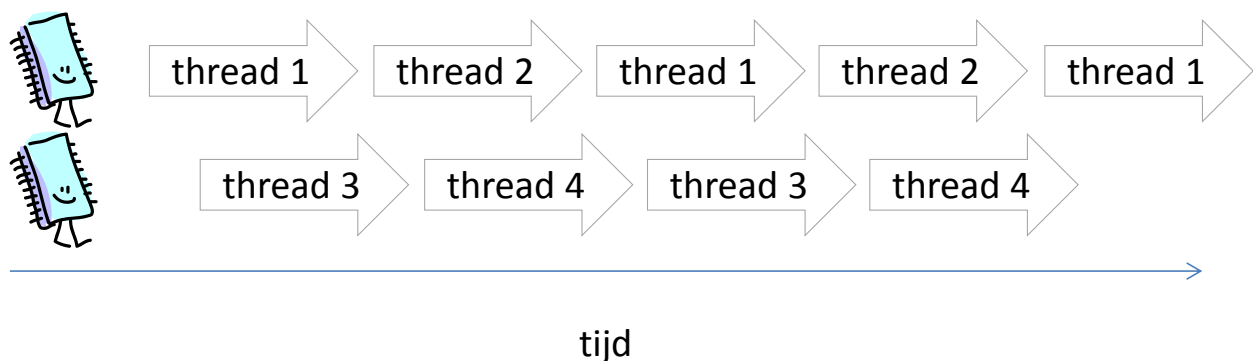
Als de computer minder processoren bevat dan het aantal uit te voeren threads, verdeelt het besturingssysteem de threads over de processoren.

Als er twee processoren zijn en vier threads, voert de ene processor twee threads uit en de andere processor de overige twee threads.

Een processor kan echter op een bepaald moment maar één thread uitvoeren.

Om dit probleem op te lossen gebruikt het besturingssysteem timeslicing:

de processor voert een aantal milliseconden code uit van de eerste thread en zet dan deze thread op pauze. De processor voert daarna een aantal milliseconden code uit van de tweede thread en zet dan deze thread op pauze. De processor voert terug een aantal milliseconden code uit van de eerste thread en zet dan deze thread op pauze, ...



Je leert in dit hoofdstuk hoe je in een Java applicatie (ook een proces) meerdere taken gelijktijdig kan uitvoeren, door meerdere threads te gebruiken.

Een Java applicatie heeft minstens één thread: de thread die de code uitvoert die begint bij `public static void main(String[] args)`.

## 16.3 Threads in Java

Een thread is een object.

Je kan de class, die een thread object voorstelt, op twee manieren maken:

- Als een class die erft van de class Thread.
- Als een class die de interface Runnable implementeert.

De tweede manier wordt aangeraden. Bij de tweede manier kan je nog vrij kiezen om de class te erven van om het even welke class. Bij de eerste manier kan je class niet meer erven van om het even welke class, want de class erft al van Thread.

We zien eerst de (wat eenvoudiger) eerste manier, daarna de tweede manier.

Bij de cursus horen de bestanden *insecten1.csv* en *insecten2.csv*.

Beide bestanden bevatten informatie over insecten.

Één regel bevat de naam en de prijs (gescheiden door een ;) van één insect.

Je plaatst deze bestanden in een directory *c:\teksten*.

Je toont in een applicatie de regels met een prijs tot en met 3.

Je zoekt met één thread in *insecten1.csv*. Je zoekt tegelijk met een tweede thread in *insecten2.csv*.

Om de output van de twee threads te onderscheiden, stuurt de ene thread zijn output naar System.out. NetBeans toont deze output met zwarte letters. De tweede thread stuurt zijn output naar System.err (waar je normaal gezien enkel foutberichten naar stuurt). NetBeans toont deze output met rode letters.

### 16.3.1 Een class die erft van de class Thread

De class die de thread voorstelt

- Je maakt een class (bijvoorbeeld MyThread) die erft van de class Thread.
- Je override de method run (die je erft van Thread).
- Je schrijft in deze method de code die je in een thread wil uitvoeren.

De thread uitvoeren:

- Je maakt een object van je thread class  
`MyThread myThread = new MyThread();`
- Je voert op dit object de method start uit.  
Deze method vraagt aan het besturingssysteem een nieuwe thread en voert met die thread de code uit in de method run (van de class MyThread).

Je maakt een nieuw project.

Je maakt daarin een package *be.vdab* en daarin een class *InsectenLezer*:

```
package be.vdab;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
```

```

import java.io.PrintStream;
import java.math.BigDecimal;

public class InsectenLezer extends Thread {
    private String bestandsNaam; // zal insecten1.csv of insecten2.csv zijn
    private BigDecimal maximum = BigDecimal.valueOf(3);
    private PrintStream stream; // staat voor System.out of System.err

    public InsectenLezer(String bestand, PrintStream stream) {
        this.bestandsNaam = bestand;
        this.stream = stream;
    }

    @Override
    public void run() {
        try (BufferedReader reader = new BufferedReader(new FileReader(new
File(bestandsNaam)))) {
            String regel = reader.readLine();
            while (regel != null) {
                String[] regelOnderdelen = regel.split(";");
                BigDecimal prijs = new BigDecimal(regelOnderdelen[1]);
                if (prijs.compareTo(maximum) <= 0) {
                    stream.println(bestandsNaam + ':' + regel);
                }
                regel = reader.readLine();
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}

```

Je maakt in de package be.vdab ook een class InsectenMain:

```

package be.vdab2;

public class InsectenMain {
    public static void main(String[] args) {
        InsectenLezer thread1 =
            new InsectenLezer("c:/teksten/insecten1.csv", System.out);
        InsectenLezer thread2 =
            new InsectenLezer("c:/teksten/insecten2.csv", System.err);
        thread1.start();
        thread2.start();
    }
}

```

Je kan de applicatie uitproberen.

### 16.3.2 Een class die de interface Runnable implementeert

De class die de thread voorstelt:

- Je implementeert in de class (bijvoorbeeld MyRunner) de interface Runnable.
- Je implementeert de method run (gedecclareerd in Runnable).
- Je schrijft in deze method de code die je in een thread wil uitvoeren.

De thread uitvoeren:

- Je maakt een object van je eigen class  
MyRunner myRunner = new MyRunner();
- Je maakt een Thread object en je geef hierbij het object van je eigen class mee aan de constructor van Thread.  
Thread thread = new Thread(myRunner);
- Je voert op dit Thread object de method start uit.  
Deze method vraagt aan het besturingssysteem een nieuwe thread en voert met die thread de code uit in de method run (van het MyRunner object).

Je probeert dit uit.

Je wijzigt in de class InsectenLezer de regel

```
public class InsectenLezer extends Thread
```

naar

```
public class InsectenLezer implements Runnable {
```

Je wijzig de method main van de class InsectenMain:

```
package be.vdab;
public class InsectenMain {
    public static void main(String[] args) {
        InsectenLezer insectenLezer1 =
            new InsectenLezer("c:/teksten/insecten1.csv", System.out);
        InsectenLezer insectenLezer2 =
            new InsectenLezer("c:/teksten/insecten2.csv", System.err);
        Thread thread1 = new Thread(insectenLezer1);
        Thread thread2 = new Thread(insectenLezer2);
        thread1.start();
        thread2.start();
    }
}
```

Je kan de applicatie uitproberen

## 16.4 De method join van een Thread object

Als je in een thread *a* de method *join* uitvoert op een thread object *b*, pauzeert Java de uitvoering van de thread *a* tot de method *run* van het object *b* helemaal uitgevoerd is.

Dit is noodzakelijk als thread *a* het eindresultaat van het werk van het thread object *b* nodig heeft. Thread *a* mag het resultaat van het thread object *b* maar opvragen nadat het thread object *b* zijn resultaat volledig aangemaakt heeft. Thread *a* mag het resultaat van het thread object *b* nog niet opvragen als het thread object *b* zijn resultaat nog aan het opbouwen is.

Je past de applicatie aan. De threads tonen niet de regels met een maximum prijs 3, maar tellen deze regels. De class InsectenMain toont de som van deze twee tellers.

Je voegt aan de class InsectenLezer een private variabele toe:

```
private int aantalRegels;
```

Je vervangt in de method `run` de regel:

```
stream.println(bestandsNaam + ':' + regel);
```

door:

```
++aantalRegels;
```

Je voegt een method `getAantalRegels` toe:

```
public int getAantalRegels() {
    return aantalRegels;
}
```

Je vraagt eerst in de class `InsectenMain` het eindresultaat aan beide threads, zonder te wachten tot ze hun werk gedaan hebben, om te zien dat je dan een verkeerd resultaat krijgt.

Je voegt na de regel:

```
thread2.start();
```

volgende regels toe:

```
System.out.print("aantal regels:");
System.out.println(insectenLezer1.getAantalRegels() +
    insectenLezer2.getAantalRegels());
```

Je voert het programma uit. Je krijgt een verkeerd resultaat. Het juiste resultaat is 6121.

Je lost het probleem nu op. Je voegt juist na de regel:

```
thread2.start();
```

volgende regels toe:

```
try {
    thread1.join();
    thread2.join();
} catch (InterruptedException ex) {
    // Het uitvoeren van de join method kan een InterruptedException werpen
    // Je ziet hierover meer later in de cursus
    System.err.println(ex);
}
```

Je voert het programma uit en je krijgt wel het juiste resultaat (6121).

## 16.5 De static method `sleep` van de class `Thread`

Je kan in om het even welke thread de static method `sleep` van de class `Thread` oproepen. Je geeft als parameter een aantal miliseconden mee. Java zet de thread waarin je deze method oproep doet, evenveel milliseconden op pauze.

Je probeert dit uit in een nieuwe class `Klok`.

Je toont in deze class één keer per seconde de systeemtijd.

```
package be.vdab;
```

```
import java.util.Date;


public class Klok implements Runnable {
    @Override
    public void run() {
        while (true) {
            System.out.println(new Date());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                // het uitvoeren van de sleep method
                // kan een InterruptedException werpen
                // Je ziet hierover meer later in de cursus
                System.err.println(ex);
            }
        }
    }
}
```

Je schrijft een class KlokMain:

```
package be.vdab;

public class KlokMain {
    public static void main(String[] args) {
        Klok klok = new Klok();
        Thread thread = new Thread(klok);
        thread.start();
    }
}
```

Je kan het programma uitvoeren.

Het programma stopt niet omdat de method run van de class Klok een oneindige lus bevat. Je moet het programma afbreken, met een klik op , rechts onder in NetBeans.

Je lost dit probleem onmiddellijk op.

## 16.6 De method interrupt van een Thread object

Je kan aan een thread aangeven dat je zijn uitvoering wenst stop te zetten door op het Thread object de method interrupt uit te voeren. Deze method stop de thread niet, maar doet één van volgende handelingen:

- Als de thread op pauze staat (tijdens het uitvoeren van de Thread.sleep()) of het uitvoeren van de join opdracht op een andere thread), krijgt de thread een InterruptedException.
- Anders komt de thread in de “interrupted” status. In die status blijft de code van de thread lopen. De thread kan op eigen initiatief opvragen of het zich in de interrupted status bevindt, via de static method interrupted van de class Thread. Deze method geeft true terug als de huidige thread zich in de interrupted status bevindt. De thread kan dan eventueel stop gezet worden.

Je wijzigt de applicatie. Wanneer de gebruiker op Enter drukt, stop je de applicatie. Je controleert het toetsenbord in de thread van de class KlokMain. Wanneer de gebruiker op Enter drukt, voer je de method *interrupt* uit op het object thread.

Je voegt in de class KlokMain na de regel:

```
thread.start();
```

volgende regels toe:

```
Scanner scanner = new Scanner(System.in);
scanner.nextLine(); // deze method wacht tot de gebruiker Enter drukt
thread.interrupt();
```

Je wijzigt in de class Klok de method run:

```
@Override
public void run() {
    boolean verderDoen = true;
    while (verderDoen) {
        System.out.println(new Date());
        if (Thread.interrupted()) {
            verderDoen = false; // klok stopzetten
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            verderDoen = false; // klok stopzetten
        }
    }
}
```

Je kan de applicatie uitproberen.

## 16.7 Daemon threads

Normaal stopt een applicatie pas als al zijn threads hun werk gedaan hebben.

Je kan op een Thread object de method setDaemon(true) uitvoeren. Je maakt van die thread een daemon thread. Een applicatie kan wél stoppen terwijl daemon threads hun werk nog niet gedaan hebben.

Je maakt van de thread die de klok afbeeldt een daemon thread.

Je verwijdert in de method run van de class Klok de if structuur.

Je verwijdert in de method main van de class KlokMain de opdracht thread.interrupt();

Je voegt in de method main na de opdracht:

```
Thread thread = new Thread(klok);
```

volgende opdracht toe:

```
thread.setDaemon(true);
```

Je kan de applicatie uitproberen.



## 16.8 Synchronized

Primitieve types (int, long, ...) en de meeste objecten zijn niet thread safe.

Dit betekent dat je ze niet gelijktijdig met meerdere threads mag wijzigen.

Als je dit toch doet, bevatten ze een verkeerde waarde, of werpt Java een exception.

Java bevat een keyword *synchronized*, waarmee je de toegang tot niet thread safe onderdelen van je applicatie synchroniseert. Terwijl één thread het onderdeel wijzigt, kunnen andere threads hetzelfde onderdeel niet wijzigen. Als ze dit toch proberen, pauzeert Java die andere threads, tot de eerste thread zijn wijzigingen gedaan heeft.

Je leert eerst het probleem kennen en daarna hoe je het met synchronized oplost.

### 16.8.1 Voorbeeldapplicatie met het probleem

De class Stapel zal een stapel pannenkoeken voorstellen. De class houdt bij hoeveel pannenkoeken de stapel bevat.

De class Kok stelt een kok voor, die pannenkoeken bakt en op de stapel legt.

De applicatie bevat twee gelijktijdige threads.

Iedere thread stelt een kok voor die 100 pannenkoeken bakt en toevoegt aan de stapel.

Na het uitvoeren van de threads zou de stapel 200 pannenkoeken moeten bevatten.

Dit zal echter niet het geval zijn. Terwijl de ene kok een pannenkoek toevoegt aan de stapel kan de andere kok dit ook doen. Omdat de toegang tot de teller met het aantal pannenkoeken niet gesynchroniseerd is, bevat deze teller snel een verkeerde waarde.

Je voegt een class Stapel toe:

```
package be.vdab;

public class Stapel {
    private int aantalPannenkoeken;

    public void voegPannenkoekToe() {
        ++aantalPannenkoeken;
    }
    public int getAantalPannenkoeken() {
        return aantalPannenkoeken;
    }
}
```

Je voegt een class Kok toe:

```
package be.vdab;

public class Kok implements Runnable {
    private final Stapel stapel;

    public Kok(Stapel stapel) {
        this.stapel = stapel;
    }
}
```

```

@Override
public void run() {
    for (int i = 0; i != 100; i++) {
        stapel.voegPannenkoekToe();
        try {
            Thread.sleep(10);
        } catch (InterruptedException ex) {
            System.err.println(ex);
        }
    }
}
}

```

Je maakt een class PannenkoekenMain:

```

package be.vdab;

public class PannenkoekenMain {
    public static void main(String[] args) {
        Stapel stapel = new Stapel();
        Thread thread1 = new Thread(new Kok(stapel));
        Thread thread2 = new Thread(new Kok(stapel));

        thread1.start();
        thread2.start();
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException ex) {
            System.err.println(ex);
        }
        System.out.println(stapel.getAantalPannenkoeken());
    }
}

```

Je voert de applicatie enkele keren uit. Je zal zelden het correcte resultaat (200) zien.

Je leer hier onder hoe het probleem ontstaat.

De opdracht `++aantalPannenkoeken;` in de method `voegPannenkoekToe` van de class `Stapel`, wordt in machinecode in drie stappen uitgevoerd:

1. De waarde van de variabele `aantalPannenkoeken` van RAM naar CPU brengen
2. Deze waarde in de CPU met één verhogen.
3. De waarde van CPU brengen naar de variabele `aantalPannenkoeken` in RAM.

Als thread A en thread B tegelijk deze stappen uitvoeren, kan het volgende probleem zich voordoen: (we veronderstellen dat `aantalPannenkoeken` nul bevat).

1. Thread A brengt `aantalPannenkoeken` van RAM naar CPU (de waarde in de CPU is dus 0).
2. Thread A verhoogt de waarde in de CPU (de waarde in de CPU is nu 1).
3. Thread B brengt `aantalPannenkoeken` van RAM naar CPU. (de waarde in de CPU is terug 0).

4. Thread B verhoogt de waarde in de CPU  
(de waarde in de CPU is terug 1)
5. Thread A brengt de waarde van de CPU naar RAM  
(de variabele `aantalPannenkoeken` bevat 1)
6. Thread B brengt de waarde van de CPU naar RAM  
(de variabele `aantalPannenkoeken` bevat 1)

Er is dus een fout gebeurt: de variabele bevat 1, terwijl ze 2 zou moeten bevatten.

### 16.8.2 De oplossing van dit probleem met een `synchronized` method

Bij ieder Java object hoort een intern monitor object. Een thread A kan deze monitor vergrendelen. Als een andere thread B dezelfde monitor probeert te vergrendelen, terwijl thread A deze monitor vergrendeld heeft, zet Java de uitvoering van thread B op pauze. Wanneer thread A de monitor ontgrendelt, haalt Java de uitvoering van thread B uit pauze. Thread B vergrendelt de monitor en heeft op zijn beurt de monitor voor zich tot hij de monitor ontgrendelt.

Zoals gezegd heeft ieder Java object een bijbehorende monitor. Een object van de class `Stapel` heeft dus ook een monitor. Terwijl de ene kok een pannenkoek toevoegt aan het `Stapel` object zal hij de monitor van dit `Stapel` object vergrendelen. Hij verhindert zo dat gelijktijdig een andere kok een pannenkoek toevoegt aan de stapel.

Om de monitor van het huidige object (`this`) te vergrendelen tik je voor een method het sleutelwoord `synchronized`. Als een thread de method oproept, vergrendelt Java de monitor van het object waarop de thread de method uitvoert. Terwijl de thread de method uitvoert, zet Java andere threads die dezelfde method willen uitvoeren op hetzelfde object, op pauze. Als de eerste thread op het einde van de method gekomen is, ontgrendelt Java de monitor en laat de threads die wachten op het vrijkomen van de monitor verder werken.

Je wijzigt in de class `Stapel` de declaratie van de method `voegPannenkoekToe`:

```
synchronized public void voegPannenkoekToe () {
```

Je kan de applicatie opnieuw uitproberen. Je krijgt nu een correct resultaat.

Je mag `synchronized` ook schrijven na de sleutelwoorden *public*, *private* of *protected*:

```
public synchronized void voegPannenkoekToe () {
```

### 16.8.3 Een `synchronized` blok

Het is belangrijk de monitor niet langer te vergrendelen dan nodig, om de performantie van je applicatie hoog te houden. Een method kan veel opdrachten bevatten.

Je moet de monitor niet altijd bij al deze opdrachten vergrendelen. Je kan met een `synchronized` blok een monitor locken gedurende het uitvoeren van een deel van de method. Je past dit toe in de method `voegPannenkoekToe` in de class `Stapel`:

```
public void voegPannenkoekToe () {  
    System.out.println("Nog een pannenkoek");  
    synchronized (this) {  
        ++aantalPannenkoeken;  
    }  
}
```

```
}
}
```

- Bij de method zelf is het sleutelwoord `synchronized` weg. Een monitor wordt dus niet vergrendeld zodra een thread de method uitvoert.
- Een `synchronized` blok begint met het sleutelwoord *synchronized*. Je geeft tussen haakjes een object mee (hier het huidig Stapel object). Als een thread in het `synchronized` blok binnenkomt, vergrendelt Java de monitor van dit object. Daarna komt een open-accolade.
- Java ontgrendelt de monitor pas als de thread het `synchronized` blok heeft uitgevoerd. Dit is bij de sluit-accolade van het `synchronized` blok.

Je kan de applicatie uitvoeren.

## 16.9 Thread safe classes in de Java library

De standaard Java library bevat enkele thread safe classes.

In de documentatie van deze classes staat expliciet dat ze thread safe zijn.

Dit zijn classes die je wel met meerdere threads gelijktijdig mag aanspreken en correct werken, ook als je ze niet aanspreekt binnen een `synchronized` method of een `synchronized` blok.

Voorbeelden:

Thread safe class	Niet-thread safe class met dezelfde werking
StringBuffer	StringBuilder
CopyOnWriteArrayList	ArrayList
CopyOnWriteArraySet	LinkedHashSet
ConcurrentHashMap	HashMap

## 16.10 Oefeningen

Zie takenbundel: maak oefeningen die horen bij hoofdstuk 16:

- GemiddeldeRekenaar
- Pannenkoek (uitbreiding)



## COLOFON

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	Brigitte Loenders
Auteurs	Hans Desmet - Brigitte Loenders
Versie	02/02/2016
Codes	Peoplesoftcode: Wettelijk depot:

### Omschrijving module-inhoud

Abstract	Doelgroep	Opleiding Enterprise Java Ontwikkelaar
	Aanpak	Begeleide zelfstudie
	Doelstelling	De fundamentals van de programmeertaal Java leren.
Trefwoorden		JPF
Bronnen/meer info		