



# **Java Programming Fundamentals**

Deze cursus is eigendom van VDAB Competentiecentra ©





## INHOUDSOPGAVE

<b>HOOFDSTUK 1</b>	<b>ALGEMENE SITUERING VAN JAVA.....</b>	<b>1</b>
1.1	Inleiding .....	1
1.2	Kenmerken van Java .....	2
1.3	Evolutie van Java .....	2
1.4	Varianten .....	3
1.5	Hoe realiseert men in Java de platformonafhankelijkheid?.....	3
1.6	Wat hebben we nodig om Java te gebruiken? .....	5
1.7	Java en GUI (Graphical User Interface).....	9
1.8	Enkele afkortingen verklaard .....	10
1.8.1	APPLET .....	10
1.8.2	SERVLET .....	10
1.8.3	JSP .....	10
1.8.4	Java BEANS .....	10
1.8.5	JDBC.....	10
1.8.6	EJB .....	10
1.8.7	DESIGN PATTERNS .....	10
1.9	Java-libraries / packages / jar's / classpath .....	11
<b>HOOFDSTUK 2</b>	<b>VARIABELEN EN OPERATOREN .....</b>	<b>12</b>
2.1	Een eenvoudig programma .....	12
2.2	Variabelen en literals .....	16
2.2.1	Data types .....	16
2.2.2	Variabelen, references en literals .....	16
2.2.3	Constanten.....	18
2.2.4	Enumeration.....	19
2.2.5	Identifiers .....	19
2.2.6	Statements en commentaar .....	20
2.3	Operatoren .....	23
2.3.1	Soorten operatoren .....	23
2.3.2	Prioriteitsregels .....	25
2.3.3	Type casting.....	25
2.3.4	Stringoperatoren .....	26



2.4	Praktijkvoorbeelden.....	26
2.5	User-input .....	28
2.6	Oefeningen .....	29
<b>HOOFDSTUK 3 ARRAYS .....</b>		<b>30</b>
3.1	Arrays van primitive data types of strings.....	30
3.2	Arrays van classes .....	32
3.3	Arrays van arrays .....	33
3.4	Oefeningen .....	34
<b>HOOFDSTUK 4 PROGRAMMAVERLOOP .....</b>		<b>35</b>
4.1	Blokken van statements .....	35
4.2	Keuzes .....	36
4.2.1	De if-instructie .....	36
4.2.2	De conditional operator ? .....	37
4.2.3	De switch.....	38
4.3	Lussen .....	40
4.3.1	De while-lus.....	41
4.3.2	De do...while-lus .....	41
4.3.3	De for-lus.....	41
4.3.4	Breaks en labels.....	42
4.4	Oefeningen .....	47
<b>HOOFDSTUK 5 OO, CLASSES EN OBJECTS .....</b>		<b>48</b>
5.1	Java en Object Oriëntatie.....	48
5.2	Onze eerste class .....	49
5.3	Oefening .....	56
5.4	Het doorgeven van parameters bij een method-aanroep .....	56
5.5	Method overloading.....	56
5.6	Private methods .....	58
5.7	Instance members versus class members .....	59
5.8	Finalize method.....	61
5.9	Scope regels .....	61



5.10	Een kopie maken van een object.....	62
5.11	Oefeningen .....	63

## **HOOFDSTUK 6 INHERITANCE.....64**

6.1	Introductie .....	64
6.2	Method overriding .....	68
6.3	Final methods en final classes.....	70
6.4	Methods van Object overriden .....	70
6.4.1	toString() .....	70
6.4.2	clone() .....	72
6.4.3	equals() .....	72
6.5	Abstracte classes en methods.....	72
6.5.1	Abstracte classes .....	72
6.5.2	Abstracte methods .....	73
6.6	Polymorphism .....	75
6.7	Annotations.....	77
6.8	Oefeningen .....	78

## **HOOFDSTUK 7 STRINGS .....79**

7.1	Introductie .....	79
7.2	Speciale tekens in een string .....	80
7.3	Bewerkingen met strings .....	80
7.3.1	Het vergelijken van strings .....	80
7.3.2	Strings wijzigen .....	81
7.3.3	Strings onderzoeken .....	82
7.3.4	Een voorbeeld .....	83
7.4	Strings: conversie van en naar primitive types .....	84
7.5	Strings opsplitsen .....	84
7.5.1	De method String.split().....	84
7.5.2	De class StringTokenizer.....	85
7.6	De class StringBuffer .....	87
7.7	Oefeningen .....	88



<b>HOOFDSTUK 8</b>	<b>INTERFACES.....</b>	<b>89</b>
8.1	Declaratie en beschrijving .....	90
8.2	Implementatie in een class.....	90
8.3	Interface als een data type .....	93
8.4	Een toepassing op interfaces: cloning – de interface Cloneable .....	94
8.5	Oefeningen .....	96
<b>HOOFDSTUK 9</b>	<b>PACKAGES .....</b>	<b>97</b>
9.1	Algemeen .....	97
9.2	Naamgeving conventie voor packages .....	97
9.2.1	Algemeen .....	97
9.3	Voorwaarden voor classes in packages .....	98
9.3.1	Een package maken in NetBeans .....	98
9.3.2	Een nieuwe class toevoegen aan een bestaande package .....	99
9.3.3	Een nieuwe class toevoegen aan een nieuwe package.....	99
9.3.4	Een class naar een andere package verplaatsen .....	99
9.4	Verwijzen naar interfaces en classes uit een package .....	99
9.4.1	Algemeen .....	99
9.4.2	Ondersteuning van NetBeans.....	100
9.5	Jar-bestand .....	101
9.6	Class path .....	103
9.6.1	Algemeen .....	103
9.6.2	De class path uitbreiden in NetBeans.....	103
9.7	Oefeningen .....	104
<b>HOOFDSTUK 10</b>	<b>EXCEPTION HANDLING .....</b>	<b>105</b>
10.1	Exceptions afhandelen .....	105
10.2	Eigen exceptions maken .....	111
10.3	Oefeningen .....	113
<b>HOOFDSTUK 11</b>	<b>BIGDECIMAL.....</b>	<b>114</b>
11.1	Probleemstelling.....	114
11.2	De BigDecimal class .....	114



11.2.1	Constructors .....	114
11.2.2	Methods .....	115
11.3	De oplossing mét BigDecimal .....	115

## **HOOFDSTUK 12 COLLECTIONS..... 116**

12.1	Inleiding .....	116
12.1.1	Wat is een collection? .....	116
12.1.2	Concepten van collections .....	118
12.2	De collection interface .....	118
12.3	De list interface .....	121
12.3.1	ListIterator .....	123
12.3.2	ArrayList .....	124
12.3.3	LinkedList .....	132
12.3.4	Vershil tussen ArrayList en LinkedList .....	137
12.3.5	Oefeningen.....	137
12.4	De set interface .....	138
12.4.1	De hashcode .....	138
12.4.2	De hashtable .....	140
12.4.3	Hoe werken de HashSet en de LinkedHashSet? .....	140
12.4.4	De HashSet .....	143
12.4.5	De LinkedHashSet.....	145
12.4.6	De TreeSet .....	147
12.4.7	Oefeningen.....	156
12.5	Generics .....	157
12.5.1	Generic classes .....	158
12.5.2	Interface Comparable<T> .....	160
12.5.3	Generic collections .....	162
12.5.4	Argumenten van een wildcard type .....	166
12.6	De Map interface .....	171
12.6.1	Key-value paren .....	172
12.6.2	Interface Map .....	173
12.6.3	HashMap .....	174
12.6.4	LinkedHashMap.....	177
12.6.5	TreeMap .....	178
12.6.6	Oefeningen.....	185



12.7	Sorted collection implementations.....	186
12.7.1	Interface Comparator <T>.....	186
12.7.2	Comparator class als inner class .....	190
12.8	Class collections .....	192
12.8.1	Methods van de class Collections .....	192
12.8.2	Toelichting bij de methods .....	193
12.8.3	Enkele voorbeelden .....	194
12.8.4	Oefeningen .....	196
12.9	Tot slot.....	196

## **HOOFDSTUK 13 STREAMS ..... 198**

13.1	Byte streams .....	198
13.1.1	Eenvoudige in- en output.....	198
13.1.2	Gebufferde in- en output.....	200
13.1.3	In- en output van andere datatypes .....	203
13.2	Character streams.....	205
13.3	Andere file operaties .....	208
13.4	Object streams .....	210
13.4.1	Objecten wegschrijven.....	210
13.4.2	Objecten inlezen .....	212
13.4.3	Transient variabelen .....	212
13.4.4	SerialVersionUID .....	213
13.5	Oefeningen .....	216

## **HOOFDSTUK 14 MULTITHREADING ..... 217**

14.1	Processen en threads .....	217
14.1.1	Proces .....	217
14.1.2	Thread.....	217
14.2	Het verdelen van threads over processoren .....	217
14.3	Threads in Java.....	219
14.3.1	Een class die erft van de class Thread .....	219
14.3.2	Een class die de interface Runnable implementeert .....	221
14.4	De method join van een Thread object .....	221
14.5	De static method sleep van de class Thread .....	222





14.6	De method interrupt van een Thread object .....	223
14.7	Daemon threads .....	224
14.8	Synchronized .....	224
14.8.1	Voorbeeldapplicatie met het probleem.....	225
14.8.2	De oplossing van dit probleem met een synchronized method .....	227
14.8.3	Een synchronized blok.....	227
14.9	Thread safe classes in de Java library.....	228
14.10	Oefeningen.....	228

## **HOOFDSTUK 15 GRAPHICAL USER INTERFACE .....229**

15.1	Swing versus AWT .....	229
15.2	“Hello World” in GUI-versie.....	230
15.3	Een eerste event-driven voorbeeld.....	232
15.4	De belangrijkste swing-componenten.....	234
15.4.1	JFrame .....	234
15.4.2	JPanel .....	235
15.4.3	JButton .....	237
15.4.4	JLabel.....	239
15.4.5	JList.....	239
15.4.6	JComboBox.....	243
15.4.7	JTextField.....	243
15.4.8	JTextArea .....	243
15.4.9	JCheckBox .....	244
15.4.10	JRadioButton.....	245
15.4.11	DialogBoxes .....	246
15.4.12	JMenu .....	247
15.4.13	Andere nuttige swingcomponenten.....	249
15.5	Layout en layout-managers .....	250
15.5.1	FlowLayout .....	250
15.5.2	BorderLayout.....	251
15.5.3	GridLayout.....	253
15.5.4	Het nesten van layouts .....	254
15.6	Events & EventListeners.....	255
15.6.1	Een overzicht.....	255



15.6.2	Het gebruik van Adapters .....	257
15.7	Oefeningen .....	260
15.8	Een GUI creëren met de NetBeans IDE.....	260
15.8.1	Een eenvoudig voorbeeld .....	260
15.8.2	Radiobuttons en checkboxes .....	263
15.8.3	Menu's en dialogs.....	270
<b>HOOFDSTUK 16 BIJLAGEN .....</b>		<b>279</b>
16.1	Het gebruik van de API .....	279
16.1.1	Inleiding .....	279
16.1.2	Structuur van de API.....	280
16.1.3	Voorbeeld .....	282
16.2	Debuggen .....	285
16.2.1	Soorten fouten .....	285
16.2.2	Het programma stap voor stap uitvoeren .....	285
16.2.3	BreakPoints .....	286
16.2.4	Informatie verzamelen .....	286
<b>COLOFON.....</b>		<b>287</b>



# Hoofdstuk 1 Algemene situering van Java

---

## 1.1 Inleiding

Waar komt Java vandaan?

Begin 1990 werd bij Sun een ontwikkelteam samengesteld onder de leiding van James Gosling met als opdracht: maak een kleine, eenvoudige, robuuste programmeertaal om in te zetten bij de programmering van embedded microchips in TV's, wasmachines, afstandsbedieningen,...

Het moest dus een taal worden die op verschillende soorten hardware kon draaien.

C++ was geen optie wegens te complex, te gevaarlijk in de handen van minder-ervaren programmeurs, ...

Men zou dus een nieuwe oplossing bouwen, C++-like, (maar dan zonder de nadelen) en rekening houdend met de laatste stand van zaken in de OOP (Object Oriented Programming) wereld. Het werd **OAK**. Toen bleek dat er al een programmeertaal bestond met die naam, werd de taal herdoopt naar **JAVA** (genoemd naar hun favoriete koffiemark).

Het werd geen succes, de industrie evolueerde te traag en pikte onvoldoende in op de mogelijkheden van Java.

Midden van de jaren 90 werd echter het internet heel populair en zie, dit was de perfecte omgeving voor een product als Java: een client-server architectuur waarbij men rekening moest houden met een variatie aan (onbekende) client's: WinTel-systemen, Linuxsystemen, Mac's, ...

Java werd dan ook met open armen binnengehaald in de internetwereld en kende een enorme opbloei met **applets** (application-let: kleine applicaties die samen met de internetpagina gedownload worden bij de client en die daar uitgevoerd worden). In eerste instantie werden die applets gebruikt om statische pagina's te voorzien van interactie met de gebruiker, van animaties, ...

We zijn nu 10 jaar later en applets zijn intussen voorbijgestoken door Flash. Java heeft echter de tijd gekregen om zijn waarde te bewijzen en is intussen uitgegroeid tot een volwaardige programmeertaal, die nog altijd toepassingen heeft binnen het internet (applets, servlets, ...), die ook als 'general purpose language' gebruikt wordt en die (hoe langer hoe meer) toegepast wordt in apparaten (denk maar aan de Java-enabled GSM's, MP3-spelers, ...). Actueel kan men stellen dat het hoofdgebruik van Java zich situeert in server-side toepassingen waar dan, afhankelijk van het gebruikte materiaal, een front-end (met GUI – veelal HTML) aan gekoppeld wordt.

Deze cursus focust zich dan ook minder op het bouwen van een GUI maar meer op de mogelijkheden van OOP.



## 1.2 Kenmerken van Java

- Java is klein: hiermee wordt niet bedoeld dat het alleen geschikt is voor kleine toepassingen, maar dat de instructieset beperkt is. Alle overbodige franjes en opties zijn weggelaten.
- Java is objectgeoriënteerd: meegaand met de nieuwe en succesvolle manier van systemen bouwen heeft men Java ook van in den beginne ontworpen als een volwaardige OOP. Men moest geen compromissen sluiten om continuïteit met legacy-systemen te waarborgen.
- Java is algemeen toepasbaar: dankzij het beperkte aantal kleine bouwstenen kan men alles aan (denk maar aan de architectuur waar men met eenvoudige middelen als bakstenen, zand, cement, ... zowel eenvoudige bouwsels als de meest prestigieuze kathedralen, paleizen, ... kan optrekken en hoe kleiner de bakstenen, hoe mooier vormen men kan maken).
- Java is platformonafhankelijk: de taal is ontworpen om op een grote verscheidenheid van apparaten ingezet te worden. Java-programma's draaien dan ook op bijna alle computers, onder bijna alle besturingssystemen zonder dat er wijzigingen moeten aangebracht worden. Hoe dat gerealiseerd is, vertellen we later.
- Java is robuust: de voorbereiding (coderen en compileren) moet aan zeer strikte regels voldoen. Een programma dat die controles passeert, is (bijna zeker) foutvrij. Hiernaast is er ook in het concept ingebouwd dat een Java-programma nooit een operatingsysteem mag laten crashen. Zelfs als er runtime-fouten optreden, is het enkel het Java-programma dat stopt, de computer en alle andere programma's blijven draaien.
- Java heeft bibliotheken: een schare enthousiaste ontwikkelaars hebben zich op Java gestort en hebben een (steeds groeiende) verzameling van class-bibliotheken gebouwd. Als Java-programmeur word je geconfronteerd met een berg aan mogelijkheden. Gelukkig is er een uitgebreide help beschikbaar.

## 1.3 Evolutie van Java

De eerste versie Java 1.0 verscheen in 1996. Versie 1.1 met een aantal verbeteringen in 1997 en versie 1.2 (de eerste echt bruikbare, stabiele versie) in 1998. Vanaf versie 1.2 sprak men van het "Java 2 platform".

Nieuwe versies (1.3 in 2000, 1.4 in 2001) brachten vooral uitbreidingen. Aan de basis werd niet meer geraakt. Sinds februari 2005 hebben we versie 1.5 (ook gekend als Java 5). Vanaf december 2006 is versie 1.6 (gekend als Java 6) beschikbaar. Deze versie is 'open source'.

Hoe zit het eigenlijk met die nummering? Het product version number van java is 6 (vandaar java 6) en het developer version number is 1.6.0. Zo staat in de documentatie van java.

De cursus is gebaseerd op Java 6.



## 1.4 Varianten

Java bestaat in drie varianten:

- **JSE: Java Standard Edition.**  
Dit is de klassieke basis. Het gaat hier over de (uitgebreide) basisfuncties van Java: ong. 2900 classes.
- **JEE: Java Enterprise Edition.**  
De standaard uitvoering + alle bibliotheken voor zeer uitgebreide client-server applicaties. Hierin zitten meer dan 6500 classes.
- **JME: Java Micro Edition.**  
Een sterk afgeslankte versie om gebruikt te worden op 'kleine' apparaten als WAP-GSM, MP3-spelers, ...

Voor deze cursus hebben we voldoende aan JSE (vroeger J2SE genoemd).

## 1.5 Hoe realiseert men in Java de platformonafhankelijkheid?

In het vóór-Java tijdperk deelde men de programmeertalen in in twee categorieën: interpretertalen en compilertalen.

Alle talen worden geschreven in een source-taal die door mensen kan gelezen en begrepen worden. De computer echter kent alleen binaire code (0 en 1).

Er moet dus altijd een vertaling gebeuren tussen de source-code en de executable code.

Bij interpretertalen gebeurt die vertaling bij de uitvoering. De source-code wordt lijn per lijn gelezen, vertaald, en uitgevoerd.

Bij de compilertalen wordt de source-code op voorhand vertaald naar de uitvoerbare code en wordt bij uitvoering alleen nog de uitvoerbare vorm geladen en uitgevoerd.

### Voor- & nadelen

Interpretertalen	<p>Relatief traag: de source-code moet telkens vertaald worden.</p> <p>Gemakkelijk te debuggen. Men kan, met een debugger, bij uitvoering regel per regel volgen wat er gebeurt en eventueel ingrijpen.</p> <p>Onveilig: men moet de leesbare source-code verspreiden. Die kan gemakkelijk gekopieerd en/of gewijzigd worden.</p> <p>Flexibel: het programma kan tijdens de uitvoering zelf nog gewijzigd worden.</p> <p>Source-compatible: de programma-source is een normaal tekstbestand. Dit kan gemakkelijk gelezen worden door verschillende computers en indien daar een aangepaste vertaler (<b>interpreter</b>) aanwezig is, kan het ook uitgevoerd worden.</p>
Compilertalen	<p>Snel: de programma's worden op voorhand vertaald (gecompileerd) naar een geoptimaliseerde uitvoerbare code.</p> <p>Veilig: enkel de (binaire) uitvoerbare code wordt verspreid over alle computers die het programma moeten uitvoeren. Deze binaire code is voor de mens niet interpreteerbaar.</p>



	<p>Niet-overdraagbaar: de gegenereerde binaire code wordt aangemaakt in functie van één type processor en van één type operating systeem. Het programma kan alleen op dat soort computers uitgevoerd worden. Wil men meerdere varianten, dan zal men meerdere vertaalprogramma's (compilers) moeten hebben, ieder in functie van de target-machine. Dit impliceert ook dat bij wijziging, men opnieuw alle versies moet aanmaken (compileren).</p> <p>Niet flexibel: eens vertaald, ligt alles vast.</p>
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Wat doet Java?

Java is een mengvorm tussen beide mogelijkheden.

In eerste instantie is het een compilertaal maar waarbij gecompileerd wordt naar een virtuele computer. Men spreekt ook wel over **JVM**: de **Java Virtual Machine**.

Het resultaat van deze compilatie is geen uitvoerbare code (tenzij men de virtuele computer in hardware zou realiseren). Men noemt het **bytecode**.

Op iedere computer waar men deze programma's wil draaien moet dus (**in software**) die virtuele machine nagebootst worden. Binnen deze (softwarematige) virtuele machine wordt de bytecode behandeld als een interpretertaal: iedere bytecode-instructie wordt omgezet naar een voor die machine specifieke binaire code, worden voor de machine specifieke routines opgeroepen en wordt de instructie uitgevoerd.

Men noemt dit wel de **JRE**: de **Java Runtime Environment**.

De JRE is dus een (softwarematige) JVM + alle libraries, api's, ... die machinespecifiek zijn. Zo'n JRE is intussentijd beschikbaar voor de meeste computers (ook voor main-frames!).

Voor- & nadelen:

Relatief snel: het grootste deel van het vertaalwerk, syntax-checking, ... wordt op voorhand gedaan. Er is de laatste jaren ook veel geïnvesteerd in de uitvoering van de laatste fase: de interpreter. Men spreekt van JIT-compilers, HOT-SPOT compilers, ... die telkens een verdere optimalisatie brengen van het vertaalproces. Men kan dan ook stellen dat de huidige uitvoeringssnelheid die van volledig gecompileerde talen als C, C++, ... benadert.

Veilig: - enkel de (onleesbare) bytecode moet verdeeld worden, niet meer de source,

- het concept van de softwarematige JVM is dusdanig uitgewerkt dat er nooit interferenties kunnen komen met andere programma's, met het operatingsysteem, ...

Opmerkingen:

- Er zijn ooit pogingen geweest om de JVM in hardware te realiseren. Dit heeft echter geen succes gekend.
- Er bestaan compilers die Java-source vertalen naar echte uitvoerbare code. Dit is dan, zoals bij alle compilertalen, geen overdraagbare code.
- Dat dit concept succesvol is, bewijzen de concurrenten (Microsoft) die het geïmiteerd hebben met hun .net framework.



## 1.6 Wat hebben we nodig om Java te gebruiken?

### Om Java-programma's uit te voeren:

De JRE. Voor het uitvoeren van applets is er in de meeste browsers een JRE ingebouwd.

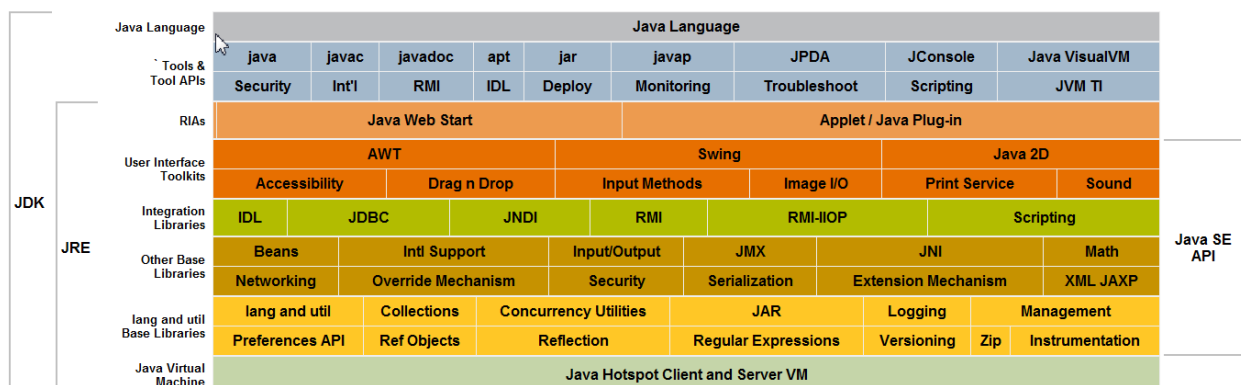
### Om Java-programma's te ontwikkelen:

#### - De JDK:

De **Java Development Kit**. Deze (gratis te downloaden) ontwikkelkit bevat de compiler, de JRE en alle bibliotheken die noodzakelijk zijn om Java-programma's te ontwikkelen. Hiernaast bevat de ontwikkelkit ook een aantal hulpprogramma's voor het debuggen, het genereren van documentatie, het testen van applets, ...

Waar van toepassing worden deze hulpprogramma's in de verdere cursus toegelicht.

Een overzicht:



De JRE bestaat uit een client-compiler (of interpreter) die als stand-alone programma of als plug-in (in browsers) kan aanwezig zijn. Daarbij komen alle libraries die de routines bevatten nodig om de applicaties te laten werken + de HotSpot runtime die voor de echte verwerking zorgt. Dit alles is client-specifiek! Er bestaan versies voor Windows, Linux, Mac, ...

De SDK (of JDK) bevat de JRE (specifiek voor de machine waarop gaat ontwikkeld worden) + een aantal programma's als: de compiler (`javac`), de debugger (`jdb`) en andere.

Bovenop dit alles kan een editor of IDE (Integrated Development Environment) gebruikt worden, bijv. NetBeans.

- **Een editor:** in principe is kladblok voldoende, maar er bestaan verschillende (gratis) dedicated editors die behoorlijk wat meer comfort geven dan kladblok. Voor deze cursus gebruiken wij NetBeans (op de site <http://www.netbeans.org/>, bij het

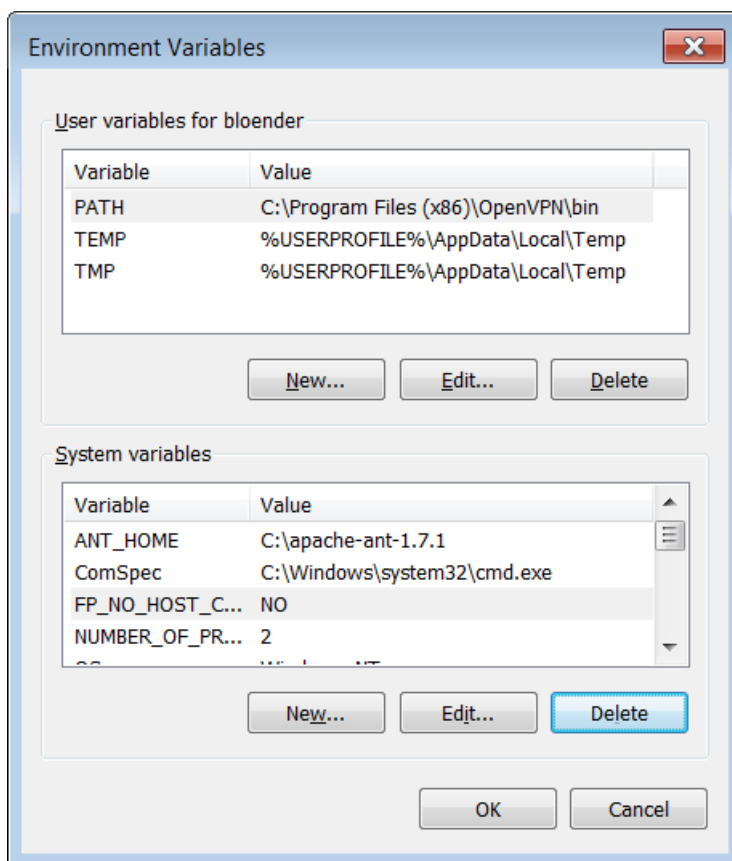


tabblad 'Docs & Support' vind je steeds de installatieprocedure – in functie van de juiste versie en van het operating systeem – en een tutorial 'Using NetBeans' – ook als PDF).

Wanneer de JDK geïnstalleerd is, is het gebruikelijk om de omgevingsvariabele JAVA\_HOME in te stellen. Deze omgevingsvariabele bevat het pad waar de JDK van java geïnstalleerd is. Default is dit C:\Program Files\Java\jdk1.6.0\_18 waarbij uiteraard het versienr van java kan verschillen!

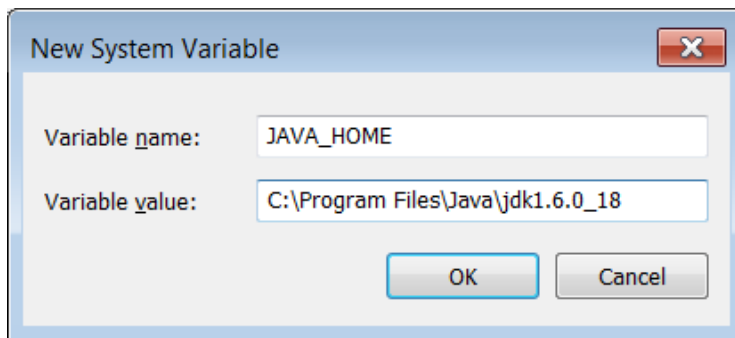
Het instellen van JAVA\_HOME doe je als volgt:

Via: Control Panel – System – Advanced system settings – knop *Environment Variables...* verschijnt er volgend scherm:

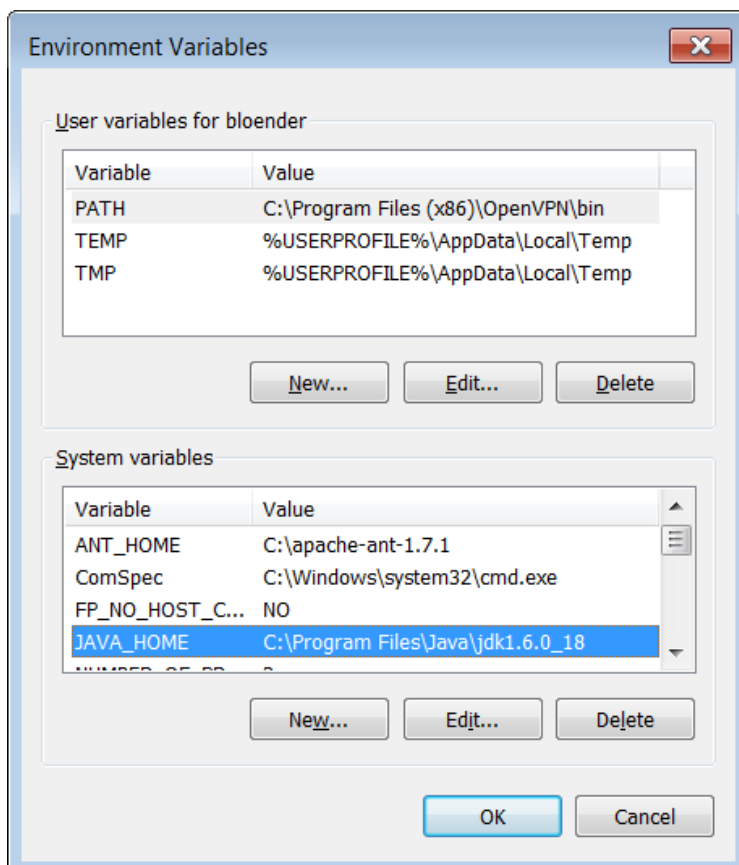


Bij de **System variables** kies je voor **New** en je vult het dialoogvenster als volgt in:





Je bevestigt met OK en je vindt vervolgens de nieuwe omgevingsvariabele in de lijst van Environment variables:



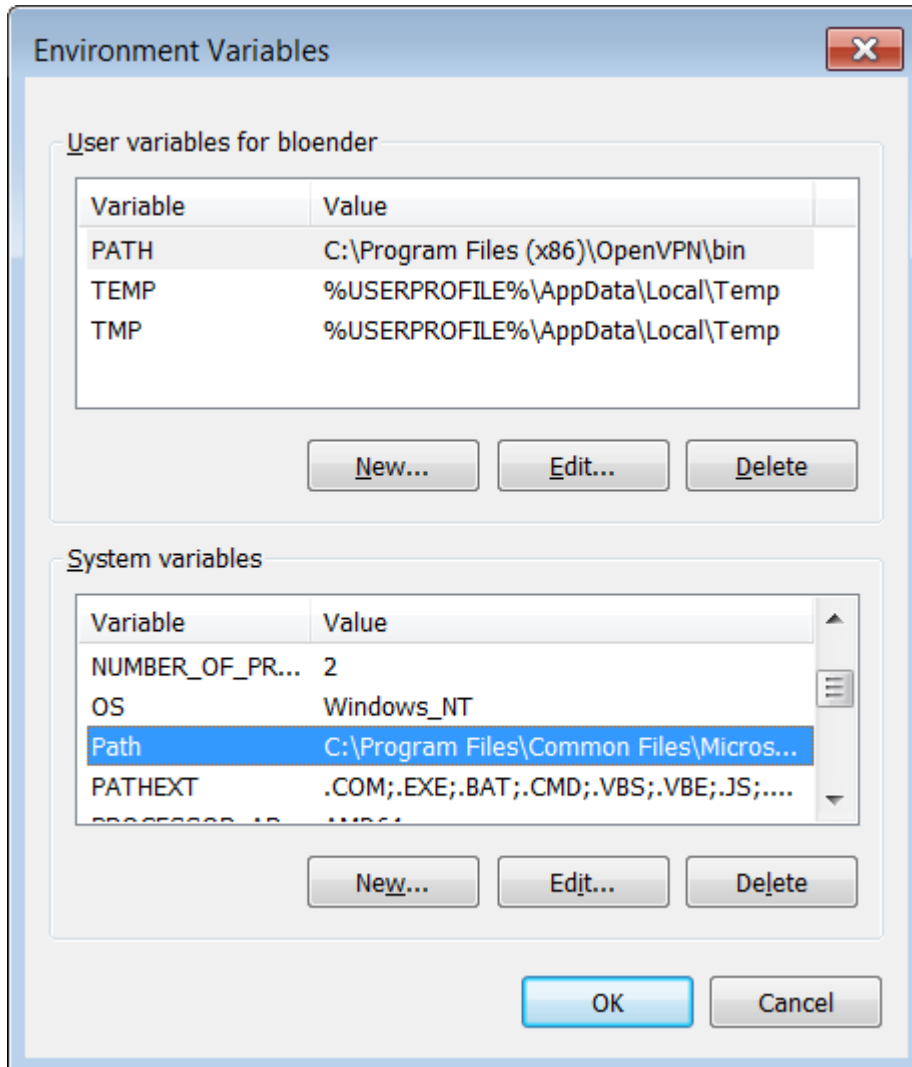
Tot slot voeg je de installatie van Java toe aan het pad, maar via de JAVA\_HOME variabele. Alle uitvoerbare files van de JDK staan in de bin-directory waar java geïnstalleerd is, dus in C:\Program Files\Java\jdk1.6.0\_18\bin.

De directory waar Java geïnstalleerd is, is ingesteld in JAVA\_HOME. Voeg daarom het volgende toe aan het path: %JAVA\_HOME%\bin.



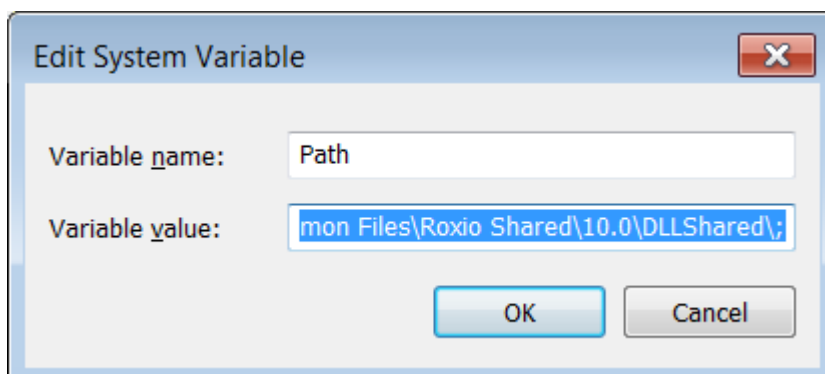
De %-tekens zorgen ervoor dat de waarde van JAVA\_HOME genomen wordt, dus C:\Program Files\Java\jdk1.6.0\_18. Dit wordt aangevuld met \bin.

Om dit te kunnen toevoegen aan het pad, moet je de omgevingsvariabele *Path* wijzigen. Ga terug naar het venster van de omgevingsvariabelen:

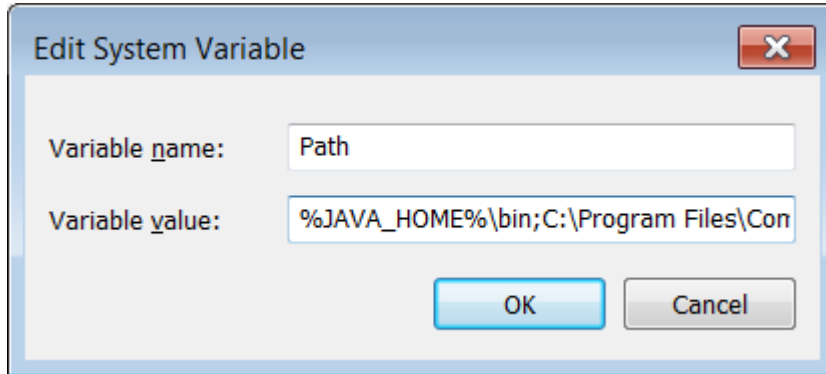


Selecteer de variabele **Path** uit de **System variables** en kies voor Edit...

Volgend scherm verschijnt:



Het is de bedoeling om `%JAVA_HOME%\bin` **VOORAAN** in te voegen bij het path, gescheiden met een `;` van datgene wat volgt. (Vooraan omdat er anders via de paden van windows een andere jre aangetroffen wordt).



Klik op OK om de wijziging op te slaan.

Vanaf nu bevat de systeemvariabele path de geïnstalleerde JDK voor java.

## 1.7 Java en GUI (Graphical User Interface)

Sinds de versie 1.1 kan men grafische gebruikers interfaces maken. Men ontwikkelde hiervoor de **AWT**: de **A**bstract **W**indows **T**oolkit.

Dit is een verzameling classes die beroep doet op het onderliggende grafische operating systeem om buttons, textfields, scrollbars, ... te genereren.

Probleem hierbij is dat Java bedoeld is om op een verscheidenheid van hardware en van operatingsystemen te draaien en dat men dus slechts kan gebruik maken van die 'dingen' die overal bekend zijn. Men kreeg dus de grootste gemene deler van wat in de verschillende systemen aanwezig was. Daarbij komt ook nog het fenomeen dat een button, textfield, ... in Windows er iets anders uit ziet dan in de KDE van Linux, dan in GNome van Linux, dan in de Mac, ...: men krijgt dus geen standaard look & feel.

Vanaf Java versie 1.2 kwam er naast de AWT ook **SWING** (aanvankelijk **JFC Java foundation classes** genoemd). Bij swing volgt men een ander spoor. Aangezien alle onderliggende (grafische) operatingsystemen in staat zijn om te 'tekenen', heeft men nu een aantal classes in het leven geroepen die een button, een textfield, ... gaan tekenen. Het voordeel is nu dat de look & feel gestandaardiseerd is en dat men niet beperkt is tot de ggd (grootste gemene deler) van de mogelijkheden.

De Java-versie waar we nu mee werken kent de twee mogelijkheden.



## 1.8 Enkele afkortingen verklaard

### 1.8.1 APPLET

Een kleine applicatie (application-let) die samen met een html-pagina geladen wordt vanaf een internetserver en die op de client-computer wordt uitgevoerd. Hiervoor beschikken de (meeste) browsers over een aangepaste plug-in.

Voordeel:

De pagina's worden dynamisch, interactie met de gebruiker is mogelijk.

Nadelen:

Het programmaatje moet klein zijn, zoniet lange laadtijden voor de pagina.

Binnen (sommige) browsers kan men de mogelijkheid om applets te draaien uitschakelen.

Er zijn enorme veiligheidsbeperkingen (en dus ook functionele beperkingen), een applet mag geen gebruik maken van de resources op de client-computer.

### 1.8.2 SERVLET

Een klein programmaatje dat op de server draait. Wordt vooral gebruikt om dynamisch HTML-pagina's te genereren. Een servlet is in de eerste plaats een Java-programma waarin als output HTML gegenereerd wordt.

### 1.8.3 JSP

Java server pages: eenvoudiger manier om server-side dynamische HTML-pagina's te genereren. Gemaakt naar analogie met ASP. In background worden JSP-pagina's nog altijd omgezet naar servlets. Een JSP is in de eerste plaats een HTML-pagina waarin Java wordt aangeroepen.

### 1.8.4 Java BEANS

De bedoeling was om een Java-alternatief te bieden voor visual basic componenten. Het zijn meestal grafische componenten (met eigenschappen en gedrag) die op een standaard manier gecodeerd zijn zodat iedereen ze op een eenvoudige wijze kan integreren in zijn applicaties. Veel grafische componenten (o.a. een swing-button) zijn gebouwd als een Java bean. Java beans worden toegepast bij de presentatie van de gegevens en dit gebeurt dus aan de client kant.

### 1.8.5 JDBC

Java DataBase Connectivity: de verzameling classes die een eenvoudige toegang tot databases mogelijk maakt.

### 1.8.6 EJB

Enterprise Java Beans: de verdere evolutie van de Java beans. EJB's draaien op de server. Zij zijn niet grafisch georiënteerd maar zullen eerder taken verrichten in databasemanipulatie, netwerkbeheer, ...

### 1.8.7 DESIGN PATTERNS

Dit zijn 'slimme' oplossingen voor complexe problemen die ons onder de vorm van abstracte beschrijvingen worden gegeven.



*“design patterns help you learn from others successes  
instead of our own failures”*

## 1.9 Java-libraries / packages / jar's / classpath

Zoals gesteld bevat Java duizenden classes, aangemaakt door honderden programmeurs. Om nu te vermijden dat er naamconflicten (dezelfde naam voor verschillende dingen) ontstaan, heeft men de techniek van de **packages** ingevoerd.

Een package is een verzameling logisch bij elkaar horende classes, die in een eigen directory / subdirectory – structuur worden opgeslagen. De volledige naam van een class bestaat dan uit de package-naam, een . (punt), de classnaam.

Kiest men de namen van de packages zorgvuldig, dan is er geen risico op naamconflicten. Een algemeen aangenomen afspraak hierover is dat de package-naam begint met de omgekeerde domeinnaam van het bedrijf: vb. `be.vdab`.

Van iedere class die opgenomen wordt in een specifieke package, begint de sourcecode met de instructie **package ...**.

Een class waarvan de code niet begint met een **package**-instructie, wordt opgenomen in een default package (dit wordt, zeker vanaf versie 1.5, ten zeerste afgeraden). Onder invloed van de XML-hype spreekt men ook binnen Java af en toe van **namespaces**.

Een **.jar** is een Java archief. Om de hoeveelheid code die bevat zit in de duizenden classes manipuleerbaar te houden, slaat men ze op in gecomprimeerde archieven.

Zowel de Java-compiler (`javac.exe`) als de Java-runtime (`java.exe`) kunnen die archieven lezen zonder ze eerst te moeten decomprimeren.

### **Classpath:**

Indien men in een java-programma refereert naar een externe class, waar gaat “Java” (beter de compiler) die class dan zoeken?

De basisclasses (alle classes uit de package `java.lang`) zijn altijd beschikbaar. Is het een andere class uit de (standaard) Java libraries, dan volstaat het om naar de class te verwijzen via de ‘fully qualified name’: `packagenaam.classnaam` of om in je java-programma een import te doen van de juiste package.

Is het een zelfgemaakte Java-class, dan werkt Java vergelijkbaar met DOS. In de eerste plaats wordt er gezocht in de actuele directory (en in alle subdirectories) en indien daar niet aanwezig, gaat men een pad af dat ingesteld wordt via het **classpath**. Het classpath geeft aan waar je andere zelfgeschreven classes vindt. Dit classpath is zeer belangrijk wanneer je java-programma's ontwikkelt en compileert aan de prompt (zonder editor zoals NetBeans of Eclipse). Dan moet het classpath opgegeven worden opdat de compiler alle classes die je gebruikt daadwerkelijk vindt. Maar vermits we gaan werken met een IDE, gaan we hier nu niet verder op in.



## Hoofdstuk 2 Variabelen en operatoren

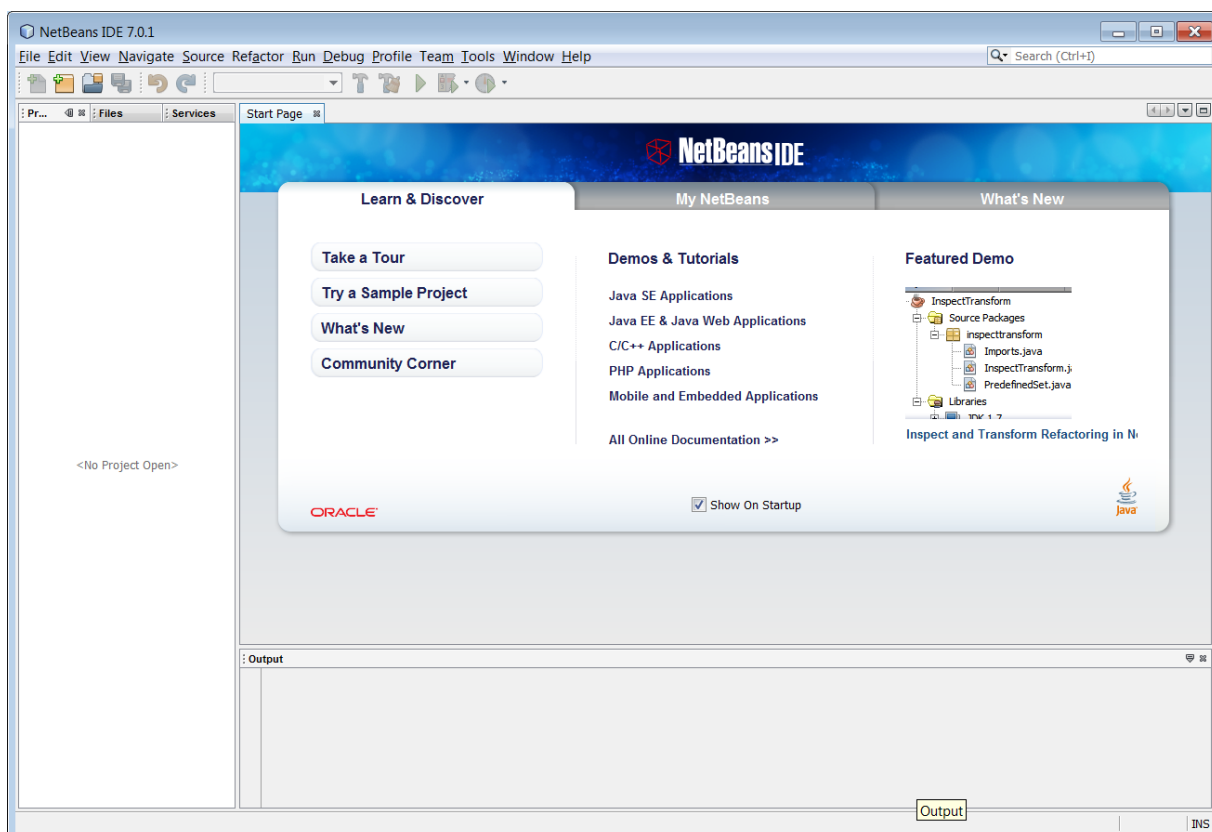
In dit hoofdstuk bekijken we de structuur van een eenvoudige Java-applicatie. We leren werken met variabelen van verschillende gegevenstypes en gebruiken allerlei operatoren. De volgorde van verwerking van deze operatoren komt aan bod alsook het gebruik van type casting. Dit alles bekijken we aan de hand van een voorbeeld uitgewerkt in NetBeans.

### 2.1 Een eenvoudig programma

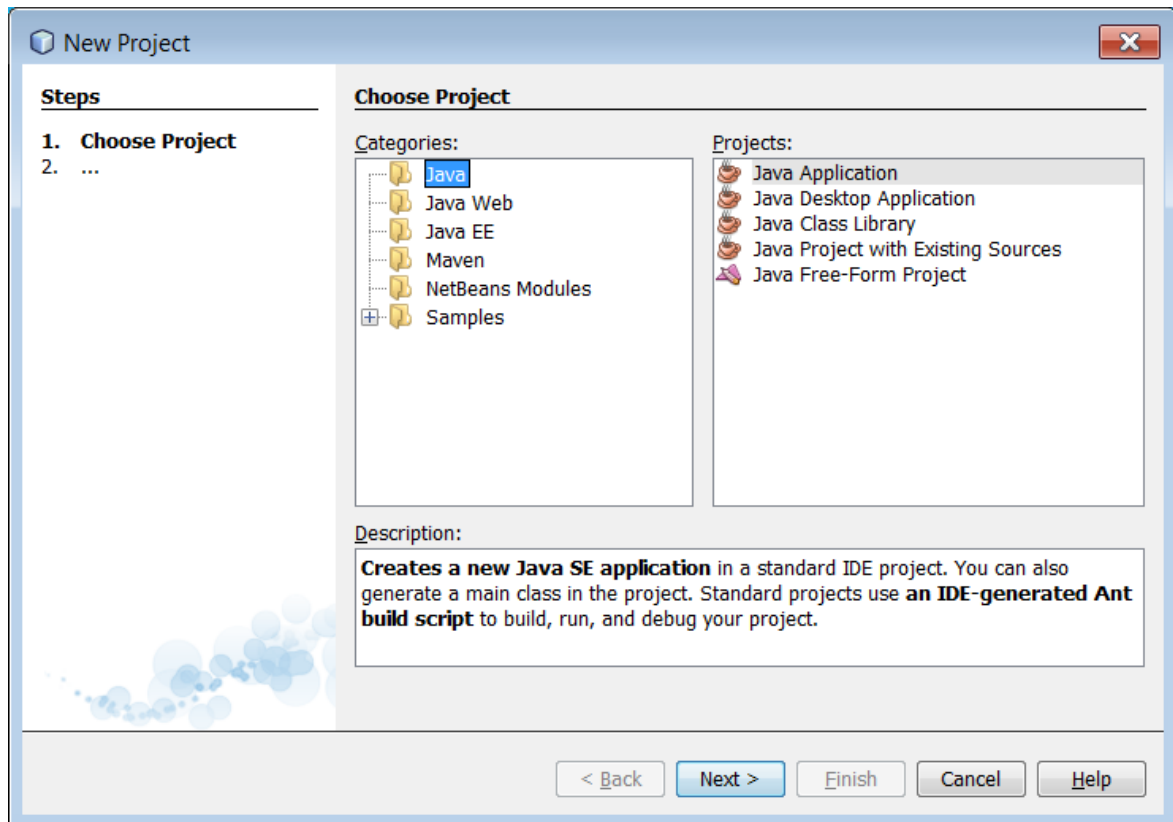
In deze paragraaf gaan we een eenvoudig programma maken met de NetBeans ontwikkelomgeving. Naarmate dit hoofdstuk vordert zal het programma ook groter en vooral zinnvoller worden en meerdere elementen van de Java programmeertaal bevatten.

We starten alvast met in NetBeans een framework te maken voor een Java-programma. Dit doen we aan de hand van de *New Project Wizard*.

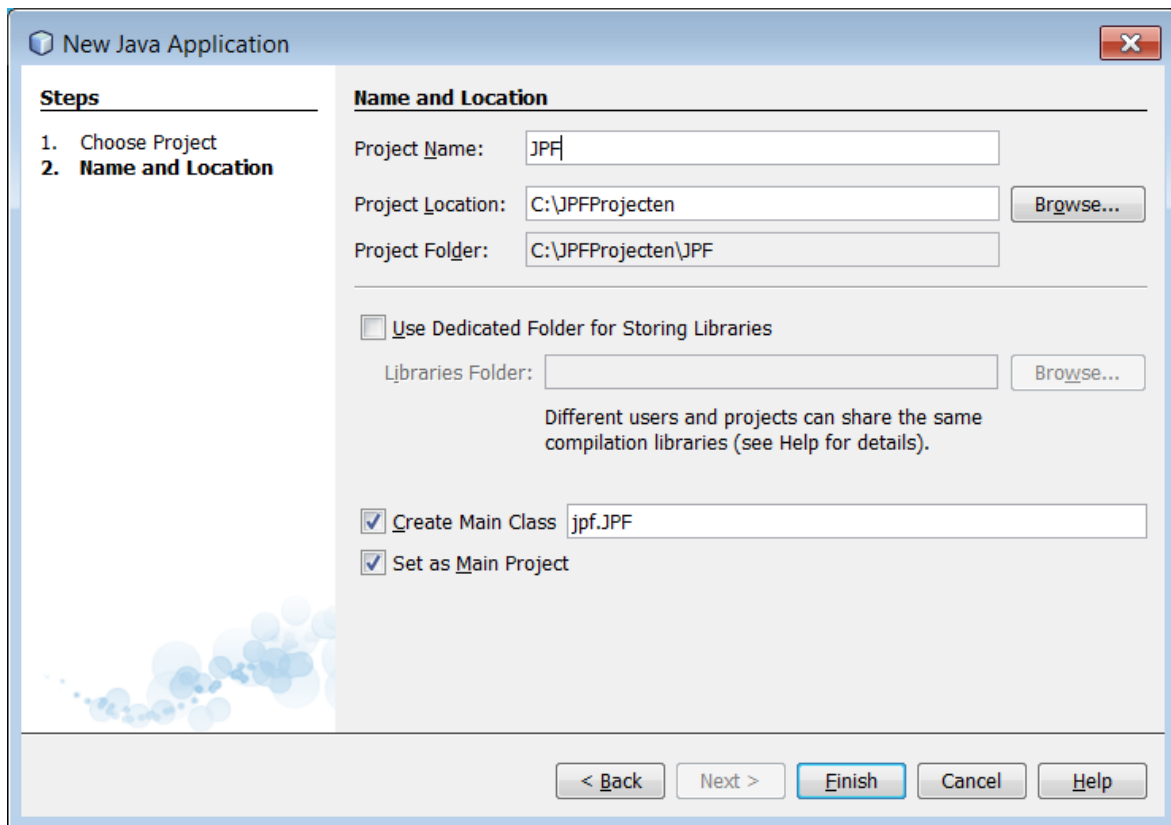
- Start NetBeans (hier versie 7.0) op. We krijgen volgend dialoogvenster te zien:



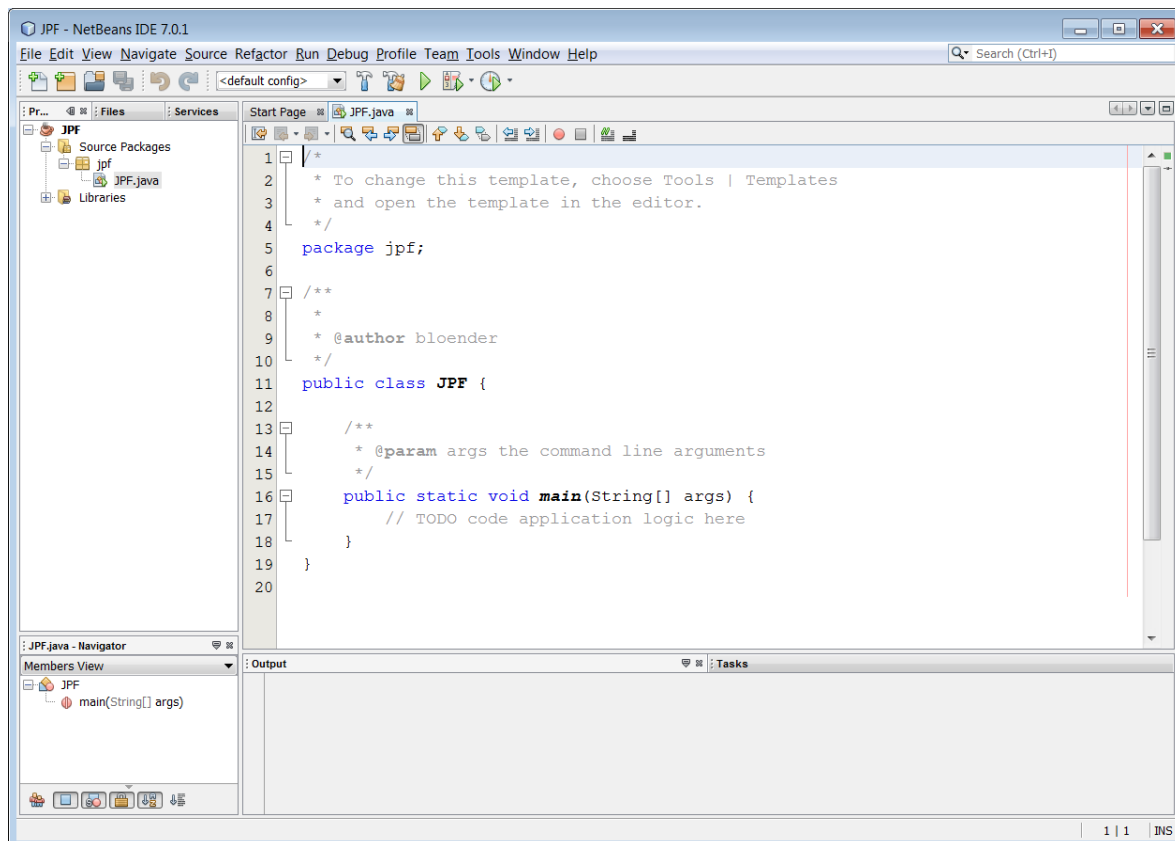
- Kies in het menu *File - New Project*, zodat je het volgende venster krijgt:



- We willen een standaard Java applicatie maken en dit vinden we terug onder de categorie Java. Klik dus op Java, dan op Java Application en daarna op Next.
- De wizard vraagt:
  - een naam voor het project: **JPF**
  - de plaats waar het project en bijhorende bestanden worden opgeslagen (in een folderstructuur die door NetBeans wordt aangemaakt in de opgegeven folder): kies voor **JPFProjecten**
  - en de naam van de folder waar de source zal worden opgeslagen: automatisch wordt dezelfde naam als die van het project voorgesteld.
- Vul dan het dialoogvenster in zoals hieronder is weergegeven:




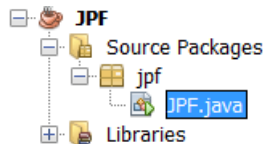
- Klik op Finish. Je eerste programma is alvast een feit en ziet er als volgt uit:





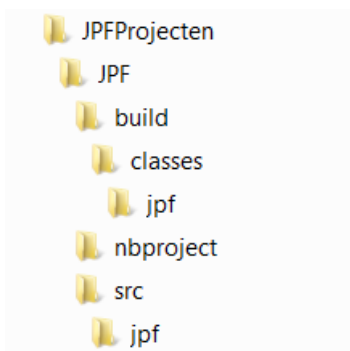
- Voorlopig bekijken we dit programma nog niet in detail. We voeren het eerst eens uit: kies daarvoor in het menu Run-Run Main Project (F6) om het programma te starten. In het output venster onderaan verschijnt in het groen:  
**BUILD SUCCESSFUL (total time: 0 seconds)**

Je kan ook bovenaan in de balk op de knop met de groene pijl klikken  om het programma te runnen, of links in het projectvenster rechtsklikken op het uitvoerbare programma en kiezen voor Run file:



NetBeans heeft de programmacode gecompileerd, gebuild en uitgevoerd. Uiteraard zie je voorlopig geen uitvoer, aangezien we nog geen eigen code hebben geschreven.

- We kijken in de map welke files er aangemaakt zijn. Zoek in Windows Verkenner de map **C:\JPFProjecten\JPF** op. Hier heeft NetBeans een aantal mappen gemaakt waaronder de mappen 'build' en 'src'. De volledige structuur zou er als volgt moeten uitzien:



In de map C:\JPFProjecten\JPF\src\jpf vind je de sourcefile JPF.java. Dit is een zuivere tekstfile waarin de code van ons programma staat. In de map C:\JPFProjecten\JPF\build\classes\jpf genereerde NetBeans een file JPF.class. Dit is de gecompileerde versie van het programma.



Je sluit het project door in het menu te kiezen voor File – Close “JPF”. Later kan je dit project snel weer openen door in het menu File te kiezen voor Open Recent Project – JPF.



## 2.2 Variabelen en literals

Als we programma's maken dan zullen we berekeningen willen uitvoeren en resultaten (tijdelijk) willen bewaren in variabelen. We maken alvast een goede start door eerst de verschillende soorten variabelen te bekijken.

### 2.2.1 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, *double* en *float* voor getallen met cijfers na de komma, *boolean* voor logische waarden en tenslotte *char* voor tekens. Het volgende tabelletje maakt duidelijk welke waarden variabelen van elk data type kunnen bevatten:

Type	Grootte	Inhoud
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
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
boolean	JVM-specifiek	De waarde true of false
char	2 bytes	Individueel teken zoals een letter, cijfer, leesteken of een ander symbool

Naast deze primitive data types kent Java zogenaamde *class types*. Het type van de variabele is dan een class. Dit kan een class zijn die je zelf gemaakt hebt (bijvoorbeeld een class *Coördinaat* bestaande uit de floats *x* en *y*) of een class uit één of andere library zoals bijvoorbeeld de class *String*, die een tekenreeks kan bevatten.

### 2.2.2 Variabelen, references en literals

Hoe maak je nu zo een **variabele** in een Java programma? Het maken of declareren van een variabele gebeurt door het type te vermelden en dan de naam van de variabele. Een declaratie is een Java-statement of anders gezegd een Java-opdracht en dient zoals alle statements afgesloten te worden met een punt-komma.

Enkele voorbeelden van declaraties:

```
byte aantalKinderen;  
double salaris;  
boolean gehuwd;  
String voornaam;  
Coördinaat doelwit;
```

---

<sup>1</sup> In deze cursus proberen we zoveel mogelijk engelse termen te gebruiken in zoverre het de leesbaarheid van de cursus niet verstoort. We hebben het dus over primitive data types en niet over primitieve gegevenstypes, over references en niet over referenties maar wel over variabelen en niet over variables.



De namen van variabelen beginnen met een kleine letter. De primitive data types beginnen met een kleine letter, terwijl de class types met een hoofdletter beginnen. Java is een hoofdlettergevoelige programmeertaal. Zorg dus voor een correct gebruik van hoofdletters en kleine letters!

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**. De eerste 3 regels code hierboven reserveren in het geheugen plaats voor een byte, een double en een boolean. De laatste 2 regels maken een soort pointer (een **reference**) die kan wijzen naar, in dit geval, een String of een Coördinaat. Je reserveert enkel het geheugen voor de referentievariabele waar later het geheugenadres van het String- of Coördinaat-object in geplaatst wordt. Geheugenruimte reserveren voor het String- of Coördinaat-object zelf doe je dan met het sleutelwoord **new**:

```
doelwit = new Coördinaat();
```

De code hierboven betekent dan: "Reserveer voldoende geheugenruimte voor een object Coördinaat en laat de reference doelwit wijzen naar deze gereserveerde geheugenruimte.

Voorlopig hebben onze variabelen nog geen waarde. Dikwijls zal je bij de declaratie van de variabele, deze meteen een waarde willen meegeven. 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 met tegelijk een initialisatie aan de hand van een literal:

```
byte aantalKinderen=0;  
double salaris=1978.45;  
boolean gehuwd=true;  
char letter='Q';  
String voornaam="Roland";
```

Voor integer en floating point literals gelden volgende regels :

- Elke integer literal wordt als een int beschouwd. Dus ook bijvoorbeeld het getal 100 dat eveneens in een byte past.
- Elk decimale literal (= ieder getal met een decimaal teken) wordt beschouwd als een double.



- Als je een geheel getal dat binnen de int-grenswaarden valt toch als een long wil laten beschouwen, dan moet je er de letter L achter plaatsen. Bijvoorbeeld: 123L is een long. Ook integer literals die groter zijn dan de maximum int-waarde moeten gevolgd worden door een **L** (of een **l**) zodat ze een long worden. Anders krijg je een foutmelding. Gebruik best een hoofdletter L omdat in sommige lettertypes er bijna geen onderscheid is tussen de letter l en het cijfer 1 wat tot verwarring kan zorgen.
- Wens je een geheel getal in octaal formaat te noteren dan laat je het getal door een 0 voorafgaan. De literal 01234 is het octale getal 1234. Voor hexadecimale notaties volstaat het 0x voor het getal te zetten. Voorbeeld: 0xFA en 0x9B zijn hexadecimale literals.
- Alle numerische literals met een (decimale) punt erin zijn doubles. Wil je dat de literal als een float wordt beschouwd dan moet je er een **F** (of **f**) achter plaatsen. Achter een double mag je altijd een **D** (of **d**) plaatsen als je vindt dat dit duidelijker is. Nogmaals: gebruik best een hoofdletter F of D.
- Je kan floats en doubles ook in exponentiële notatie noteren door een e of E tussen mantisse en exponent te plaatsen. Voorbeeld: 1.8e-13 of 0.9E77.

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 simpel".
- 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. Voorbeeld: "Hij zei: \"Hier volgt een \nnieuwe lijn.\""

### 2.2.3 Constanten

Sommige variabelen zijn helemaal niet variabel maar horen gedurende het ganse programma dezelfde waarde te behouden. In de meeste programmeertalen worden dit constanten genoemd, in Java heeft men het over final variables.

Om een constante aan te maken creëer je gewoon een variabele maar je plaatst het keyword *final* net voor het data type.

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.

## 2.2.4 Enumeration

De *enumeration* is een nieuw data type in Java (sinds Java 5). Het is een opsommingstype. Dit data type wordt beschreven aan de hand van een geordende opsomming van waarden. Voorbeelden van enumerations zijn bijvoorbeeld de namen van de dagen, maanden of seizoenen, de opsomming van een set kleuren, de namen van werelddelen,...

Enkele voorbeelden van enumerations :

```
enum DagenVanDeWeek {MAANDAG, DINSDAG, WOENSDAG, DONDERDAG, VRIJDAG,
    ZATERDAG, ZONDAG}

enum Seizoenen {LENTE, ZOMER, HERFST, WINTER}
```

De individuele waarden die het data type kan aannemen worden *enumeration constants* genoemd. Andere waarden dan deze die tussen de accolades zijn opgesomd, kan het data type niet aannemen.

Aangezien de beschrijving van een *enum* een typedefinitie is, hoeft er geen puntkomma achter de lijn code te staan.

De declaratie (en initialisatie) van een variabele van een bepaald enumeration type is als volgt :

```
DagenVanDeWeek kuisdag = DagenVanDeWeek.VRIJDAG;
Seizoenen favorieteSeizoen = Seizoenen.LENTE;
```

De beschrijving van een enumeration mag niet lokaal (dus niet binnen een method) gebeuren. Wat dit precies betekent wordt pas in een later hoofdstuk duidelijk. Onthou voorlopig dat een enumeration eveneens een data type is, hoe je een enumeration definieert, declareert en initialiseert.

## 2.2.5 Identifiers

Ook voor het gebruik van namen van variabelen, zogenaamde identifiers, bestaan enkele regels :

- De naam van een variabele mag beginnen met een letter, een underscore-teken ( \_ ) of een dollarteken (\$).
- Een identifier is case sensitive.

Ook hier heeft men een conventie: namen van variabelen (en van methods) beginnen met een kleine letter en worden verder in "Camelcase" geschreven: iedere nieuw woord begint met een hoofdletter. Bijvoorbeeld: hijDieNietGenoemdMagWorden. Underscores ( \_ ) worden eigenlijk niet meer gebruikt.



### 2.2.6 Statements en commentaar

Tijd om nog eens naar ons voorbeeldprogramma terug te keren. We gaan een beetje code toevoegen waarin we mijn bodymassindex berekenen en uitvoeren. Ook bekijken we de rest van de code even in detail.

Zoals elke goede programmeur plaatsen we de nodige commentaar bij onze code.

- *Om mijn bodymassindex te berekenen hebben we een aantal variabelen nodig: mijn lengte, mijn gewicht en het resultaat stoppen we in de variabelen gewicht, lengte en bmi. Voeg de 3 statements uit onderstaande afbeelding toe aan je programma net na de regel*

*//TODO code application logic here.*

```
public static void main(String[] args) {  
    // TODO code application logic here  
    float gewicht = 89.9F;  
    float lengte = 1.86F;  
    float bmi;  
}
```

- *Hoewel we pas in een volgende paragraaf allerhande operatoren bekijken, berekenen we hier alvast de bmi door het gewicht te delen door het kwadraat van de lengte. We voegen nog een regel code bij :*

```
public static void main(String[] args) {  
    // TODO code application logic here  
    float gewicht = 89.9F;  
    float lengte = 1.86F;  
    float bmi;  
  
    bmi = gewicht / (lengte * lengte);  
}
```

- *Tenslotte voegen we nog een regel code toe om het resultaat uit te voeren naar het scherm :*

```
public static void main(String[] args) {  
    // TODO code application logic here  
    float gewicht = 89.9F;  
    float lengte = 1.86F;  
    float bmi;  
  
    bmi = gewicht / (lengte * lengte);  
  
    System.out.println("Mijn bodymassindex is " + bmi);  
}
```

Met de `System.out.println()`-functie kunnen we uitvoer op het scherm zetten. We geven een stuk tekst weer samen met de uitkomst die in de variabele `bmi` is opgeslagen.

- *Voer het programma uit door op F6 te drukken. Of ik wel groot genoeg ben voor mijn gewicht of gewoon heel veel spieren heb, zie je in het outputvenstertje.*

Een beetje uitleg over de volledige structuur van het programma nu. In ons programma zie je heel wat regels in licht grijs met één of meerdere sterretjes erin. Dit zijn allemaal commentaarregels. Commentaarblokken beginnen met `/*` en eindigen met `*/`. Alles wat tussen deze tekens genoteerd staat, is commentaar. Zo vind je 4 commentaarblokken in onze code.

Een andere manier om commentaar te noteren, vind je net boven de variabele-declaraties. Alles wat op een regel **na de tekens `//`** komt, is eveneens commentaar.

Tenslotte kent Java ook documentatiecommentaar. De bedoeling is al je classes en methods via een tool in een set html-pagina's te documenteren en te beschrijven. Dit soort commentaar bestaat uit een beginregel die bestaat uit de tekens `/**`, één of meerdere regels commentaar die beginnen met een `*` en tenslotte een laatste regel die enkel bestaat uit `*/`. Een voorbeeld van documentatiecommentaar vind je net voor de main-functie en net voor de classbeschrijving in ons voorbeeldprogramma. Je kan de set html-pagina's in NetBeans laten genereren door in het menu Run - Generate Javadoc (JPF) te kiezen.

- *Voeg bij wijze van oefening commentaar toe bij de variabele-declaraties, bij de berekening en bij de uitvoer.*



Een kleine tip bij het gebruik van commentaarblokken: je kan testcode heel makkelijk aan- of uitschakelen door er net vóór de tekens `/*` te zetten en er net na de tekens `*/`. Door de eerste `/` weg te laten wordt alles commentaar.

```
/*  
getal = 5;  
invoer = "niets";  
*/
```

In bovenstaande code worden de twee statements uitgevoerd.

```
/*  
getal = 5;  
invoer = "niets";  
*/
```

Door telkens de eerste `/` te wissen staan beide statements in commentaar.

Wie NetBeans gebruikt kan ook gewoon regels code in commentaar zetten via het menu-item Source-Comment (Ctrl+Shift+T). De omgekeerde beweging gebeurt via Source-Uncomment (Ctrl+Shift+D).

Tot zover de commentaar in ons programma. Nu een woordje uitleg over de code in dit programma. Bovenaan het programma zorgt de regel `package jpf;` er voor dat de class `JPF.class` in een package met naam `jpf` komt.

Eronder staat een regel `public class JPF {` met eronder een aantal regels code die geïndenteerd zijn en tenslotte een `}`. Alles wat tussen accolades staat noemt men een *block*. In dit blok vind je de beschrijving van een class met de naam `JPF`. NetBeans noemt deze class standaard naar de naam van het project (dit is gebeurd tijdens de creatie van het project met de wizard), maar je zou deze class evengoed anders kunnen noemen.

In het block staat een publieke statische functie: `public static void main(String[] args)`. Het is de bedoeling om hier uitvoerbare code te schrijven. Deze functie wordt als eerste uitgevoerd wanneer je het programma runt. Er kunnen eventuele argumenten als parameter worden meegegeven.



## 2.3 Operatoren

In deze paragraaf gaan we wat dieper in op het gebruik van allerlei *operatoren* in *expressies*. Onder een expressie verstaan we een opdracht of een berekening die een resultaat oplevert. Dit resultaat kan dan toegekend worden aan een variabele.

### 2.3.1 Soorten operatoren

We bekijken eerst volgende vijf soorten operatoren: *rekenkundige operatoren*, *toekeningsoperatoren* (assignment operators), *unaire operatoren*, *vergelijkingsoperatoren* en *logische operatoren*.



Unaire operatoren zijn operatoren die slechts één operand gebruiken, dit in tegenstelling tot de binaire waar altijd twee operands nodig zijn en de ternaire operator die 3 operands nodig heeft.

### Rekenkundige operatoren

In ons voorbeeldprogramma maakten we reeds gebruik van twee rekenkundige operatoren: de vermenigvuldiging (\*) en de deling (/). Daarnaast kent Java ook de optelling (+), de aftrekking (-) en de restbepaling bij gehele deling of modulus (%). Elk van deze operatoren neemt twee operanden. Het teken voor de aftrekking kan ook als unaire operator worden gebruikt om een getal negatief te maken.

### Assignment operator

Over de assignment operator (=) hoeft weinig extra verteld te worden, behalve misschien het volgende. 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 hebben uiteindelijk zowel `a` als `b` de waarde 5.

### 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++`). 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. De increment verhoogt de waarde met 1, de decrement vermindert de waarde met 1.



Voorbeeld:

```
int a = 2;
int b = a++ * 3;

int c = ++a * 3;
```

// b wordt 6! Eerst 2 en 3 met elkaar  
// vermenigvuldigen en aan b toekennen,  
// daarna wordt a verhoogd, a is nu 3  
// c wordt 12: eerst wordt a verhoogd (wordt 4)  
// daarna wordt er vermenigvuldigd met 3

## Vergelijkingsoperatoren

In Java kan je gebruik maken van 6 vergelijkingsoperatoren: kleiner dan (<), groter dan (>), kleiner dan of gelijk aan (<=), groter dan of gelijk aan (>=), gelijk aan (==) en niet gelijk aan (!=). Het resultaat van een vergelijking is een boolean.

Het gebruik van vergelijkingsoperatoren werkt dus zoals verwacht indien we ze gebruiken met primitieve datatypes. Anders wordt het wanneer ze gebruikt worden met objecten. Wat betekent groter dan tussen twee objecten? Voor objecten worden de referentievariabelen met elkaar vergeleken. Dit levert echter zelden het gewenste vergelijkingsresultaat op. Daarom gaan we bij objecten andere middelen gebruiken om te vergelijken. Meer hierover in een later hoofdstuk.

## Logische operatoren

Je kan meerdere vergelijkingen combineren met de logische operatoren en (& en &&), of (| en ||), exclusieve of (^) en de logische niet (!). Bij de eerste twee logische operatoren zijn er twee schrijfwijzen. Eén enkele & of | ofwel een dubbele. Bij de dubbele notatie wordt de evaluatie van de expressie gestopt van zodra de uitkomst met zekerheid gekend is. De exclusieve of levert true op als beide operanden een verschillende logische waarde hebben. De logische not (!) keert een logische waarde om.



Voorbeeld:

```
int a = 7;
int b = 3;
boolean c = 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 geïncrementeerd ! b blijft = 3

boolean d = a < 10 | ++b == 4;

// Het eerste deel van de expressie is true, het tweede deel eveneens
// want b wordt eerst geïncrementeerd tot waarde 4
```



Operatoren zoals binaire shift operatoren en het gebruik van rekenkundige operatoren op char types bespreken we hier niet omwille van hun heel specifieke karakter. Meer over dit soort operatoren o.a. in "Ivor Horton's Beginning Java 2" op pagina 70 en 61.

### 2.3.2 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 vergelijingsoperatoren 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: xyz++, xyz--
	Unaire operatoren in prefix notatie: ++xyz, --xyz, !
	Typeconversie: (nieuwtype)xyz
	*, /, %
	+ -
	Binaire shift-operatoren (<<, >>)
	Vergelijingsoperatoren <, <=, >, >=
	Gelijkheidsoperatoren ==, !=
	& (logische en)
	exclusieve or ^
	(logische of)
	&& (logische en)
	(logische of)
	De conditionele operator ? :
Laag	De toekenningsoperatoren: =, +=, -=, *=, /=, %=



Voor een meer gedetailleerde beschrijving van de prioriteitsregels verwijzen we naar het boek "Ivor Horton's Beginning Java 2" pagina 80.

### 2.3.3 Type casting

Aan de hand van allerlei operatoren kunnen we nu berekeningen gaan uitvoeren op diverse gegevenstypes. Maar van welk type is het resultaat van een bewerking met twee verschillende types? Kan je het type van een variabele omzetten in een ander type? Gelukkig wel en de term ervoor is *type casting*.



Eigenlijk kunnen er zich twee situaties voordoen. Ofwel wil je een variabele omzetten naar een type waar meer opslagmogelijkheden zijn. Bijvoorbeeld van een byte naar een int of van een float naar een double. Dus in de tabel op pagina 16 in de neerwaartse richting. Ofwel converteer je naar een beperkter type met mogelijk een verlies aan precisie.

In de eerste situatie is er geen specifieke actie van de programmeur vereist. Java neemt de waarde van de variabele van het beperktere type en stopt deze waarde in de andere variabele. Men noemt dit een impliciete cast.

In het andere geval, dus als je wil converteren naar een beperkter type moet je dit aangeven met een expliciete cast. Er is namelijk wel verlies aan precisie als je converteert van een floattype naar een integertype. 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 voor de bewerking of de variabele te plaatsen: (doeltype)waarde. Voorbeelden: (int)a, (byte)(17\*2).

### 2.3.4 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. Een voorbeeld hiervan heb je reeds gezien op pagina 20 waar we een string en een float samenvoegden.

Tot slot nog even vermelden dat je ook hier met de verkorte notatie += kan werken om een string aan een andere string toe te voegen. Voorbeeld:  
boodschappenlijst += "de krant";

Meer over strings in het aparte hoofdstuk over strings.

## 2.4 Praktijkvoorbeelden

Tot zover de theorie. Tijd voor wat praktijk.

- We keren terug naar ons voorbeeld in NetBeans. Als eerste voorbeeld zetten we de gemiddelde lichaamstemperatuur om van graden Celsius naar graden Fahrenheit. Hiertoe declareren we eerst twee floats: gemLichTempCel en gemLichTempFahr. Eerstgenoemde float initialiseren we meteen op 37.0F. Vervolgens berekenen we de temperatuur in graden Fahrenheit en voeren we uit naar het scherm. De code:

```
float gemLichTempCel = 37.0F;
float gemLichTempFahr;
gemLichTempFahr = gemLichTempCel * 9 / 5 + 32;

System.out.println("Gemiddelde lichaamstemperatuur in  ←
graden Celsius: " + gemLichTempCel);
System.out.println("Gemiddelde lichaamstemperatuur in  ←
graden Fahrenheit: " + gemLichTempFahr);
```

In bovenstaand voorbeeld vermenigvuldigen we de temperatuur in graden Celsius met 9. Dit is een bewerking tussen een float en een int-literal. Het resultaat is dan een float. Vervolgens delen we deze float door opnieuw een int-literal wat ons terug een float oplevert en daar

tellen we dan een int-literal bij op. Het eindresultaat is een float dus krijgen we geen waarschuwing als we dit resultaat toekennen aan `gemLichTempFahr`.

In de bovenstaande codelijnen merkte je een zwart pijltje. Hiermee geven we in deze cursus aan dat de code om layout redenen op een volgende lijn gezet is.

- In een volgend voorbeeld controleren we of een opgegeven btw-nummer klopt. Om geldig te zijn moet je het getal nemen gevormd door de eerste 7 cijfers en dit delen door 97. De rest van deze deling moet afgetrokken worden van 97 en dit resultaat moet dan gelijk zijn aan de laatste 2 cijfers van het BTW-nummer. De code:

```
int btwNummer = 213252520;           (1)
int deeltal = btwNummer / 100;        (2)
byte rest = (byte)(deeltal % 97);      (3)
byte controle = (byte)(btwNummer % 100); (4)

System.out.println(controle == 97-rest); (5)
```

- (1) Een btw-nummer past in een int dus beginnen we met het declareren en initialiseren van een int `btwNummer`.
- (2) Vervolgens bepalen we het getal dat door de eerste 7 cijfers wordt gevormd. We voeren daarvoor een gehele deling uit door 100, dwz. dat het resultaat van deze deling het gedeelte geeft van de uitkomst van de deling voor de komma. Het resultaat wordt in een int `deeltal` gestopt.
- (3) We moeten nu de rest bepalen van dit getal bij deling door 97. Bij wijze van oefening, stoppen we deze rest in een byte. Als we de code

```
byte rest = deeltal % 97;
```

schrijven dan krijgen we in de marge een kruisje te zien. Als je de muiscursor boven dit kruisje plaatst, krijg je de uitleg:

```
possible loss of precision
found   : int
required : byte
```

Het resultaat van de modulus bewerking met twee int-operanden levert een int op en die proberen we in een byte te stoppen. Omdat we per se het resultaat in een byte willen, moeten we de bewerking casten naar een byte.

- (4) Op een gelijkaardige manier bepalen we het getal dat met de 2



laatste cijfers van het btw-nummer kan gevormd worden.

- (5) Tenslotte vergelijken we dit getal met het verschil tussen 97 en de restbepaling.
- Nu is het jouw beurt: test of het rekeningnummer 737524091952 een correct rekeningnummer is. Je kan dit testen door het getal gevormd door de eerste 10 cijfers te nemen en te delen door 97. De rest van deze deling zou gelijk moeten zijn aan het getal gevormd door de 2 laatste cijfers van het rekeningnummer. Gebruik bij wijze van oefening zo klein mogelijke variabelen. Dus niet overal long waar je even goed (of beter) een byte zou kunnen gebruiken en controleer of je geen foutmeldingen krijgt in de marge.



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

Strings worden verder ook in detail behandeld in Hoofdstuk 7: Strings.

## 2.5 User-input

Sinds Java 2 bestaat er een zeer handige manier om input te vragen aan de user, namelijk met een Scanner-object. Hoe dat gaat tonen we meteen a.d.h.v. een voorbeeldje :

```
System.out.print("Geef een getal: ");  
Scanner sc = new Scanner(System.in);  
int getal = sc.nextInt();
```

In bovenstaand voorbeeld staan een heel aantal zaken die pas in de komende hoofdstukken zullen uitgelegd worden. We geven toch nu reeds een kleine verklaring van deze code. Je toont eerst aan de gebruiker wat er precies moet ingetikt worden. Je maakt dan een nieuw Scanner-object met de naam `sc` en je geeft aan waar de invoer vandaan moet komen: `System.in`, het keyboard dus. Met `sc.nextInt()` vraag je een int op vanaf het zonet opgegeven invoermedium.

Ook voor de andere data-types bestaan er specifieke methods (`nextBoolean()`, `nextFloat()`,...). Een String vraag je op met `next()`.

Opgelet: vergeet niet een import te doen van `java.util.Scanner`. Dat doe je door bovenaan de code (tussen de regel `package ...` en de regel `public class Main { }`) de volgende lijn code te tikken:

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

Meer over imports in Hoofdstuk 9.



## 2.6 Oefeningen



Maak oefeningen 1 t.e.m. 4 uit de oefenmap.



## Hoofdstuk 3 Arrays

In het voorbije hoofdstuk leerden we werken met variabelen van allerlei types. In dit hoofdstuk bekijken we reeksen of arrays van primitive data types en van classes.

### 3.1 Arrays van primitive data types of strings

Arrays hebben twee belangrijke eigenschappen. Het zijn vooreerst geordende reeksen van genummerde items. Je kan dus elk element uit een array apart benaderen via de naam van de array en het volgnummer van het item. Daarnaast moeten alle elementen van de array van hetzelfde type zijn. Dus allemaal van het type int, double, String,...

Om arrays te kunnen gebruiken, dienen we die net zoals gewone variabelen eerst te declareren. Een arraydeclaratie heeft de volgende syntax :

```
type naamarray[];  
ofwel  
type [] naamarray;
```

Zoals je ziet zijn er twee mogelijke schrijfwijzen. **De tweede geniet de voorkeur.**

Enkele voorbeelden :

```
byte[] lottogetallen;  
float[] lonenDecember;  
double[] metingen;  
String[] voornamen;  
char[] s;
```

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



#### Opgepast !

```
double p[], q;
```

Hier is enkel **p** een reference naar een array, **q** is een primitive type.

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

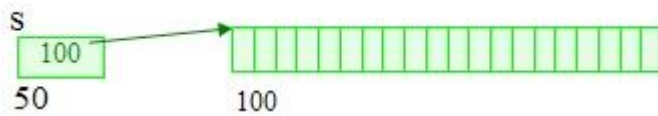
```
naamarray = new type [aantalelementen];
```

Voorbeelden :

```
lottogetallen = new byte [7];  
voornamen = new String [2];  
s = new char [20];
```



Grafisch voorgesteld :



Geheugenplaats 50 is gereserveerd voor de arrayvariabele *s* die wijst naar geheugenplaats 100 vanaf waar er 20 chars kunnen gestockeerd worden. 100 is dus het adres van het eerste element van de array.

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

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

Opgelet: ook in dit geval 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. Opgelet: de volgnummers van de elementen gaan van 0 tot en met *n*-1 als *n* het aantal elementen is.

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

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

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

```
arraynaam.length
```

Wil je nu bijvoorbeeld het laatste lottogetal veranderen dan kan dat met de volgende code :

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



De grootte van een array kan je niet wijzigen. Er bestaat toch een oplossing om aan een array één of meerdere elementen toe te voegen :

- Je maakt een nieuwe, grotere array, uiteraard van hetzelfde datatype.
- Met de method *arraycopy* vul je de grotere array (gedeeltelijk) op met de inhoud van de kleine array. Je kan het ook zonder method *arraycopy* doen, namelijk element per element de inhoud toewijzen aan de arrayelementen van de grotere array.
- Vervolgens laat je de reference van de kleine array verwijzen naar de grotere array en de reference naar de grotere array naar null (nergens).

De code :

```
int[] kleine = new int[6];  
int[] grotere = new int[kleine.length+1];
```



```
//kopiëer de inhoud van de kleine tabel in de grotere
System.arraycopy(kleine,0,grotere,0,kleine.length);

//laat de reference kleine naar grotere wijzen
//en grotere naar null
kleine = grotere;
grotere = null;
```

De method `arraycopy` (tesamen met duizenden anderen) maakt deel uit van de `java-libraries`. Om nu precies te weten hoe zo'n method werkt, welke parameters er moeten meegegeven worden en welk resultaat je kan verwachten moet je gaan bekijken in de standaard API-documentatie van `java`. Meer uitleg over het gebruik van de de standaard API vind je in een bijlage.

### 3.2 Arrays van classes

Hoewel we pas in een later hoofdstuk de classes in detail gaan bestuderen, bekijken we hier al even hoe we een rij van classes kunnen maken. De declaratie van een array van classes verschilt niet van deze van primitive data types of `Strings`. Een voorbeeld :

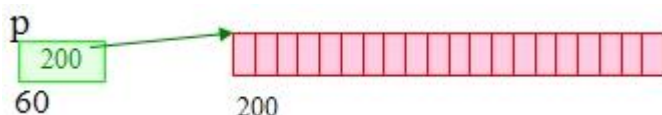
```
Coordinaat p[];
ofwel
Coordinaat[] p;
```

Ook nu hebben we nog geen lijst maar wel een reference `p` die kan verwijzen naar een lijst. Deze reference kunnen we laten verwijzen naar een bestaande array van coördinaten of naar een nieuwe lijst. Bijvoorbeeld :

```
p = new Coordinaat[20];
```

In de bovenstaande code hebben we een array van references naar instances van de class `Coordinaat` gemaakt. De references verwijzen voorlopig nog nergens naartoe.

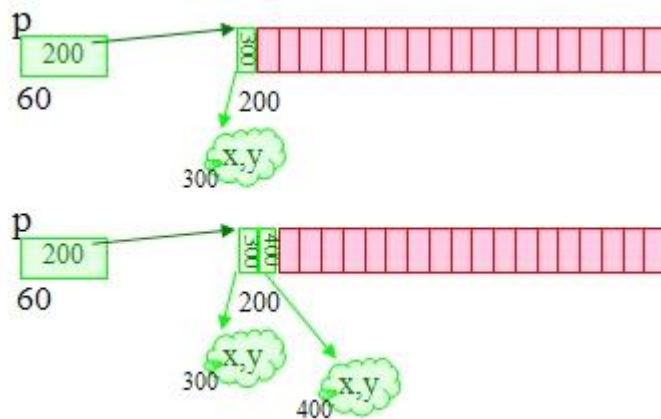
Schematisch :



In de figuur zien we een arrayvariabele `p` op geheugenplaats 60 die verwijst naar een array van references naar instances van de class `Coordinaat` vanaf geheugenplaats 200.

Je kan nu de references naar een nieuwe of een reeds bestaande instance van de class `Coordinaat` laten verwijzen. Voorbeeld :

```
p[0] = new Coordinaat();
p[1] = q;
```



In bovenstaande figuur zien we opnieuw de arrayvariabele `p`. Na de eerste instructie is er een nieuwe instance van de class `Coordinaat` aangemaakt en het eerste element van de array verwijst hiernaar.

Het onderste deel van de afbeelding is het resultaat na de tweede instructie. Het tweede element van de array verwijst naar geheugenpositie 400. Dit is de plaats waarnaar de reference `q` verwijst.

### 3.3 Arrays van arrays

Java kent geen meerdimensionale arrays. Een arrayelement `coordinaten[1,3]` bestaat dus niet.

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

```
double[][] coordinaten = new double [10][5];
coordinaten[0][0] = 0.735;
```

In bovenstaand voorbeeld declareren we een array *coordinaten* van het type `double`. De array heeft 10 elementen die alle 10 naar een array van 5 doubles wijzen.

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

```
//er wordt een array van 4 elementen gedefinieerd, waarvan
//alle elementen naar een array van ints verwijzen
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 is er een array *getallen* die bestaat uit 4 elementen die elk naar een array van ints verwijzen. Vervolgens maken we ruimte voor 4 arrays, allemaal met een verschillende dimensie. Merk op dat bij de declaratie het aantal elementen bij de



tweede dimensie niet wordt opgegeven. Uiteraard dien je wel de lengte van elke array goed in de gaten te houden, zodat je geen onbestaand element aanspreekt.



In het hoofdstuk Programmaverloop krijgen we nog meer voorbeelden te zien van het gebruik van arrays.

### 3.4 Oefeningen



Maak oefening 5 uit de oefenmap.

## Hoofdstuk 4 Programmaverloop

### 4.1 Blokken van statements

In Java worden statements gegroepeerd in *blocks*. Accolades geven het begin en het einde van een block aan. Ook de body van een class staat in een block. Een voorbeeld daarvan kan je vinden in het hoofdstuk over classes.

Blocks kunnen ook genest worden. Dus binnen een block kan je een ander block schrijven. Voorbeeld :

```
public static void main(String[] args) {  
    // TODO code application logic here  
  
    int a = 5;  
    {  
        //Dit is een block binnen het mainblock  
        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);  
}
```

FOUT !

Het nut van blocks binnen blocks moeten we zoeken bij de *scope* van variabelen. Met de scope bedoelen we de levensduur van een variabele. Variabelen die binnen een block zijn gedefinieerd 'leven' maar tot aan het einde van een block. In het bovenstaande voorbeeld krijgen we dan ook een foutmelding als we *b* willen gebruiken na het block waarin *b* is gedeclareerd.

Tot slot van deze paragraaf nog een kleine opmerking over witruimte en het vervolgen van codelijnen. Java negeert alle witruimte. Als je een statement dus wil verdelen over meerdere lijnen, dan kan dit zonder meer. Je hoeft dus geen vervolgteken (underscore) te gebruiken zoals in Visual Basic. In de onderstaande afbeelding zie je een voorbeeld. Uiteraard moet de code wel leesbaar blijven en is volgend voorbeeld dan ook een beetje overdreven.

```
public  
    static  
    void  
        main(String[] args) {  
        // TODO code application logic here  
  
        int a =  
            5;  
        {
```



## 4.2 Keuzes

Accolades en blocks worden ook gebruikt in keuzestructuren. In deze paragraaf behandelen we drie keuzestructuren: de *if*, de *conditionele operator ?* en de *switch*.

### 4.2.1 De if-instructie

Af en toe zal je in Java bepaalde code al dan niet willen laten uitvoeren afhankelijk van de waarde van één of andere variabele. Hiertoe gebruik je de *if-instructie*. Na de *if* plaats je een *expressie* die moet resulteren in een *boolean*. Afhankelijk van de waarde van deze boolean-expressie wordt een statement of een block al dan niet uitgevoerd. De test moet tussen *ronde haken* staan.

Een voorbeeld met één enkel statement :

```
int getal = 6;

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

Een voorbeeld met een block :

```
char weer = 'z';    // z van zonnig
if (weer == 'r') {  // r van regenchtig
    System.out.println("Laarsjes aan");
    System.out.println("Paraplu open");
}
```

Een block is verplicht wanneer er meer dan één statement bij de *if* of *else* hoort. Een volledige *if-else*-structuur is één statement. Advies: gebruik altijd een block om de onderhoudbaarheid van de code te bevorderen.

Er bestaat ook een variante op bovenstaande *if*-constructie, namelijk de *if-else*. Ook hier opnieuw een test en een statement of block dat uitgevoerd moet worden als aan de voorwaarde voldaan is. Maar er is ook een block of statement dat moet uitgevoerd worden als er niet aan de voorwaarde voldaan is.

```
char weer = 'z';
if (weer == 'r') {
    System.out.println("Laarsjes aan");
    System.out.println("Paraplu open");
}
else {
    System.out.println("Laarsjes uit");
    System.out.println("Paraplu dicht");
}
```

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

```
char weer = 'z';
boolean voldoendeCentenBij = true;
if (weer == 'r')
{
    System.out.println("Laarsjes aan");
    System.out.println("Paraplu open");
}
else
    if (voldoendeCentenBij)
        System.out.println("Terrasje doen");
    else
        System.out.println("Een goeie maat zoeken");
```

#### 4.2.2 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 :

*test ? waardeIndienVoldaan : waardeIndienNietVoldaan*

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

```
int a = 5;
int b = 7;
int grootste = a < b ? b : a;
```

Het bovenstaande voorbeeld initialiseert de variabele *grootste* op de grootste waarde van *a* en *b*.



### 4.2.3 De switch

Bij een if-instructie heb je maar 2 mogelijkheden: de test is *true* of *false*. Indien je een keuze moet maken tussen meerdere mogelijkheden, dan kan een geneste if-instructie een uitweg bieden, maar meestal wordt de constructie dan zeer onleesbaar.

De switch kan hier een uitweg bieden. We gaan een *expressie* evalueren en vergelijken met een aantal mogelijkheden. Is het resultaat van de expressie verschillend van elk van deze mogelijkheden, dan kan er een *defaultcode* worden uitgevoerd. De syntax :

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

Een voorbeeld:

```
int budget = 30;  
String menu;  
switch (budget) {  
    case 10:  
        menu = "dagschotel";  
        break;  
    case 20:  
        menu = "weekendmenu";  
        break;  
    case 30:  
        menu = "zakenlunch";  
        break;  
    case 40:  
        menu = "fijnproeversmenu";  
        break;  
    default: menu = "Verkeerde waarde";  
        break;  
}
```



In bovenstaand voorbeeld wordt de *int*-variabele *budget* vergeleken met een aantal mogelijkheden. Telkens wordt de *string*-variabele *menu* op een gepaste waarde gezet. Om dan telkens de switch te verlaten wordt de instructie *break* gebruikt. Indien deze er niet zou staan, dan zou de code die volgt gewoon verder worden uitgevoerd.

Een voorbeeld zonder breaks:

```
int budget = 30;
switch (budget) {
    case 50 : System.out.println("Champagne");
    case 40 : System.out.println("Voorgerecht");
    case 30 : System.out.println("Dessert en koffie");
    case 20 : System.out.println("Hoofdgerecht");
}
```

In dit voorbeeld wordt de waarde van *budget* geëvalueerd. Er wordt gesprongen naar de juiste case en vanaf daar worden alle statements die volgen eveneens uitgevoerd. Voor een budget van 30 euro heeft u dus een hoofdgerecht, dessert en koffie, voor 50 euro staat alles op het menu: champagne, voorgerecht, dessert, koffie en hoofdgerecht (in werkelijkheid hopelijk niet in die volgorde).

Laten we een ander voorbeeld nemen:

```
String hetweer = "guur";
switch (hetweer){
    case "guur":
        System.out.println("Muts en handschoenen"); break;
    case "koud":
        System.out.println("Jas"); break;
    case "fris":
        System.out.println("Trui"); break;
    case "aangenaam":
        System.out.println("T-shirt"); break;
    default:
        System.out.println("Belgisch weer"); break;
}
```

Kan enkel vanaf Java 7

Als je dit uitprobeert dan kan je een foutmelding krijgen, afhankelijk van de Java versie die je gebruikt.

Java aanvaardt als type voor de te evalueren switch-variabele:

- primitive data types *byte*, *short*, *char* en *int*
- enum
- String (sinds Java 7)



Een nadeel van de switch is dat de te evalueren variabele telkens exact gelijk moet zijn aan een mogelijke waarde. Je kan bijvoorbeeld evenmin het volgende schrijven:

```
int temperatuur = 12;
switch (temperatuur) {
    case (temperatuur<0) :
        System.out.println("Het vriest");
        break;
    case (temperatuur>=0 && temperatuur<10) :
        System.out.println("Koud weer");
        break;
    case (temperatuur>=10 && temperatuur<20) :
        System.out.println("Normaal");
        break;
    case (temperatuur>=20 && temperatuur<30) :
        System.out.println("Warm weer");
        break;
    case (temperatuur>30) :
        System.out.println("Hittegolf");
}
```

FOUT !

FOUT !

FOUT !

FOUT !

FOUT !

### 4.3 Lussen

In Java (en vrijwel alle andere programmeertalen) kan je één of meerdere statements een aantal keer laten herhalen. Hoeveel keer hangt meestal af van de waarde van één of meerdere variabelen. Dit herhalen noemen we een iteratie. Java kent 3 soorten iteraties:

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

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

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

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

- **for**

```
for (initialisatie ; voorwaarde ; iteratie-expressie) {  
    statements;  
}
```

### 4.3.1 De while-lus

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

Een voorbeeld :

```
int countdown = 10;  
while (countdown >= 0) {  
    System.out.println(countdown);  
    countdown--;  
}
```

In bovenstaand voorbeeld wordt er afgeteld van 10 naar 0. De voorwaarde tussen de ronde haken moet van het type boolean zijn.

### 4.3.2 De do...while-lus

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

Een voorbeeld :

```
int countdown = 10;  
do {  
    System.out.println(countdown);  
    countdown--;  
} while (countdown >= 0);
```

### 4.3.3 De for-lus

Bij een *for-lus* wordt een aantal statements herhaald zolang aan een bepaalde voorwaarde is voldaan.

Een voorbeeld :

```
for (int countdown = 10; countdown >=0; countdown--) {  
    System.out.println(countdown);  
}
```



De for-lus bestaat dus uit 3 delen: een initialisatie, een voorwaarde om in de lus te blijven en een iteratie-expressie. De initialisatie wordt één enkele keer uitgevoerd, namelijk vooraleer de for-lus een aantal malen wordt uitgevoerd. Je kan er dus perfect een variabele in declareren en initialiseren. Declareer je hier een variabele, dan is deze na de for-lus niet meer gekend.

De test, die een boolean moet opleveren, wordt uitgevoerd vóór iedere uitvoering van de lus. Iedere keer nadat de lus is doorlopen, wordt de iteratie-expressie uitgevoerd.

Bemerk dat deze code een stuk bondiger is dan de twee vorige lusstructuren. Bovendien zou je hier zelfs de accolades kunnen weglaten, aangezien het te herhalen block slechts uit één enkel statement bestaat.

Het kan zelfs nog korter: zowel het initialisatie-gedeelte als de iteratie-expressie van de for kan meerdere statements bevatten, van elkaar gescheiden door komma's. Je kan de code dus ook als volgt schrijven :

```
for (int countdown = 10; countdown >=0;
    System.out.println(countdown), countdown--);
```

Wie het helemaal kort wil houden, kan ook gewoon het volgende schrijven :

```
for (int countdown = 10; countdown >=0;
    System.out.println(countdown--));
```

Deze verkorte schrijfwijze wordt niet aanbevolen. Zij hypothekeert de onderhoudbaarheid en leesbaarheid van het programma.

#### 4.3.4 Breaks en labels

Bij alle soorten lussen die we net bekeken, werd de lus beëindigd wanneer niet langer aan een bepaalde voorwaarde werd voldaan. Nu kan het zijn dat de lus om één of andere onvoorziene omstandigheid toch eerder moet verlaten worden.

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

In het volgende voorbeeld bepalen we de faculteit van de getallen uit een array. We willen dat het resultaat een int is want we hebben een int nodig voor een latere toepassing die enkel met int's kan werken. Voor sommige getallen zal het resultaat niet in een int passen en krijgen we een overflow. Java zal geen foutmelding geven en dus moeten we zelf de berekening stoppen van zodra de maximale waarde van een int overschreden wordt. Er wordt ook een gepaste melding gegeven. Het programma gaat dan verder met het volgende getal.

```
//declaratie en initialisatie van getallenreeks
byte[] getallen = {7, 12, 28, 2, 9};

//declaratie hulpvariabelen
int i=0;
int faculteit;
long tussenresultaat;

//lus die de array doorloopt
while (i< getallen.length) {

    //initialisatie hulpvariabelen
    tussenresultaat = getallen[i];
    int j=1;

    //berekening faculteit
    while (j<getallen[i]) {
        tussenresultaat *= j;
        if (tussenresultaat > Integer.MAX_VALUE) break;
        j++; }

    //uitvoer resultaat
    if (j==getallen[i]) {
        faculteit = (int)tussenresultaat;
        System.out.println("De faculteit van " + getallen[i] +
                           " is " + faculteit);
    } else
        System.out.println("De faculteit van " + getallen[i] +
                           " is te groot voor een int");

    //teller verhogen
    i++;
}
```

In de berekeningslus testen we of het tussenresultaat al groter is dan de maximum integerwaarde. Gelukkig hoef je niet zelf te weten wat de grootste waarde is van een int maar bestaat er een class Integer met een property MAX\_VALUE die op 2.147.483.647 staat (meer over classes en de class Integer in het bijzonder in het volgende hoofdstuk).

Is het tussenresultaat effectief groter, dan wordt de lus beëindigd door het *break-statement*. Als de berekening volledig uitgevoerd is zonder overflow, dan krijg je het resultaat te zien, anders een melding dat het resultaat te groot is voor een int.



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 iteratie-expressie uitgevoerd en vervolgens wordt er opnieuw getest om eventueel in de lus te blijven.

Een voorbeeld :

```
//declaratie en initialisatie van getallenreeks
byte[] getallen = {7, 12, 28, 2, 9};

//declaratie hulpvariabelen
int i=0;
int faculteit;
long tussenresultaat;

//lus die de array doorloopt
while (i++ < getallen.length) {

    //initialisatie hulpvariabelen
    tussenresultaat = getallen[i-1];
    int j=1;

    //berekening faculteit
    while (j<getallen[i-1]) {
        tussenresultaat *= j;
        if (tussenresultaat > Integer.MAX_VALUE) break;
        j++;
    }

    //lus verlaten indien overflow
    if (j<getallen[i-1]) continue;

    //resultaat tonen
    faculteit = (int)tussenresultaat;
    System.out.println("De faculteit van " + getallen[i-1] + " is " +
        faculteit);
}
```

Deze keer tonen we enkel een resultaat als de faculteit helemaal uitgerekend is. Is er overflow, dan tonen we niets, maar gaan we met het *continue* statement naar het begin van de buitenste lus waarin het *continue-statement* staat.

In de bovenstaande voorbeelden hadden de `break` en `continue` statements telkens betrekking op de binnenste lus. Aan de hand van een label en een *aangepast continue-statement* kan je de code laten verderzetten aan het begin van de buitenste lus. Met een *aangepast break-statement* kan je uit een buitenste lus springen.

We gaan dit even demonstreren door ons bovenstaand voorbeeld enigszins aan te passen. Als de faculteit van een getal te groot wordt voor een `int`, stoppen we volledig met het berekenen van de faculteit van alle getallen. Dat gaat als volgt :

```
//declaratie en initialisatie van getallenreeks
byte[] getallen = {7, 12, 28, 2, 9};

//declaratie hulpvariabelen
int i=0;
int faculteit;
long tussenresultaat;

//lus die de array doorloopt
uit:
while (i++ < getallen.length) {

    //initialisatie hulpvariabelen
    tussenresultaat = getallen[i-1];
    int j=1;

    //berekening faculteit
    while (j<getallen[i-1]) {
        tussenresultaat *= j;
        if (tussenresultaat > Integer.MAX_VALUE) break uit;
        j++;
    }

    //resultaat uitvoeren
    faculteit = (int)tussenresultaat;
    System.out.println("De faculteit van " + getallen[i-1] +
        " is " + faculteit);
}
```

In dit voorbeeld hebben we de buitenste loop een label gegeven. Dit doe je door vóór de loop een labelnaam te plaatsen gevolgd door een dubbelpunt. In plaats van een gewoon `break` statement, zetten we nu `break` en de naam van het label.



Op een gelijkaardige manier kunnen we de *continue met een label* gebruiken. Opnieuw een voorbeeld :

```
//declaratie en initialisatie van getallenreeks
byte[] getallen = {7, 12, 28, 2, 9};

//declaratie hulpvariabelen
int i=0;
int faculteit;
long tussenresultaat;

//lus die de array doorloopt
uit:
while (i++ < getallen.length) {

    //initialisatie hulpvariabelen
    tussenresultaat = getallen[i-1];
    int j=1;

    //berekening faculteit
    while (j<getallen[i-1]) {
        tussenresultaat *= j;
        if (tussenresultaat>Integer.MAX_VALUE) continue uit;
        j++;
    }

    //resultaat uitvoeren
    faculteit = (int)tussenresultaat;
    System.out.println("De faculteit van " + getallen[i-1] +
        " is " + faculteit);
}
```

Het resultaat is nu dat de binnenste lus wordt stopgezet van zodra een te hoog getal wordt bereikt, en dat er gesprongen wordt naar het begin van de buitenste lus. Het proces gaat verder met het volgende getal. Bemerk dat deze code korter is dan het voorbeeld op pagina 44.



Terwijl het net de bedoeling is code te vereenvoudigen, kan je met breaks, continues en labels code hopeloos complex maken. We spreken dan eerder van een spaghettistructuur dan van gestructureerd programmeren. Wees dus spaarzaam met breaks, continues en labels.





## 4.4 Oefeningen



Maak oefeningen 6 t.e.m. 10 uit de oefenmap.



## Hoofdstuk 5 OO, Classes en Objects

---

### 5.1 Java en Object Oriëntatie

Aangezien men bij de ontwikkeling van Java van nul is begonnen en er geen voor-geschiedenis moest in rekening gebracht worden, heeft men de programmeertaal behoorlijk zuiver laten aansluiten bij de OO-theorie. Voor de theorie zelf: zie de cursus 'Object Oriëntatie'.

De paar afwijkingen op de regels zijn er meestal gekomen voor het comfort van de programmeur, bijvoorbeeld de primitive datatypes maken het ons gemakkelijk maar puur theoretisch zouden dit ook classes moeten zijn.

In Java spreekt men dus niet meer van programma's. Men bouwt applicaties die bestaan uit één of meer objecten die samenwerken om het beoogde resultaat te leveren.

Vanwaar komen die objecten? Objecten worden aangemaakt in de code zelf, op basis van een 'sjabloon', een 'blauwdruk', ... wat algemeen een **class** genoemd wordt.

Een class is een 'abstracte' beschrijving van een 'ding' (een concreet ding als een TV, een auto, ... of een abstract ding als een functie zoals 'manager', 'boekhouder', ...)

Een class beschrijft eigenschappen en gedrag van het ding. De eigenschappen worden **data members** of **properties** genoemd, het gedrag wordt vastgelegd in **methods**.

De data members van een class bevatten geen concrete waarde. Bijvoorbeeld bij een class TV zal er een property *grootteBeeld* zijn, maar het is enkel indien er van die class een concreet TV-object gecreëerd (**geïntanceerd**) wordt, dat men van dit specifieke object kan zeggen dat de *grootteBeeld* 72cm is (of 90 cm, of 108 cm, ...). Voor ieder ander TV-object dat gecreëerd wordt op basis van dezelfde class TV, kan de property *grootteBeeld* een andere waarde hebben.

Conclusies:

- Van iedere class kunnen meerdere objecten geïntanceerd worden (met uitzondering van een Singleton, een class die speciaal geconstrueerd is om maar één object toe te laten).
- Ieder object heeft zijn eigen set data members! Iedere persoon heeft een eigen schoenmaat, iedere auto zijn eigen nummerplaat.
- Methods die worden aangeroepen vanuit een specifiek object, werken met de property-set van dit specifiek object!

Opmerkingen:

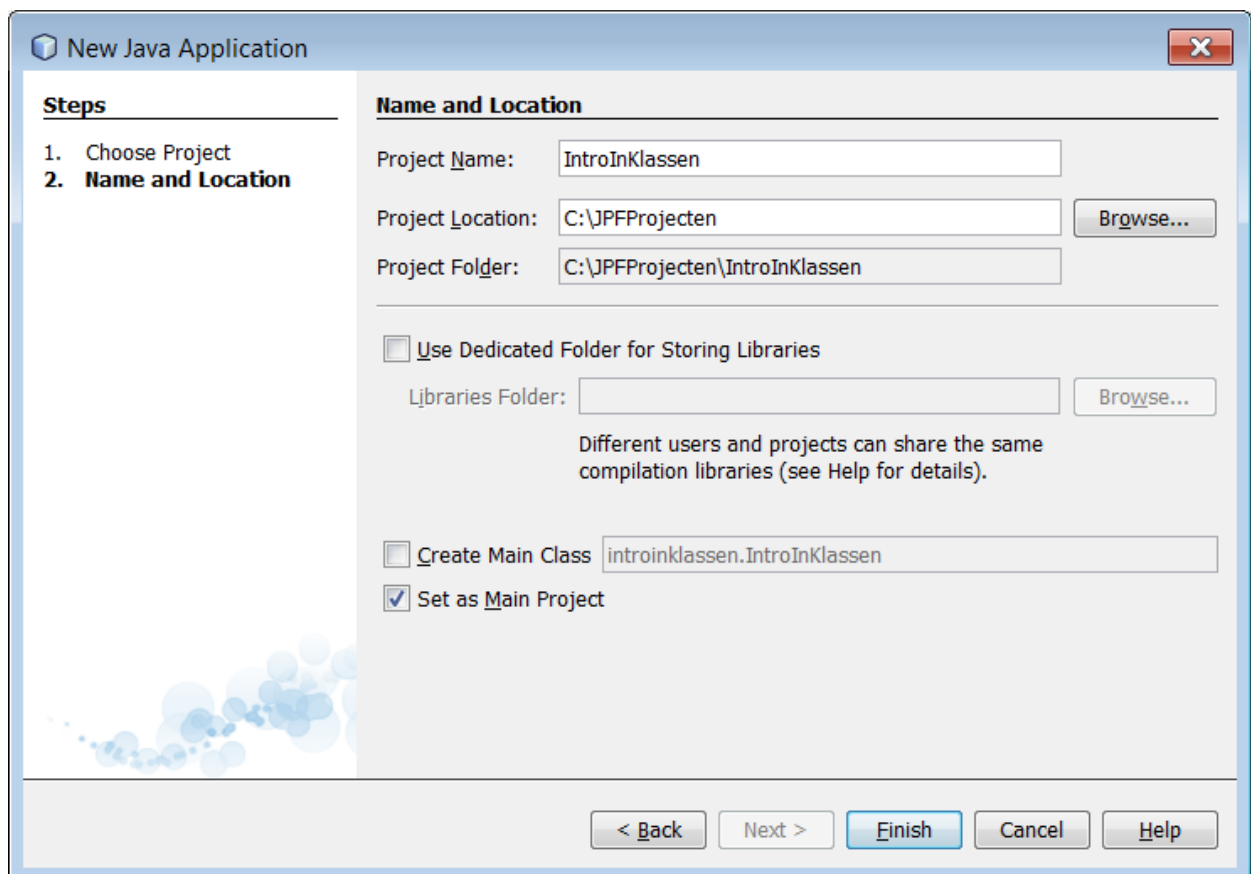
- Het is mogelijk om data members te definiëren in de class, die niet specifiek zijn per object, maar die daarentegen gedeeld worden door alle instances van die class. Men spreekt van data members (of properties) met class-bereik (zie cursus OO en paragraaf 5.7 "Instance members versus class members").
- Eveneens is het mogelijk om methods te beschrijven die niet werken met de property-set van een specifiek object, maar die enkel werken met de data members met class-bereik. Deze methods kunnen aangeroepen worden via de naam van de class.

Zelfs het meest eenvoudige voorbeeld (zie bmi-berekening in een vorig hoofdstuk) bestaat al uit meer dan één object: het zelfgemaakte object “Main” en de (standaard Java-)class System, die wordt gebruikt om output naar buiten te sturen.

## 5.2 Onze eerste class

We maken in NetBeans een eerste class: **SpaarRekening**.

- *We sluiten het vorige project File – Close Project (JPF) en we maken onder C:\JPFPProjecten een nieuw project (Java – Java Application) met de naam ‘IntroInKlassen’ aan. Haal het vinkje vóór Create Main Class weg. We willen immers nog geen Main Class maken. Klik tenslotte op Finish.*



- *Als je wil kan je de sourcecode van regelnummers voorzien. Dat doe je via het menu View – Show Line Numbers.*
- *We maken nu in ons project via File – New File: Categorie Java en File type Java Class een nieuwe class aan. Je kan ook rechtsklikken op de naam van het project in het Projects-venster en dan kiezen voor New – Java Class. Wanneer deze keuze nog niet beschikbaar is, dien je eerst via de laatste optie Other... te kiezen voor Categorie Java en File type Java Class. Dit hoeft je slechts één keer te doen, vanaf dan is de optie beschikbaar.*



- *We geven de class een naam: `SpaarRekening` (een classnaam begint altijd met een hoofdletter, verder zal ieder nieuw woord met een hoofdletter beginnen!).*

New Java Class

**Steps**

1. Choose File Type
2. **Name and Location**

**Name and Location**

Class Name:

Project:

Location:

Package:

Created File:

Warning: It is highly recommended that you do NOT place Java classes in the default package.

< Back Next > Finish Cancel Help

- *Je kan nu op **Finish** klikken.*

Resultaat:

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5
6  /**
7   *
8   * @author slucas
9   */
10 public class SpaarRekening {
11
12 }
13
```

Wat zit er nu in de class `SpaarRekening`?

**Regel 1 – 4:** normaal commentaar.

**Regel 6 – 9:** documentatiecommentaar: zie pagina 21.

**Regel 10:** de header van de classdefinitie. In dit geval, een publieke class (een class die dus door de buitenwereld kan aangeroepen worden) met de naam `SpaarRekening`. Let wel, de naam van de publieke class **moet** ook de naam zijn van de sourcefile (uitgebreid met het suffix **.Java**): `SpaarRekening.Java`

De body van de class komt tussen de accolades op regel 10 en 12. Volgens de java-naming-convention begint de naam van een class met een hoofdletter.

In verdere voorbeelden gebeurt het soms dat de source van meerdere classes gegroepeerd staat in één source file. Dit is om didactische redenen. In de praktijk (en dit is ook wat NetBeans doet) heeft men één sourcefile per class. Groepeert men meerdere classdefinities in één sourcefile dan moet (mag) er slechts één class **public** zijn, alle andere krijgen geen accessmodifier en hebben dus (by default) package-access (zie later). Deze éne public class geeft dan zijn naam aan de sourcefile.

We gaan de SpaarRekening een aantal data members geven en een aantal methods. Voeg de code zoals hieronder weergegeven toe:

```
/*
 * SpaarRekening.java
 */
import java.util.*;                                     (1)
/**
 *
 * @author slucas
 */
public class SpaarRekening {
    private String rekeningNummer;                       (2)
    private double saldo;
    private double intrest;

    /** Creates a new instance of SpaarRekening */
    public SpaarRekening(String reknr, double intrest) { (3)
        rekeningNummer = reknr;
        this.intrest = intrest;
        saldo = 0;
    }

    public void storten (double bedrag) {                (4)
        saldo += bedrag;
    }
    public void afhalen (double bedrag) {
        saldo -= bedrag;
    }
    public void overschrijven(SpaarRekening spaarRek, double bedrag,
                               Date datum) {
        saldo -= bedrag;
        spaarRek.storten(bedrag);
    }
    public double geefSaldo(){
        return saldo;
    }
}
```



Verklaring:

(1) `import java.util.*`

We gaan in onze class de mogelijkheid voorzien om een overschrijving uit te voeren op een bepaalde datum. Deze method (zie later) krijgt dus een datum als parameter. Een datum-object is afgeleid van de class *Date*. Deze class zit in de package *java.util*. Om te vermijden dat we bij een verwijzing naar de *Date*-class zouden moeten schrijven ***java.util.Date***, vermelden we liever het import statement. De vermelding van de *\** betekent dat alle classes uit de *java.util*-package beschikbaar zijn. Dit impliceert niet dat de sourcecode groter wordt. Enkel die classes uit de package die echt nodig zijn worden toegevoegd aan de source.

Het is ook mogelijk om: ***import java.util.Date;*** te schrijven. Dit kan soms helpen om naamconflicten te vermijden: bij gebruik van meerdere imports met de *\** kunnen in de verschillende packages classes zitten met dezelfde naam. Indien men hiernaar refereert kan de JRE niet weten uit welk package moet gekozen worden.

Zelfs indien men gebruik maakt van imports, kan men in de sourcecode zelf nog altijd de volledige naam gebruiken :

***java.util.Date mijnDatum = new java.util.Date();***

(2) Declaratie van 3 ***private*** variabelen

De plaats waar deze variabelen gedeclareerd worden (bovenaan, buiten een method) maakt dat ze globaal bekend zullen zijn in de ganse class. In dit voorbeeld zijn het geen variabelen met een class-bereik (hiervoor ontbreekt het sleutelwoord ***static***). Men noemt ze dan ook ***data members***, ***properties*** of ***object variabelen***.

Data members worden automatisch geïnitialiseerd. 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***.

Dat ze ***private*** zijn, is een logische keuze. Men gaat zoveel mogelijk de data members inkapselen (***encapsulation***) zodat de buitenwereld geen rechtstreekse controle heeft over de inhoud. De benadering van de inhoud gaat altijd via ***methods*** wat aan de class-bouwer de mogelijkheid geeft om controle uit te oefenen over de waarden die men zou willen wijzigen. Men kan in dit geval bijvoorbeeld weigeren om negatieve bedragen te aanvaarden voor een storting. Dit geeft ook de mogelijkheid om een data member read-only te maken: men voorziet wel een method om de gegevens op te vragen, maar niet om de waarde te wijzigen.

(3) Dit is een ***constructor***. Een constructor method is in die zin speciaal dat ze een identieke naam heeft als de class (hoofdlettergevoelig!) en dat ze niets zegt over een teruggeefwaarde (ze zegt zelfs niet dat er niets – ***void*** – teruggegeven wordt).

De constructor method wordt uitgevoerd op het ogenblik dat er een object geïntantieerd wordt van een class. In praktijk kan die gebruikt worden om een aantal gegevens te initialiseren.

Opmerking: als men een class maakt **zonder constructor** method, dan zal de compiler automatisch een default constructor aanmaken. Eens men zelf een expliciete constructor heeft gemaakt (met of zonder parameters) dan wordt er geen default constructor meer aangemaakt.

Een constructor is meestal **public**, dit impliceert dat iedereen die toegang heeft tot de class er ook een instance kan van maken. Wil men dit vermijden dan maakt men een constructor **private**. De math-class heeft bijvoorbeeld een private constructor. Het is niet de bedoeling dat iemand een eigen math-object maakt. De verschillende methods uit de math-class zijn aan te spreken zonder dat men er een instance moet van maken.

In de SpaarRekening heeft de constructor method twee parameters: het rekeningnummer en een intrestvoet. Binnen de method-body worden de drie data members geïntialiseerd.

Opmerking: de tweede parameter heeft dezelfde naam als een data member, daarom gebruik je het sleutelwoord **this**, een punt en de naam van de property om naar de property te verwijzen en niet naar de parameter. **this** verwijst altijd naar het actuele object.

(4) Definitie en werking van de 4 public methods.

Er wordt, voor de eenvoud, geen controle uitgevoerd op de ingegeven bedragen. De methods **storten()**, **afhalen()** en **overschrijven()** geven geen enkele waarde terug, daarom het sleutelwoord **void**. De method **geefSaldo()** geeft wel een waarde terug en wel van het type **double**.

De method **overschrijven()** verwacht drie parameters: één van het type **SpaarRekening**, één van het type **double** en één van het type **Date**. In de verwerking doen we nu nog niets met de datum.



Belangrijke opmerking:

Om de leesbaarheid van je code te bevorderen geef je een method die een property wijzigt een naam die begint met **set** en dan de naam van de property met beginhoofdletter. De property zelf heeft een kleine beginletter.

Dus als je een property **naam** hebt dan maak je een method **setNaam()**.

Een method die gebruikt wordt om de waarde van een data member op te vragen laat je beginnen met **get** en dan de naam van de property met beginhoofdletter. Dus bijvoorbeeld **getNaam()**.

Dit is wat gebruikelijk is binnen alle standaardclasses van Java.

De method **geefSaldo()** wordt dus eigenlijk beter **getSaldo()** genoemd.

Een get-method wordt ook wel een “getter” genoemd, een set-method een “setter”.



En nu?

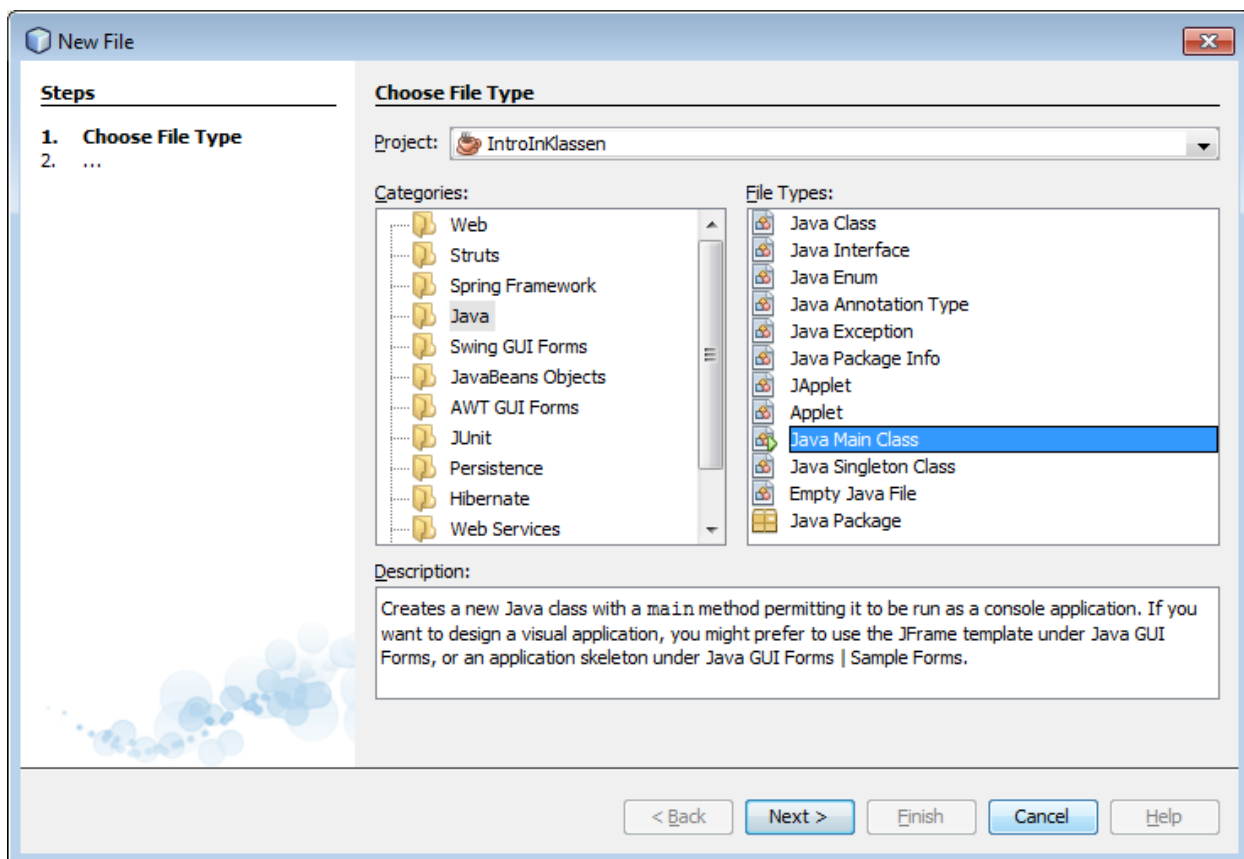
We hebben nu een class van het type `SpaarRekening` gemaakt (en gecompileerd), maar hoe gaan we die nu gebruiken?

We hebben nood aan een nieuwe class (`BankBediende`), die voor ons `SpaarRekening`-objecten kan maken, die voor ons geld kan storten, kan afhalen, kan overschrijven, ...

- We maken dus een nieuwe class `BankBediende` binnen hetzelfde project via *File - New File*: Categorie *Java* en File Type **Java Main Class**  
Je kan ook rechtsklikken op de package en vervolgens kiezen voor *New, Java Class* (eventueel de eerste keer via *Other...*)

Een Java Main Class is een class die als startpunt van een applicatie kan dienen. De JRE zal een applicatie altijd opstarten via een **main()-method**. Classes die niet gebruikt worden als beginpunt van een applicatie, moeten geen `main()`-method hebben.

- Klik Next en geef de class de naam **BankBediende** en klik op Finish.



NetBeans genereert nu voor ons een publieke class `BankBediende` met een **main()**-method. De `main()`-method is **public** (de buitenwereld - JRE in dit geval - moet er aan kunnen), ze is **static**, wat betekent dat het een method is met class-bereik (dus te gebruiken zonder dat er noodzakelijk een instance bestaat, dit komt later nog aan bod),



ze levert niets terug (**void**) en verwacht een **array van Strings** (dit zijn eventuele parameters die meegegeven worden bij het opstarten van de applicatie).

- We vullen aan met code:

```
public static void main(String[] args) {  
    double stand;  
    SpaarRekening spaar1 = new SpaarRekening("123", 5.0);  
    SpaarRekening spaar2 = new SpaarRekening("888", 4.0);           (1)  
    spaar1.storten(100.0);  
    stand = spaar1.geefSaldo();  
    System.out.println("Saldo van spaar1: " + stand);  
    spaar1.overschrijven(spaar2, 80.0, new Date());                 (2)  
    stand = spaar1.geefSaldo();  
    System.out.println("Saldo van spaar1 na overschrijving: " + stand);  
    stand = spaar2.geefSaldo();  
    System.out.println("Saldo van spaar2 na overschrijving: " + stand);  
}
```

- Je kan de code nu uitproberen. Mogelijk komt NetBeans vragen naar het main project en de main class. Kies dan voor IntroInKlassen en vervolgens voor BankBediende.

Het resultaat van het programma:

```
Saldo van spaar1: 100.0  
Saldo van spaar1 na overschrijving: 20.0  
Saldo van spaar2 na overschrijving: 80.0
```

Verklaring bij de code:

- (1) Er worden twee spaarrekeningen aangemaakt spaar1 en spaar2.  
Bij de aanmaak worden er waarden meegegeven die de constructor verwacht.
- (2) De method **overschrijven()** verwacht o.a. een datum. De constructie **new Date()** levert een anoniem object op van de class datum met als waarde de huidige datum.

Opmerking:

In dit voorbeeld is er gekozen om de twee class-definities (SpaarRekening en BankBediende) in aparte source files te steken binnen hetzelfde project (en binnen dezelfde package).

Door het feit dat we de twee classes binnen hetzelfde project hebben gestoken, moeten we in de class BankBediende ook geen import-instructie opnemen om de class SpaarRekening te vinden. JRE zoekt – by default – in de actuele package.



## 5.3 Oefening



Maak oefening 11 uit de oefenmap.

## 5.4 Het doorgeven van parameters bij een method-aanroep

Zoals uit bovenstaande voorbeelden blijkt, kan men alles doorgeven als parameter (getallen, strings, volledige objecten, ...) zolang in de parameterlijst van de method-definitie maar de gepaste types voorzien zijn.

Toch is er een verschil: zijn de parameters van een **primitive datatype**, dan wordt de **waarde** doorgegeven en wordt er binnen de method gewerkt op een kopie van het origineel. Men noemt dit **passing by value**. De value of waarde wordt doorgegeven, niet het origineel.

Is de parameter een **referencetype** (een reference naar een array, een string, een rekening, ...), dan wordt de **referencevariabele** meegegeven. Binnen de method wordt een kopie gemaakt van de inhoud van die referencevariabele. Een **referencevariabele** bevat het 'geheugenadres' van het object waaraan hij gekoppeld is. De kopie (met zelfde inhoud) refereert dus naar hetzelfde originele object. Wijzigt men binnen de method iets aan de status van het object, dan is het origineel gewijzigd. Men noemt dit **passing by reference**.

Opgelet: gaat men binnen de method de kopie-referencevariabele naar iets anders laten wijzen (door bijvoorbeeld een assignment), dan werkt men niet meer op het originele object!



Opmerking:

Geeft men als parameter een array mee (bijv. een integer-array van 20 elementen) dan moet men in de method-header, in de parameterlijst, alleen vermelden dat men een integer-array binnenkrijgt: **`int[] mijnTabel`**.

De lengte van de array zal moeten bepaald worden via de **length** property van de bewuste tabel: **`mijnTabel.length`** zal de waarde 20 opleveren. Hetzelfde is waar indien men een array als return-value gebruikt.

## 5.5 Method overloading

We hebben nu een class *SpaarRekening* waarin wij een method *overschrijven()* hebben voorzien. Deze method verwacht een (memo)datum (waar we in de actuele oplossing niets mee doen). Hoe lossen we nu het probleem op van de overschrijvingen zonder memodatum?

Optionele parameters – met eventueel een default value - bestaan er niet in Java.

Een oplossing zou kunnen zijn om twee methods te voorzien met een verschillende naam bijv. *overschrijvenMetDatum()* en *overschrijvenZonderDatum()*. Dit verschuift het probleem naar de aanroeper van de method.

Een elegantere oplossing is **method overloading**.

We schrijven twee methods met dezelfde naam maar met een verschillend aantal parameters.

```
public void overschrijven(SpaarRekening spaarRek, double bedrag,
                          Date datum) {
    saldo -= bedrag;
    spaarRek.storten(bedrag);
}

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

- Voeg de tweede versie van de method 'overschrijven' toe in jouw code.

Voor de gebruiker van de class wordt het eenvoudig, bij het aanroepen van de method **overschrijven()** geeft hij twee of drie parameters mee, volgens behoefte. Het is de **JRE** die zal bepalen welke versie van de method nu moet uitgevoerd worden.

We kunnen zoveel varianten schrijven van een method als nodig. Ze moeten allen dezelfde naam hebben en ze moeten verschillen van elkaar in het aantal parameters en/of het datatype van de parameters. We zouden dus nog een method **overschrijven()** kunnen voorzien, maar waar de parameter *bedrag* van het type **integer** is , ...

We kunnen overloading ook toepassen op de constructor. Het is dus best mogelijk om meerdere constructors te voorzien met een verschillende parameterset.

In ons voorbeeld wordt er altijd een spaarrekening aangemaakt met een beginsaldo van € 0. Men kan eenvoudig een tweede constructor voorzien die naast de twee reeds bestaande parameters, ook een beginsaldo aanneemt :

```
public SpaarRekening(String reknr, double intrest) {
    rekeningNummer = reknr;
    SpaarRekening.intrest = intrest;
    saldo = 0;
}

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

- Voeg de tweede constructor toe aan jouw code.



Een alternatieve oplossing:

```
public SpaarRekening(String reknr, double intrest) {
    rekeningNummer = reknr;
    this.intrest = intrest;
    saldo = 0;
}

public SpaarRekening(String reknr, double intrest, double beginSaldo) {
    this(reknr, intrest);
    saldo = beginSaldo;
}
```

In de tweede constructor gebruiken we de eerste constructor door het sleutelwoord **this** (**this** verwijst naar het actuele object) te vermelden met tussen haakjes 2 parameters. Daarna wordt ook nog het saldo ingevuld.

Het oproepen van een andere constructor moet de eerste instructie zijn binnen deze constructor. Deze alternatieve oplossing is handig omdat je code die reeds in een andere constructor staat niet opnieuw moet overnemen (bijv. een aantal controles).

## 5.6 Private methods

Gebruikt men binnen een class – volgens de goede aanpak van gestructureerd programmeren – hulpmethods, dan zal men die steeds private declareren. Voordeel: de ‘buitenwereld’ heeft er geen weet van, de bouwer van de class kan die hulpmethods naar eigen inzicht wijzigen, optimaliseren, ... zonder dat de gebruiker hiervan last ondervindt.

In ons voorbeeld van de SpaarRekening maken we een private method die de controle doet op de geldigheid van het rekeningnummer (de 97-controle).

- *Voeg volgende code toe in jouw programma :*

```
private boolean checkNummer(String reknr) {
    //we gaan ervan uit dat het rekeningnummer wordt
    //aangeboden in de vorm xxx-xxxxxxx-xx
    int d1=Integer.parseInt(reknr.substring(0,3));
    int d2=Integer.parseInt(reknr.substring(4,11));
    int d3=Integer.parseInt(reknr.substring(12,14));

    long deeltal=d1 * 100000000L + d2;
    int rest=(int)(deeltal % 97);
    if (rest==0) rest=97;
    return (rest==d3);
}
```

Verklaring bij de code:

- (1) De parameter **reknr** is een String. Enig speurwerk in de documentatie van de String-class leert dat er een method **substring()** bestaat, die uit een bepaalde string een deel kan kopiëren. Deze method wordt driemaal toegepast om het rekeningnummer op te splitsen in zijn drie numerieke delen. Meer over deze method in paragraaf 7.3.3.

Via de method **Integer.parseInt()** kan men de respectievelijke numerieke delen omvormen tot integers. Opgelet: deze conversie kan fout gaan. Indien er iets anders dan cijfers voorkomt in één van de drie delen, dan ontstaat er een fout. Hoe ga je daar mee om? Dat zie je in een later hoofdstuk: Exception handling.

- (2) Van de eerste twee numerieke delen wordt één getal gemaakt. Het bereik van een integer is onvoldoende om (sommige) rekeningnummers te bevatten. Daarom wordt alles omgezet naar een **long**.
- (3) Een modulus-deling levert de rest van de deling door 97.
- (4) Een rekening eindigt nooit op 00. Indien het rekeningnummer een veelvoud van 97 is, wordt het controlegetal op 97 gezet.
- (5) De method retourneert het resultaat van de vergelijking: **true** of **false**.

Opmerkingen:

- Aangezien deze method private is, kan ze alleen aangeroepen worden binnen de class SpaarRekening zelf.
- Men kan ook hier aan method-overloading doen. Varianten zouden kunnen zijn: een method **checkNummer** die drie integer parameters heeft (deel1, deel2 en deel3) of een method die de rekening binnenkrijgt als één groot getal.
- De variabelen **d1**, **d2**, **d3**, **deeltal** en **rest** zijn allen **lokale variabelen** die binnen een method gecreëerd worden. Hun **life-time** is zeer kort: zij ontstaan bij het aanroepen van de method en verdwijnen bij het verlaten van de method. Hun **scope** (of zichtbaarheid) is ook zeer beperkt: lokale variabelen zijn enkel te zien binnen het block waarbinnen ze gedeclareerd worden (in dit geval: enkel binnen de method).
- De parameter **reknr** is eveneens een lokale variabele, maar aangezien het type String is - wat op zichzelf een class is - is deze lokale variabele een **referencevariabele** die naar de originele string verwijst !

## 5.7 Instance members versus class members

De oplossing voor onze SpaarRekening bevat alleen variabelen en methods met **instance-bereik**. M.a.w. iedere instance van de class krijgt zijn eigen set variabelen en iedere method van iedere instance wordt uitgevoerd met zijn eigen set variabelen.

Met deze manier van werken is het perfect mogelijk dat verschillende objecten (spaarrekeningen) een verschillend intrestpercentage hebben. In de praktijk zal het meestal niet zo zijn, alle spaarrekeningen hebben hetzelfde intrestpercentage.



Binnen Java wordt dit gerealiseerd door een variabele te voorzien van **class-bereik**. Dit gebeurt door toevoeging van het woord **static** bij de declaratie:

```
private static double intrest;
```

- *Voeg in jouw code het woord static toe aan de private variabele intrest.*

Door deze wijziging wordt in de 2 constructors de volgende coderegel best aangepast als volgt:

```
this.intrest = intrest  
aanpassen naar:  
SpaarRekening.intrest = intrest
```

De variabele *intrest* is een static variabele, d.w.z. een variabele met class-bereik en het is daarom duidelijker om te verwijzen naar de class ipv de this-referentie te gebruiken.

Dezelfde redenering kan gevolgd worden voor methods: indien zij niet afhankelijk zijn van de 'toestand' van een object, dan kan men ze beter definiëren met class-bereik. In ons voorbeeld zou de method **checkNummer()** een method kunnen zijn met class-bereik (door toevoeging van het keyword **static**).

Een ander voorbeeld van een static method :

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

- *Voeg bovenstaande method toe in de class **SpaarRekening**.*
- *Voeg in de class **BankBediende** onderaan volgende code toe :*

```
System.out.println("Rente van spaar1: " + SpaarRekening.getIntrest());  
System.out.println("Rente van spaar2: " + SpaarRekening.getIntrest());  
System.out.println("Rente voor alle spaarrekeningen: " +  
    SpaarRekening.getIntrest());
```

Het resultaat van de verwerking:

```
Saldo van spaar1: 100.0  
Saldo van spaar1 na overschrijving: 20.0  
Saldo van spaar2 na overschrijving: 80.0  
Rente van spaar1: 4.0  
Rente van spaar2: 4.0  
Rente voor alle spaarrekeningen: 4.0
```

Opmerkingen:

- Er is maar één variabele *intrest* (met class-bereik) en deze is gemeenschappelijk voor alle objecten die van de class worden afgeleid. Na creatie van *spaar1* is de *intrest* voor alle spaarrekeningen gelijk aan 5.0. Wanneer *spaar2* wordt aangemaakt wordt de *intrest* voor alle spaarrekeningen 4.0. Dus ook voor *spaar1*.
- Methods met class-bereik kunnen ook aangeroepen worden via de naam van het object, bijv. *spaar1.getIntrest()*; doch NetBeans raadt aan om de class-referentie te gebruiken.  
Ditzelfde geldt ook voor variabelen met classbereik. Ook deze kunnen aangeroepen worden via de naam van een object of via de naam van de class zelf (indien ze tenminste public zijn, wat hier niet het geval is).

## 5.8 Finalize method

We weten nu hoe en wanneer een object ‘geboren’ wordt, maar wanneer sterft een object?

Een object sterft wanneer er geen enkele reference meer bestaat naar dit object. Dit kan gebeuren door de reference-variabele de waarde ***null*** te geven, door de reference-variabele via een assignment naar een ander object te laten verwijzen of gewoon, door het programma te stoppen.

In de *jvm* is er een proces, de **garbage collector**, dat permanent (maar met een lage prioriteit) speurt naar afgestorven objecten. Vindt het die, dan worden ze opgeruimd en worden de resources (geheugen) vrijgegeven.

Het proces van de garbage collector kunnen wij niet beïnvloeden, ook al bestaat er een statische method ***System.gc()*** die de garbage collector expliciet aanroept. De garbage collector doet zijn werk als het systeem weinig bezet is en/of als er dringend geheugen moet vrijgemaakt worden voor nieuwe objecten.

Het enige wat wij als programmeur kunnen doen, is binnen de class-definitie een method ***finalize()*** voorzien. Deze method wordt automatisch aangeroepen en uitgevoerd net voor het object wordt opgeruimd. Deze method kan gebruikt worden om zelf bepaalde resources netjes vrij te geven (database koppelingen, netwerkverbindingen, ...).

## 5.9 Scope regels

Het begrip ***scope*** slaat zowel op de toegankelijkheid als op de zichtbaarheid van (in ons geval) data members en methods.

De programmeur kan elk data member en/of method ofwel ***public*** ofwel ***private*** beschrijven.

Instructies binnen de class zelf kunnen alles gebruiken, zowel de public als de private data members en/of methods. Via objecten, geïntantieerd van die class, kan je enkel de publieke onderdelen bereiken.

‘Good practice’ leert ons om zo weinig mogelijk public methods te gebruiken. Enkel wat echt noodzakelijk is om de functionaliteit aan de buitenwereld ter beschikking te stellen maak je public. Alle andere ‘hulpmethods’ die wij om één of andere reden definiëren,



worden **private** gedeclareerd. Op die manier ‘vervuilen’ zij niet de interface die de gebruiker van onze class te zien krijgt, en krijgt enkel de programmeur van de class de flexibiliteit om die **private** methods te wijzigen, te splitsen, samen te voegen, ... zonder dat de gebruiker van onze class er weet van heeft en/of zijn programma moet aanpassen.

Variabelen worden zelden of nooit **public** gedeclareerd. Als de waarde van een variabele moet opgevraagd en/of gewijzigd worden, dan voorzien we daarvoor de nodige **public** methods (getters en setters). Dit geeft de bouwer van de class de noodzakelijke controle over wat er met de variabele gebeurt.

Alles wat **niet private** gedeclareerd wordt, behoort tot de ‘interface’ van de class. Dit is ook wat men te zien krijgt in een (standaard) documentatie.

Opmerking:

Naast de mogelijkheden **public** en **private** bestaan er ook nog de opties **protected** en zelfs een default toegangsregel (of **accessmodifier**).

**protected** betekent alleen toegankelijk binnen de class zelf en binnen de kinderen, kleinkinderen, ... van die class (zie ook cursus object oriëntatie).

De **default** accessmodifier is van toepassing indien men zelf geen expliciete keuze maakt tussen **public**, **private** of **protected**. In dat geval krijgen de variabelen en/of de methods **package-access**. 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**.

## 5.10 Een kopie maken van een object

Stel we willen in ons programma een kopie van een bepaald object maken. Het gevaar bestaat dat we dit doen door de reference van het nieuwe object te laten wijzen naar het object waarnaar de eerste reference wijst.

De code :

```
Voorwerp a = new Voorwerp(...);  
Voorwerp b = a;
```

Het probleem is nu dat als we properties van **b** wijzigen, de properties van **a** mee wijzigen en dat is niet altijd – zeg maar zelden - de bedoeling.

Er zijn drie mogelijke oplossingen om een goede kopie te maken van een object.

- Ofwel laten we **b** wijzen naar een nieuw object dat we initialiseren met de properties van **a**.
- Ofwel maken we een speciale constructor die als parameter geen individuele propertywaarden aanvaardt maar wel een object.
- Ofwel maken we een kopie van **a** met een speciale clone-method en laten we **b** ernaar wijzen.

In deze paragraaf gaan we enkel de eerste twee oplossingen bekijken. De derde oplossing zie je in paragraaf 8.4 “Een toepassing op interfaces: cloning – de interface Cloneable” op pagina 94.

Oplossing 1 :



```
Voorwerp a = new Voorwerp(...);  
Voorwerp b = new Voorwerp(a.prop1, a.prop2, ...);
```

In de bovenstaande code gaan we voor de kopie b een nieuw object maken van het type Voorwerp en dit initialiseren aan de hand van een constructor die alle properties van een dergelijk object aanvaardt.

Oplossing 2 :

```
Voorwerp a = new Voorwerp(...);  
Voorwerp b = new Voorwerp(a);
```

De eerste oplossing is veruit de eenvoudigste oplossing. Er is echter wel één groot nadeel: elke keer we een kopie willen van een object moeten een nieuw object maken én initialiseren met alle properties van het originele object. Als het om een klant gaat met 40 properties dan moet je telkens code tikken waarin 40 properties worden overgezet.

Voor de tweede oplossing hebben we een nieuwe constructor (een **copy constructor**) nodig die als parameter een gelijkaardig object aanvaardt. Binnen de constructor initialiseren we alle properties van het object met de waarden van de properties van het parameterobject. De code voor deze constructor is dan als volgt :

```
public Voorwerp(Voorwerp a) {  
    setProp1(a.prop1);  
    setProp2(a.prop2);  
}
```

Iedere keer dat we nu een kopie maken van een object moeten we enkel een nieuw object maken en het origineel meegeven als parameter. Het kopiëren van de property-waarden moet slechts één maal uitgetikt worden, namelijk in de constructor en niet elke keer we een kopie maken.

## 5.11 Oefeningen



Maak oefeningen 12 t.e.m. 14 uit de oefenmap.



## Hoofdstuk 6 Inheritance

---

### 6.1 Introductie

Inheritance (overerving) betekent dat je een class maakt 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: de nieuwe class erft (krijgt) de (public en protected) data members en methods van de bestaande class. Daarnaast kan de nieuwe class nog eigen data members en methoden bevatten.

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

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

Het realiseren van een afleiding gebeurt door in de class-header van de subclass het woordje **extends** te vermelden gevolgd door de naam van de superclass.

Een voorbeeld van waar je inheritance kan toepassen, zijn de classes ZichtRekening en SpaarRekening:

- Beide classes bevatten de data members saldo en rekeningNummer.
- Beide classes bevatten gemeenschappelijke methods: storten(), afhalen(), overschrijven(), geefSaldo().

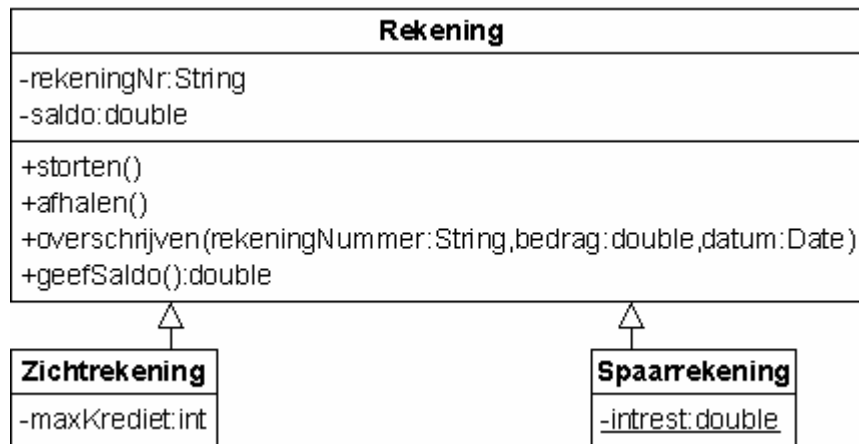
In plaats van deze gemeenschappelijke data members en methods twee keer uit te werken (in de class ZichtRekening én in de class SpaarRekening), is het interessanter deze gemeenschappelijke data members en methods één keer uit te werken in een base class: Rekening.

Daarna maak je de derived classes ZichtRekening en SpaarRekening, die erven van de class Rekening. Deze classes hebben dan automatisch de data members en methods van de base class Rekening ter beschikking.

Data members die enkel voor een spaarrekening gelden, beschrijf je in de class SpaarRekening. Bijvoorbeeld het data member intrest.

Data members die enkel voor een zichtrekening gelden, beschrijf je in de class ZichtRekening. Bijvoorbeeld het data member maxKrediet, zijnde het bedrag dat je in het rood mag gaan op een zichtrekening.

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



De class **ZichtRekening** erft van de class **Rekening** en voegt nog een data member toe dat enkel geldt voor een zichtrekening (maar niet voor andere rekeningen): `maxKrediet`. Dit data member bevat per **ZichtRekening** object het bedrag dat je maximaal mag in het rood gaan.

De class **SpaarRekening** erft van de class **Rekening** en voegt nog een data member toe, dat enkel geldt voor een spaarrekening: `intrest`. Gezien het intrestpercentage voor alle **SpaarRekening** objecten gelijk is, is dit een data member met class bereik.

Als achteraf de base class (**Rekening**) uitgebreid wordt met extra data members of methods, worden deze automatisch geërfd in de derived classes (**ZichtRekening**, **SpaarRekening**).

Voorbeeld: als de base class **Rekening** uitgebreid wordt met een data member `creatiedatum` (de datum waarop de rekening aangemaakt werd), is dit data member ook ter beschikking in de derived classes (**ZichtRekening**, **SpaarRekening**).

In Java:

Laten we ons concentreren op de overervingsstructuur:

- *We werken terug verder met het project **IntroInKlassen**. Maak een nieuwe class **Rekening**:*

```

public class Rekening {
    private String rekeningNr;
    private double saldo;

    public Rekening(String rnr) {
        rekeningNr = rnr;
        saldo = 0;
    }

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



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

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

public double geefSaldo(){
    return saldo;
}
}
```

- *Pas nu de class SpaarRekening aan zoals hieronder weergegeven :*

```
public class SpaarRekening extends Rekening {
    private static double intrest;

    public SpaarRekening(String rnr, double intrest) {
        super(rnr);
        this.intrest = intrest;
    }
}
```

- *En maak een nieuwe class ZichtRekening:*

```
public class ZichtRekening extends Rekening {
    private int maxKrediet;

    public ZichtRekening(String rnr, int bedrag) {
        super(rnr);
        maxKrediet = bedrag;
    }
}
```

- *En ook de class Bankbediende die de beheerder is van de rekeningen:*

```
public class BankBediende {  
    public static void main(String[] args) {  
        SpaarRekening spaar =  
            new SpaarRekening("BE12 3456 7890 1234", 2.5);  
        ZichtRekening zicht =  
            new ZichtRekening("BE98 7654 3210 9876", 1000);  
        spaar.storten(100.0);  
        zicht.storten(200.0);  
        zicht.afhalen(50.0);  
        System.out.println("Saldo van de spaarrekening: " +  
            spaar.geefSaldo());  
        System.out.println("Saldo van de zichtrekening: " +  
            zicht.geefSaldo());  
    }  
}
```

#### Verklaringen:

- De class Rekening is de verzameling van alle gemeenschappelijke data members en methods. De classes SpaarRekening en ZichtRekening bevatten de gespecialiseerde gegevens. Men spreekt in dit verband ook over **generalisatie** en **specialisatie**.
- De hiërarchische relatie wordt opgebouwd via het sleutelwoord **extends**. Zie bij de SpaarRekening (en ook bij de ZichtRekening) in de class header. Door deze constructie erven de kinderen alle public, protected en, indien in dezelfde package, de default data members en methods.  
Uitzondering: een constructor wordt nooit overgeërfd!
- Heeft de superclass een expliciete constructor (zoals hier het geval is), dan moeten de kinderen ook een expliciete constructor hebben en moet, in de constructor van de kinderen, als eerste instructie de constructor van de parent opgeroepen worden. Dit gebeurt via het keyword **super** (wat altijd een verwijzing is naar de parent van een class). Zie in de voorbeelden in de constructor van zowel de SpaarRekening als de ZichtRekening.

Voeren we de class Bankbediende uit, dan geeft dit volgend resultaat:

```
Saldo van de spaarrekening: 100.0  
Saldo van de zichtrekening: 150.0
```



## 6.2 Method overriding

Het bovenstaande voorbeeld werkt zoals verwacht, maar is nog verre van perfect. Wat doen we bijv. met de kredietlimiet **maxKrediet** die is ingesteld voor een zichtrekening en wanneer gebruiken we de **intrest** van de spaarrekening?

Het is dus duidelijk dat de class `ZichtRekening` een aparte versie moet hebben van de methods `afhalen()` en `overschrijven()` (het saldo mag niet onder de limiet komen) en dat de class `SpaarRekening` een eigen versie moet hebben van de method **`geefSaldo()`**, waarbij er rekening wordt gehouden met de toe te passen intrest.

We passen nu de classes `ZichtRekening` en nadien ook `SpaarRekening` aan. De classes `Rekening` en `BankBediende` blijven ongewijzigd.

- *Aan de class `ZichtRekening` voeg je nu een method `afhalen()` toe:*

```
@Override //Is een annotation: dit komt later aan bod
public void afhalen(double bedrag) {
    double testSaldo = geefSaldo() - bedrag + maxKrediet;
    if (testSaldo > 0) super.afhalen(bedrag);
}
```

Deze method heeft dezelfde signatuur als deze in de superclass (zelfde naam, zelfde returntype en zelfde parameterset). Vandaar de annotation `@Override` waarover later meer.

Enkel wanneer het nieuwe saldo groter blijft dan de kredietlimiet wordt de afhaling uitgevoerd. Door het feit dat we nog niet weten hoe met **exceptions** om te gaan, wordt er geen oplossing geboden voor die gevallen waar de kredietlimiet overschreden wordt. Dergelijke oplossingsmethoden komen later aan bod bij de exceptionafhandeling.

Door het feit dat wij in de superclass de variabele **saldo** private hebben gedeclareerd, kunnen we deze in de childclass niet rechtstreeks aanspreken. Wij moeten het saldo dus opvragen via de method van de superclass.

Als de kredietlimiet niet overschreden wordt kunnen we het geld afhalen van de rekening met de method `afhalen()` van de superclass. Vandaar op coderegel: ***`super.afhalen(bedrag)`***.

We hadden de variabele **saldo** ook **protected** kunnen maken. Dan konden we deze in `ZichtRekening` wel rechtstreeks aanspreken.

- *We passen nu ook de class **SpaarRekening** aan. Voeg hier een method **geefSaldo()** toe.*

```
@Override
public double geefSaldo() {
    double rente = berekenRente();
    storten(rente);
    return super.geefSaldo();
}

private double berekenRente() {
    double saldo = super.geefSaldo();
    return saldo * intrest / 100.0;
}
```

De eigen versie van de method **geefSaldo()** gebruikt een private hulpmethod **berekenRente()**. Verder dezelfde opmerkingen als bij **ZichtRekening**.

- *Voeg nu in de class **Bankbediende** onderaan volgende regels toe:*

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

- *Test het programma uit.*

Dit geeft volgend resultaat:

```
Saldo van de spaarrekening: 102.5
Saldo van de zichtrekening: 150.0
Saldo van de zichtrekening (na poging afhalen 2000): 150.0
```

Bij de spaarrekening is er 2.5% rente bijgeteld.

Bij de zichtrekening is de eerste afhaling (50.0) verwerkt.

De tweede afhaling (2000), die het saldo kleiner zou maken dan de toegestane kredietlimiet (1000), wordt niet uitgevoerd. Het saldo blijft ongewijzigd!

Method-overriding is dus een manier om een subclass een eigen variant te geven van een bestaande method uit de superclass.

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



## 6.3 Final methods en final classes

Maakt men in een superclass één method (of meerdere) **final**, dan kan deze in de subclass niet meer **overridden** worden. Probeert men het toch, dan geeft de compiler de volgende fout:

```
afhalen(double) in ZichtRekening cannot override afhalen(double)
in Rekening; overridden method is final
```

Maakt men een class **final**, dan is dit het einde van de overervingstructuur. Er kan niet meer verder gespecialiseerd worden.

Probeert men het toch, dan geeft de compiler de volgende fout:

```
cannot inherit from final Rekening
```

## 6.4 Methods van Object overriden

### 6.4.1 toString()

Alle classes erven impliciet van de **Object**-class. Geven wij geen keyword **extends** mee in de classdefinitie, dan zorgt de compiler ervoor dat deze wordt aangevuld met **extends Object**.

Via die overervingsstructuur krijgen we de beschikking over een aantal methods die meestal te algemeen zijn om echt bruikbaar te zijn. Wij zullen ze in onze eigen classdefinitie overriden.

Een voorbeeld: de method **toString()**. Deze method geeft ons een String-representatie van het object, waarbij meestal de properties worden getoond zonder opmaak. De **toString()**-method wordt automatisch aangeroepen telkens wanneer wij een object in een string-context gebruiken: bijv. `System.out.println(...)`.

Het resultaat van de overgeërfde method voor onze spaarrekening:

```
System.out.println(spaar);
geeft
introinklassen.SpaarRekening@23fc4bec
```

Dit geeft de packagenaam, gevolgd door een punt met daarna de naam van de class, vervolgens een @ en daarna een intern id van het object. Dit betekent dat de **toString()**-method in de class `SpaarRekening` nog niet is overriden. We erven deze dan van class `Object`, maar de informatie is onbruikbaar.

Het is aanbevolen om in elke class die je schrijft de method **toString()** te overriden.

We zullen dit doen in de class `SpaarRekening`, maar deze is afgeleid van `Rekening`, dus schrijven we ook een **toString()** method in class `Rekening`:



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

```
@Override //Is een annotation: dit komt later aan bod
public String toString() {
    return rekeningNr + ", " + saldo;
}
```

- Voeg in SpaarRekening volgende *toString()* method toe.

```
@Override
public String toString() {
    return super.toString() + ", " + intrest;           (1)
}
```

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

De bankbediende krijgt nu na de oproep `System.out.println(spaar)` ; het volgende te zien:

```
BE12 3456 7890 1234, 102.5, 2.5
```

Een vuistregel is dat de `toString()` method een String-representatie van het object levert en dus enkel de waardes van de properties bevat.

Wil je meer opmaak in de output, dan schrijf je hiervoor best een andere method.

- Voeg bijvoorbeeld in SpaarRekening volgende *toon()* method toe.

```
public void toon() {
    System.out.println("Dit is een spaarrekening waarop " +
        intrest + "% intrest gegeven wordt.");
}
```

De bankbediende krijgt nu na de oproep van `spaar.toon()` ; het volgende te zien:

```
Dit is een spaarrekening waarop 2.5% intrest gegeven wordt.
```

In een dergelijke method bepaal je dus volledig zelf welke informatie er getoond wordt.



### 6.4.2 clone()

Een andere interessante method die wij erven van de `Object`-class is de method `clone()`. Deze maakt een kopie van een object. Er is wel een voorwaarde aan verbonden: de class (van het object dat men wil clonen) moet de interface ***Cloneable*** implementeren. Wat dit precies betekent zie je in Hoofdstuk 8 Interfaces. Onthou voorlopig dat in de class header achteraan “implements Cloneable” moet staan.

Bijv. 

```
public class SpaarRekening extends Rekening
    implements Cloneable {
```

Wanneer komt dit nu van pas? Eerder in deze cursus gaven we aan dat wanneer je een object als parameter meegeeft aan een method, je dan werkt op het originele object. Wil men dit vermijden, dan geeft men een kopie (een clone) mee aan de method i.p.v. het origineel.

Bijv. 

```
SpaarRekening spaar =
    new SpaarRekening("BE56 7890 1234 5678", 3.6);
SpaarRekening kopiespaar = spaar.clone();
```

Deze implementatie van de clone method zoals voorzien in de class `Object` werkt alleen probleemloos wanneer de class en zijn base class alleen data members bevat van een primitive data type. Wanneer er reference types gebruikt zijn als data member voldoet deze oplossing dus niet. Je vindt meer informatie over dit onderwerp wanneer je in je favoriete zoekmachine de termen "shallow copy" en "deep copy" opzoekt of wanneer je later de paragraaf “Een toepassing op interfaces: cloning – de interface Cloneable” op pagina 94 doorneemt.

### 6.4.3 equals()

Een derde method die we erven van de ***Object***-class is de ***equals()*** method. Ook hier geeft de standaard-code weinig of geen significante code.

Wanneer zijn twee strings bijvoorbeeld aan elkaar gelijk? Volgens de standaard equals als het om hetzelfde object gaat (dus twee referencevariabelen die naar hetzelfde object wijzen). Beter is het om in de eigen classes zelf een equals-method te definiëren (als een overriding op de standaard) en daarin zelf te bepalen wat 'gelijkheid' inhoudt. Twee auto-objecten zijn voor mij bijvoorbeeld gelijk als het hetzelfde merk, hetzelfde type van hetzelfde bouwjaar is. Voor mij speelt de kleur niet om over gelijkheid te spreken. Voor iemand anders kan dit anders zijn. Vandaar: als bouwer van een class dient er bepaalt te worden wat gelijkheid is.

## 6.5 Abstracte classes en methods

### 6.5.1 Abstracte classes

Kijken we naar het voorbeeld vanuit het standpunt van de bank, dan bestaan er geen rekeningen. Er zijn wel zichtrekeningen, spaarrekeningen, termijnrekeningen, jongerenrekeningen, ... maar het algemeen begrip ‘rekening’ bestaat niet.

Dit kan weerspiegeld worden in de Java-implementatie:

We noemen een dergelijke class ***abstract*** en dat schrijf je zo :

```
public abstract class Rekening {
```

De enige consequentie is dat er geen objecten meer kunnen geïntanceerd worden van de class **Rekening**.

Dit voorkomt fouten en biedt toch de mogelijkheid om algemene kenmerken te groeperen in één class.

- *Maak van de class Rekening een abstracte class.*

### 6.5.2 Abstracte methods

Vooraleer we het over abstracte methods hebben gaan we eerst ons voorbeeld vereenvoudigen: de variabele **saldo** krijgt als accessmodifier **protected** en de methods **afhalen()** en **geefSaldo()** in de childclasses worden aangepast.

Dit geeft volgende code:

- Wijzig in de class Rekening de volgende regel:

```
protected double saldo;
```

*De accessmodifier is nu **protected**.*

- Wijzig in de class ZichtRekening de method afhalen() in:

```
public void afhalen(double bedrag) {  
    double testSaldo = saldo - bedrag + maxKrediet;  
    if (testSaldo > 0) saldo -= bedrag;  
};
```

*Er kan nu rechtstreeks gewerkt worden met de variabele saldo.*

- Wijzig ook in de class SpaarRekening de method geefSaldo() als volgt:

```
@Override //Is een annotation: dit komt later aan bod  
public double geefSaldo() {  
    double rente = berekenRente();  
    saldo += rente;  
    return saldo;  
}  
private double berekenRente() {  
    return saldo * interest / 100.0;  
};
```

*Ook hier kan rechtstreeks met de variabele saldo gewerkt worden.*



Veronderstellen we nu dat er bij een zichtrekening kosten aangerekend worden (0.05 EUR) telkens men het saldo opvraagt (afdrukken + opsturen van de rekeninguittreksels).

De class `ZichtRekening` krijgt dan ook zijn eigen variante van de method **`geefSaldo()`**.

- *Voeg deze code toe in `ZichtRekening`:*

```
@Override //Is een annotation: dit komt later aan bod
public double geefSaldo() {
    saldo -= 0.05;
    return saldo;
}
```

Alvorens de code uit te voeren, doen we ook nog één ingreep in de class `BankBediende`: we gaan ervoor zorgen dat de getallen netjes worden afgebeeld: 2 cijfers na het decimaal teken en, indien relevant, groepering van de cijfers.

- *Wijzig de code in `BankBediende`:*

```
import java.text.*;

public class BankBediende {

    public static void main(String[] args) {
        DecimalFormat fmt = new DecimalFormat("#,##0.00");
        SpaarRekening spaar =
            new SpaarRekening("BE12 3456 7890 1234", 2.5);
        ZichtRekening zicht =
            new ZichtRekening("BE98 7654 3210 9876", 1000);
        spaar.storten(100.0);
        zicht.storten(200.0);
        zicht.afhalen(50.0);
        System.out.println("Saldo van de spaarrekening: " +
            spaar.geefSaldo());

        System.out.println("Saldo van de zichtrekening: " +
            fmt.format(zicht.geefSaldo()));
        zicht.afhalen(2000);
        System.out.println("Saldo van de zichtrekening " +
            "(na poging afhalen 2000): " +
            fmt.format(zicht.geefSaldo()));
    }
}
```

Bovenaan de code komt er een import-instructie aangezien wij een class **`DecimalFormat`** gaan gebruiken die in deze package zit.

In de main definiëren we eerst een object **fmt** dat een formatteringsmasker krijgt.

Verder in de code wordt de formattering uitgevoerd (er gebeurt zelfs een automatische conversie naar de lokale settings: komma als decimaal teken, ...).

Het resultaat:

```
Saldo van de spaarrekening: 102.5
Saldo van de zichtrekening: 149,95
Saldo van de zichtrekening (na poging afhalen 2000): 149,90
```

Na deze aanpassing kunnen we ons de vraag stellen of de method **geefSaldo()** in de class Rekening nog moet blijven bestaan. Ze wordt nooit meer uitgevoerd. Het antwoord is ja, maar dan als abstracte method.

Wijzig daartoe het volgende in de code van de class Rekening:

```
public abstract double geefSaldo();
```

De method-definitie is gereduceerd tot één regel!

Voordelen:

- Door het feit van een abstracte method **geefSaldo()** te voorzien in de superclass worden alle kinderen verplicht om een eigen, concrete implementatie te maken van deze method. Men kan het in de child-class niet meer vergeten (de compiler waakt hierover).
- Het behouden van de abstracte method in de superclass geeft faciliteiten bij polymorphism (meer hierover later in de cursus).

Opmerking:

Eens men één abstracte method heeft in een class, moet ook de class als abstract worden gedeclareerd !

## 6.6 Polymorphism

We passen ons voorbeeld nogmaals aan: onze bankbediende heeft (in werkelijkheid) veel meer rekeningen te beheren dan de twee die we tot hier toe gebruikten. Daarom gaan we voor een oplossing met een array.

```
import java.text.*;

public class BankBediende {

    public static void main(String[] args) {
        DecimalFormat fmt = new DecimalFormat("#,##0.00");
        Rekening[] rekeningen = new Rekening[100];
        rekeningen[0] =
```



```
        new SpaarRekening("BE12 3456 7890 1234", 2.5);
rekeningen[1] =
    new ZichtRekening("BE98 7654 3210 9876", 1000);
rekeningen[0].storten(100.0);
rekeningen[1].storten(200.0);
rekeningen[1].afhalen(50.0);
for (int i=0;i<rekeningen.length;i++) {
    if (rekeningen[i] != null)
        System.out.println("Saldo van de rekening: " +
            fmt.format(rekeningen[i].geefSaldo()));
}
}
```

1. *Wijzig de code van BankBediende en voer het programma uit.*

Het resultaat:

```
Saldo van de rekening: 102,50
Saldo van de rekening: 149,95
```

Verklaring:

- Na de formaatdefinitie wordt er een array gemaakt van 100 Rekening-objecten. Alhoewel de class Rekening abstract is en er dus geen objecten kunnen van geïnstantieerd worden, kan men een abstracte class toch gebruiken als 'data-type'.
- De eerste twee elementen van de array wijzen vervolgens naar een nieuwe SpaarRekening en een ZichtRekening. Dit wordt aanvaard omdat een SpaarRekening en een ZichtRekening ook Rekeningen zijn. Men spreekt in dit verband over een '**is a**' relatie (dit in tegenstelling tot de relatie die er bestaat tussen de bankbediende en de rekeningen: dit is een associatie of een '**has a**' relatie).  
Het proces om een SpaarRekening (een meer gedetailleerde versie van een algemene rekening) te beschouwen als een algemene Rekening, noemt men **upcasting**. Dit kan impliciet gebeuren (zoals in ons voorbeeld) maar het kan ook expliciet via typecasting.
- Helemaal onderaan de code wordt de method **geefSaldo()** opgeroepen op het algemene Rekening-object en toch wordt, per object, de specifieke versie van de method uitgevoerd. Voor een object dat (in werkelijkheid) een SpaarRekening is, wordt bij het uitvoeren van **geefSaldo()** rekening gehouden met de intrest, en bij een object dat (in werkelijkheid) een ZichtRekening is, wordt rekening gehouden met de kosten.  
De algemene Rekening-objecten gedragen zich dus soms als een SpaarRekening, soms als een ZichtRekening. Dat noemen we **polymorphism**.

Voorwaarde:

Polymorphism werkt alleen maar voor die methods die in alle child-classes aanwezig zijn **en** in de superclass. In de superclass mag die method abstract zijn.

Voordelen:

Vereenvoudiging van de code. Veronderstel dat in ons voorbeeld er veel meer soorten rekeningen zijn, dan moet, zonder polymorphism, de bankbediende iedere soort apart beheren. Met polymorphism zorgt de JVM dat de juiste versie van de method wordt uitgevoerd. Dit proces, het bepalen van de juiste versie, gebeurt bij uitvoering. Men spreekt dan ook van **late-binding** of van **dynamic binding**.

Opmerking:

Het omgekeerde van **upcasting** is **downcasting**.

Dit kan alleen expliciet uitgevoerd worden via typecasting:

Bijvoorbeeld:

```
SpaarRekening nieuw = (SpaarRekening) rekening[0];
```

Dit houdt zekere gevaren in. Is men wel zeker dat in rekening[0] een SpaarRekening zit? Misschien is het een ZichtRekening?

Bij verkeerde downcasting ontstaat er een error: een **ClassCastException**.

Om dit te vermijden kan (moet?) men eerst testen of het wel een SpaarRekening is. Hiervoor bestaat de operator **instanceof**.

Betere code zou dus zijn:

```
if (rekening[0] instanceof SpaarRekening)
    SpaarRekening nieuw = (SpaarRekening) rekening[0];
```

## 6.7 Annotations

In een vorige paragraaf hebben we gezien dat we heel makkelijk een method uit een baseclass kunnen overriden in een subclass. Misschien wel iets té gemakkelijk. Wanneer we een kleine tikfout maken en een baseclass method geefSaldo() overschrijven met een method geefsaldo() (dus met kleine s) dan geeft de compiler helemaal geen fout maar beschouwt dit als een totaal andere method.

Nu bestaat er zoiets als een “annotation”. Dit is informatie voor de compiler (of in andere gevallen voor de JVM), zodat deze de nodige controles kan uitvoeren. In dit geval of er, binnen de overervingsstructuur, wel ergens een method bestaat met dezelfde naam.

De annotation die we nodig hebben voor dergelijke controle is **@Override**. We plaatsen dit vooraan de methodheader.

We proberen het even uit:



2. *Wijzig de code van `BankBediende` en haal 5000 af van de zichtrekening ipv 50.*

```
rekeningen[1].afhalen(5000.0);
```

3. *Voer het programma uit en je ziet dat behalve de kosten (0.05) er niets afgaat wegens het maximumkrediet.*

*Wijzig nu de code in `ZichtRekening`, namelijk schrijf de method `afhalen` nu met een hoofdletter: `Afhalen()`.*

```
public void Afhalen(double bedrag) {  
    ...
```

4. *Voer het programma opnieuw uit. Deze keer wordt er wel 5000 afgehaald ondanks het opgelegde maximumkrediet. Het is namelijk de method `afhalen()` van de baseclass die uitgevoerd wordt.*

*Om dit soort pijnlijke misverstanden te vermijden plaats je `@Override` vooraan de method header.*

```
@Override  
public void Afhalen(double bedrag) {  
    ...
```

5. *Je merkt dat je een foutboodschap krijgt van de compiler. Deze merkt dat je een method probeert de overriden die er in de superclass niet is.*

*“...method does not override a method from its superclass”*

*Wijzig de beginletter van de method terug naar een kleine letter.*

Een ander voorbeeld is het overriden van de `toString()` method en de `equals()` method. Vóór deze methods gaan we voortaan ook de annotation `@Override` plaatsen.

In het verdere verloop van het java-traject zal je nog meer over annotations horen. Je weet dus alvast wat het nut ervan is.

## 6.8 Oefeningen



Maak oefening 15 en 16 uit de oefenmap.



## Hoofdstuk 7 Strings

### 7.1 Introductie

Strings zijn geen primitive datatypes. In Java zijn strings geïmplementeerd als objecten afgeleid van de class `String`, met data members en methods.

Een normale creatie van een string zou dus als volgt moeten:

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

Omwille van de alom tegenwoordigheid van strings heeft men een afwijking ingebouwd en wordt ook volgende creatie toegelaten:

```
String tekst = "abcd";
```

Voor een overzicht van alle constructors en andere methods: zie de Java-API.

Een string is onveranderbaar (**immutable**). Dit betekent dat als een `String` een bepaalde waarde heeft, je deze waarde niet kan wijzigen. Is onderstaande code waarin strings worden samengevoegd dan fout?

```
String tekst = "abcd";  
tekst += "efgh";
```

Neen, ook dit is ingevoerd voor het comfort van de programmeur. Wat er in feite gebeurt, is het volgende:

- Na de eerste regel bestaat er een referencevariabele *tekst* die verwijst naar een stringobject, waarvan één van de data members de waarde "abcd" heeft.
- Bij uitvoering van de tweede regel wordt er een nieuw stringobject gemaakt, dat voldoende groot is om alles te bevatten. De waarde van de property wordt nu "abcdefgh". De bestaande referencevariabele *tekst* krijgt nu een nieuwe inhoud: hij verwijst nu naar dat nieuwe object. Het oude object (de oorspronkelijke string met inhoud "abcd") wordt door de **garbage collector** verwijderd (indien er geen andere references meer zijn naar dit object).

Dit heeft wel gevolgen voor strings die we als parameter meegeven in een method-aanroep. Binnen de method wordt wel een nieuwe referencevariabele aangemaakt die naar de originele string verwijst. Doen we, in de method, via de eigen referencevariabele aan concatenation, dan wordt de lokale referencevariabele gewijzigd zodat deze naar de nieuwe string wijst. Vanaf dat ogenblik werken wij in de method niet meer op de originele string!



Moeten we in een class (zeer) veel concatenations uitvoeren, dan ontstaan er zeer veel nieuwe objecten en heeft de garbage collector veel werk. Dit is geen efficiënte manier van werken. Een (beter) alternatief is dan het gebruik van de **StringBuffer**-class die wel concatenations toelaat (zie 7.6).



## 7.2 Speciale tekens in een string

Stringwaarden worden tussen dubbele aanhalingstekens geplaatst. Maar wat indien binnen de stringwaarde zelf aanhalingstekens nodig zijn?

We gebruiken dan het escape-teken (zie ook 2.2.2 Variabelen, references en literals).

Voorbeeld:

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

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

De normale betekenis (voor Java) van het dubbelaanhalingsteken is **stringdelimiter**, de alternatieve betekenis is een normaal aanhalingsteken.

De belangrijkste speciale tekens en hun escaped value (er zijn er meer):

	Betekenis
\\	backslash
\n	new line
\t	tab
\r	return

## 7.3 Bewerkingen met strings

### 7.3.1 Het vergelijken van strings

Beschouwen we volgende code:

```
String tekst1 = "abcd";  
tekst1 = tekst1.toUpperCase();  
String tekst2 = "ABCD";  
System.out.println(tekst1 == tekst2);
```

In bovenstaande code definiëren we een string "abcd". Deze string zetten we om naar hoofdletters met een method `toUpperCase()`. Meer over dit soort methods zo meteen. Vervolgens definiëren we een tweede string "ABCD" die dus gelijk zou moeten zijn aan de eerste string. Dit is echter niet zo. Het resultaat zal **false** zijn niettegenstaande beide strings dezelfde tekst bevatten.

De reden hiervoor is de volgende:

*tekst1* is een referentievariabele die na de bewerking verwijst naar een stringobject met inhoud "ABCD". 'Verwijzen naar' betekent dat in *tekst1* het *adres* zit van de geheugenplaats waar dat eerste stringobject is opgebouwd.

*tekst2*: idem, *tekst2* heeft als inhoud een geheugenadres dat verwijst naar een (ander) stringobject waarvan de inhoud ook "ABCD" is.

Bij de vergelijking ***tekst1 == tekst2*** testen we dus eigenlijk of de twee geheugenadressen hetzelfde zijn, en dat is niet zo!

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

- ***equals()***: een hoofdlettergevoelige vergelijking: ***tekst1.equals(tekst2)*** zal wel **true** opleveren.
- ***equalsIgnoreCase()***: zoals voorgaande, maar dan niet hoofdlettergevoelig.
- ***compareTo()***: 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.



Opmerking:

Indien we in het bovenstaande voorbeeld beide stringvariabelen zouden initialiseren op "abcd" (en verder niet veranderen) dan zou het resultaat wél true zijn. Dit is echter een uitzondering omdat de compiler merkt dat er in de source twee keer dezelfde string literal wordt gebruikt. De compiler voert een optimalisatie uit en laat beide string variabelen tóch naar dezelfde plaats in het geheugen verwijzen !

### 7.3.2 Strings wijzigen

Laten we even een aantal methods van het object String op een rijtje zetten.

Opgelet: zoals gezegd op pagina 79 is een string niet wijzigbaar. Alle volgende methods geven dus een nieuwe string terug!

- ***replace()***: vervangt in een string een teken (character) door een ander teken.

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

Voorbeeld:

```
String tekst = "banaan";  
String resultaat = tekst.replace('a', 'o');
```

⇒ na de verwerking bevat het object *resultaat* de tekst "bonoon".



- **toLowerCase():** zet de inhoud van een string om in 'kleine' letters. Enkel de hoofdletters worden beïnvloed. Cijfers, leestekens, ... blijven ongewijzigd.

```
public String toLowerCase()
```

Voorbeeld:

```
String tekst = "BANAAN";  
String resultaat = tekst.toLowerCase();
```

⇒ na de verwerking bevat het object *resultaat* de tekst "banaan";

Opgelet: de originele string *tekst* blijft ongewijzigd!

- **toUpperCase():** idem voorgaande, maar een omzetting naar hoofdletters.
- **trim():** verwijdert de witruimte voor en achter de tekst.  
Witruimte: spaties, tabs, new-lines, ...

### 7.3.3 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 `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 eindindex-1.

De index begint altijd bij 0 (zie voorbeeld controle rekeningnummer in paragraaf 5.6). Deze method kan fouten (exceptions) genereren indien `beginindex < 0`, `eindindex > lengte van de string` of `eindindex < beginindex`.

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

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

Ook hier bestaan er exceptions!

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

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

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



De meeste van bovenstaande String-methods hebben één of meer overloaded varianten. Voor de volledige lijst en voor voorbeelden: zie de API.

### 7.3.4 Een voorbeeld

Als voorbeeld op het gebruik van String-methods controleren we een email-adres 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 deze mag niet in de eerste en niet in de laatste positie staan
- Na de @ moet er minstens één punt komen

```
public class EmailControle
{
    private static String email1 = "kamiel.kafka@praag.be";
    private static String email2 = "kamiel@kafka@praag.be";
    private static String email3 = "kamiel.kafka@";

    public static void main(String[] args)
    {
        controleer(email1);
        controleer(email2);
        controleer(email3);
    }

    private static void controleer(String s)
    {
        String antw="";
        int plaats;
        int lengte = s.length();
        if (lengte < 4) antw.concat("e-mail adres is te kort\n");
        plaats = s.indexOf('@');
        if (plaats < 0)
            antw += "Er moet een @ in het adres voorkomen.\n";
        if (plaats == 0 || plaats == (--lengte))
            antw += "De @ mag niet helemaal voor- of achteraan 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.";
        if (antw.length() == 0)
        {
```



```
        antw = "Alle controles zijn goed bevonden";
    }
    System.out.println(antw);
}
}
```

## 7.4 Strings: conversie van en naar primitive types

In de praktijk zal je dikwijls getallen voorgesteld als een String willen omzetten naar een gepast primitive type. Omgekeerd kan je een getal dat in een variabele is opgeslagen dat van een primitive type is converteren naar een String.

Laten we starten met het omzetten van een String naar een primitive type. Hiervoor heeft elk primitive type een method `parseX(String str)` waarbij je X vervangt door het primitive type.

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);
```

Voor de omgekeerde beweging, van primitive type naar String gebruik je `String.valueOf()`. 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);
```

## 7.5 Strings opsplitsen

### 7.5.1 De method `String.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 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 test = "Dit is een stukje tekst";
String[] stukskes = test.split(" ");

for (int i=0;i<stukskes.length;i++) {
    System.out.println(stukskes[i]);
}
```

Het resultaat :

```
Dit
is
een
stukje
tekst
```

De tekst wordt netjes in 5 stukken gesplitst en in de array gestopt.

Een voorbeeld met ook de int-parameter :

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

Het resultaat :

```
Dit
is
een stukje tekst
```

De tekst wordt deze keer aan de hand van de delimiter  $n-1$  keer gesplitst zodat in het  $n^e$  arrayelement de rest van de tekst komt te staan.

### 7.5.2 De class StringTokenizer

Voor de volledigheid vermelden we hier de class StringTokenizer die eveneens een string in verschillende stukken kan splitsen.



De class StringTokenizer wordt gebruikt om een string te splitsen in verschillende delen, gebaseerd op een 'scheidingsteken'. Een voorbeeld :

```
/*
 * StringTokenizer
 */
import java.util.*;

public class StringTok
{
    public static void main(String[] args)
    {
        String tekst = "Een demo van de class StringTokenizer";
        StringTokenizer strTok = new StringTokenizer(tekst, " ");
        while (strTok.hasMoreTokens()) {
            System.out.println(strTok.nextToken());
        }
    }
}
```

We definiëren eerst een string en een object van het type StringTokenizer. Deze laatste initialiseren we a.d.h.v. de string. We geven ook aan dat we een spatie gebruiken als delimiter. In een lus vragen we de verschillende delen van de String op.

Het resultaat:

```
Een
demo
van
de
class
StringTokenizer
```

Opmerkingen:

- De class StringTokenizer zit in de java.util-package, vandaar de import bovenaan.
- De belangrijkste methods zijn:
  - hasMoreTokens()* wat true of false oplevert
  - en
  - nextToken()* wat het volgende onderdeel oplevert.



## 7.6 De class StringBuffer

Zoals hiervoor gesteld, is een string niet wijzigbaar. Telkenmale we strings samenvoegen met het **+**-teken (of met de `concat`-method), ontstaat er een nieuwe string, wordt de referencevariabele op het nieuwe adres gezet en wordt de 'oude' string terzelfdertijd opgeruimd door de garbage collector.

Hebben we een toepassing waarin veel strings moeten samengevoegd worden, dan is het efficiënter om een `stringbuffer` te nemen in plaats van een string. De class `StringBuffer` heeft namelijk efficiënte methods om strings samen te voegen (`append`), om strings tussen te voegen (`insert`), enz. en is bovendien niet `immutable`.

Een voorbeeld:

```
StringBuffer leeg=new StringBuffer();
System.out.println(leeg.length()+" "+leeg.capacity());

StringBuffer naam=new StringBuffer("Eddy");
System.out.println(naam.length()+" "+naam.capacity());

StringBuffer teVullen=new StringBuffer(30);
System.out.println(teVullen.length()+" "+teVullen.capacity());

naam = new StringBuffer("germaine_de_coeur_brisee");
for (int teller=0;teller<naam.length();teller++)
    if (naam.charAt(teller)=='_')
        naam.setCharAt(teller, ' ');
System.out.println(naam);

naam = new StringBuffer("Eddy");
naam.append(' ');
naam.append("Wally");
naam.append(1);
System.out.println(naam);
naam.insert(0,"De grote ");
System.out.println(naam);
naam.delete(0, 9);
System.out.println(naam);
```

De uitvoer :

```
0:16
4:20
0:30
germaine de coeur brisee
Eddy Wally1
De grote Eddy Wally1
Eddy Wally1
```



- (1) We maken 3 objecten van het type `StringBuffer` aan: eerst één a.d.h.v. de default constructor. Deze levert ons een `StringBuffer` op met lengte 0 en capaciteit 16. Vervolgens maken we een `StringBuffer` die we een `String`-parameter meegeven. Lengte en capaciteit zijn 4 en 20. De laatste `StringBuffer` geven we geen inhoud mee maar wel een capaciteit zodat we respectievelijk 0 en 30 krijgen te zien voor lengte en capaciteit.
- (2) Hier vervangen we in een `StringBuffer` alle underscores door spaties. Met de method `charAt()` bekijken we een individueel karakter uit de `StringBuffer`. De method `setCharAt()` vervangt een individueel teken uit de `StringBuffer` door een ander teken.
- (3) We initialiseren de `StringBuffer` met de tekst 'Eddy', voegen er een spatie, de tekst 'Wally' en een literal aan toe. We voegen vooraan 'De grote ' toe en verwijderen dan de eerste 9 tekens opnieuw.

Een “verbeterde” versie van de class `StringBuffer` is de class **`StringBuilder`**, beschikbaar vanaf Java 5. Deze class is performanter dan `StringBuffer` maar is niet threadsafe - wat dit laatste betekent zie je in Hoofdstuk 14 Multithreading. Verder heeft deze class dezelfde methods als `StringBuffer`.

## 7.7 Oefeningen



Maak oefeningen 17 t.e.m. 19 uit de oefenmap.

## Hoofdstuk 8 Interfaces

---

In dit hoofdstuk bekijken we hoe we in Java een *interface* kunnen implementeren. Vooraleer we dit doen herhalen (zie ook de cursus Object Oriëntatie) we nog even de betekenissen van het begrip interface.

- De interface van een class is de verzameling van alle niet-private data members en methods, zoals beschreven in de API van die class en die door gebruikers van de class kunnen gebruikt worden.
- Een interface is een abstracte class waarin alleen abstracte methods en eventueel publieke constanten (final data members) opgenomen zijn. Een dergelijke interface wordt aangemaakt met het keyword `interface` in plaats van het gebruikelijke `class`.

Voorbeeld: `public interface Gereedschap {`

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

Een class kan één of meerdere interfaces implementeren. Dit betekent dat alle (!) methods die in de interface(s) gedeclareerd werden, gedefinieerd moeten worden in de class. Een class ondertekent als het ware een contract waarmee de class zegt alle methods uit de interface(s) te zullen beschrijven. Implementeren gebeurt door in de class-header het keyword ***implements*** gevolgd door de namen van de interfaces te vermelden.

Voorbeeld: `public class Graafwerktuig implements Gereedschap {`

Voor de volledigheid vermelden we nog even dat een interface ook constanten kan bevatten en dat een interface ook kan 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 classes en je werkt het gedrag uit door de methods van de interface in de class code uit te schrijven.
- een beperkt beeld geven van een uitgebreide class. Een klant van een bank bekijkt de class `Bank` anders dan een bediende van een bank. Je maakt een interface `IBankKlant` die enkel de methods bevat vanuit het standpunt van een klant. Je maakt een interface `IBankBediende` waarin u enkel de methods plaatst vanuit het standpunt van een bediende. In een programma voor een klant van de bank, benader je een `Bank` object vanuit de interface `IBankKlant`.



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.

## 8.1 Declaratie en beschrijving

Als voorbeeld van een interface beschouwen we een bedrijfskost. In een bedrijf heb je o.a. personeelskosten, materiaalkosten, enz. De interface `Kost` heeft twee methods: `bedragKost()` en `personeelsKost()`.

Daarna laten we drie totaal verschillende objecten, nl. een werknemer, een vrachtwagen en een kopieermachine, de interface `Kost` implementeren.

De interface `Kost`:

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

Deze interface declareert dus twee methods `bedragKost()` en `personeelsKost()`. Een beschrijving van deze methods krijg je hier nog niet te zien maar je kan wel al afleiden dat de method `bedragKost` een `double` zal retourneren en de method `personeelsKost` een `boolean` (is dit al dan niet een personeelskost).

## 8.2 Implementatie in een class

Een eerste object `Werknemer` zal deze *interface implementeren*. In de class `Werknemer` zullen we dus de beide methods uit de interface moeten terugvinden.

```
public class Werknemer implements Kost {
    private String naam;
    private double wedde;
    public Werknemer(String naam,double wedde) {
        this.naam=naam;
        this.wedde=wedde;
    }
    @Override
    public double bedragKost() {
        return wedde;
    }
    @Override
    public boolean personeelsKost() {
        return true;
    }
    public double getWedde() {
        return wedde;
    }
}
```

Op de eerste lijn merk je aan '*implements Kost*' dat deze class de interface Kost implementeert.

In de class Werknemer vinden we eerst twee private data members naam en wedde en een constructor.

Vervolgens worden de beide methods uit de interface geïmplementeerd. In dit voorbeeld hebben we de code vrij eenvoudig gehouden. De method *bedragKost()* retourneert gewoon de waarde van het data member wedde en de method *personeelsKost()* retourneert hier *true* aangezien het inderdaad om een personeelskost gaat.

Vlak voor deze beide methods moet je de annotatie `@Override` plaatsen. We hebben reeds gezien dat je `@Override` plaatst vóór een method die je override van de base class. Vanaf java 6 plaats je ook `@Override` vóór een method die je implementeert vanuit een interface!

Tenslotte is er een method gedefinieerd waarmee de waarde van het data member wedde door de buitenwereld op te vragen is.

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



```
public class Kopieermachine implements Kost {
    private String merk;
    private double kostPerBlz;
    private 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 int getAantalBlz() {
        return aantalBlz;
    }
}
```

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

Verder drie private data members merk, kostPerBlz en aantalBlz en een constructor. Daaronder 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 staat er een method om de waarde van het aantalBlz op te vragen.

We laten een laatste class de interface implementeren, namelijk de class Vrachtwagen. Deze class implementeert echter nog een andere interface Afschrijving. Classes kunnen immers meerdere interfaces implementeren.

Laten we beginnen met de beschrijving van de interface Afschrijving:

```
public interface Afschrijving {
    int termijn();
    double jaarlijksBedrag();
}
```

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 zo uit:

```
public class Vrachtwagen implements Kost, Afschrijving {
    private String merk;
    private double kostPerKm;
    private int aantalKm;
    private double aankoopPrijs;
    private 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;
    }

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

    public int getAantalKm() {
        return aantalKm;
    }
}
```

### 8.3 Interface als een data type

Net zoals we bij inheritance polymorphism (zie Polymorphism p.75) een array kunnen maken van references naar objecten met eenzelfde base class, kunnen we nu een array maken van reference variabelen met als type een interface. Elk van die reference variabelen laten we dan verwijzen naar een object van een class die deze interface implementeert. We kunnen vervolgens voor elk element in de array een method oproepen die gedeclareerd is in die interface. Methods die niet in de interface voorkomen kunnen we niet uitvoeren. Voor de werknemer kan dan bijvoorbeeld getWedde() niet uitgevoerd worden.



In het onderstaande voorbeeldprogramma implementeren de classes `Werknemer`, `Kopieermachine` en `Vrachtwagen` de methods `bedragKost` en `personeelsKost` uit de interface `Kost`. We maken een array van het interface-datatype `Kost` aan, met references naar twee `Werknemers`, een `Vrachtwagen` en een `Kopieermachine`. Daarna doorlopen we de array en sommeren alle personeelskosten en niet-personeelskosten.

```
public class KostProg {
    public static void main(String[] args) {
        Kost[] kosten = new Kost[4];                                (1)
        kosten[0]=new Werknemer("Eddy",2000.0);
        kosten[1]=new Werknemer("Marva",1500.0);
        kosten[2]=new Vrachtwagen("DAF",0.35,25000,150000.0, 8);
        kosten[3]=new Kopieermachine("Konica",0.02,9000);           (2)
        double mensKosten=0.0, andereKosten=0.0;                   (3)
        for (Kost eenKost:kosten)                                   (4)
            if (eenKost.personeelsKost())                           (5)
                mensKosten+=eenKost.bedragKost();
            else
                andereKosten+=eenKost.bedragKost();
        System.out.println("Mens kosten:"+mensKosten);             (6)
        System.out.println("Andere kosten:"+andereKosten);
    }
}
```

- (1) We maken een array van het interface-datatype `Kost`.
- (2) Elk van de references laten we wijzen naar een class die de interface `Kost` implementeert.
- (3) We definiëren twee doubles om de kosten te totaliseren.
- (4) Deze instructie vergt extra uitleg. Hier staat een “collection based for loop”. Ook wel een “enhanced for loop” genoemd of een “for each loop”.  
Je leest deze instructie als volgt: “Doorloop de array `kosten` en stop de elementen één voor één in een variabele `eenKost` die van het type `Kost` is.”  
Meer over de collection based for loop op pagina 126.
- (5) We verhogen het gepaste totaalbedrag...
- (6) ...en voeren het uit.

## 8.4 Een toepassing op interfaces: cloning – de interface `Cloneable`

In de paragraaf “`clone()`” op pagina 72 hadden we het al even over de interface `Cloneable`. We hadden het toen net als in paragraaf 5.10 ‘Een kopie maken van een object’ over het kopiëren van objecten. We herhalen nog even het probleem: het gevaar bestaat dat we dit doen door de reference van het tweede object te laten wijzen naar het object waarnaar de eerste reference wijst.

De code :



```
Voorwerp a = new Voorwerp(...);  
Voorwerp b = a;
```

We gaven al twee oplossingen aan. Ofwel laten we b wijzen naar een nieuw object dat we initialiseren met de properties van a. Ofwel maken we een nieuw object gebruik makend van een constructor (copy constructor) waarin we de propertywaarden van het originele object overnemen.

Een derde oplossing bestaat erin een clone van a te maken en b ernaar te laten wijzen.

De code :

```
Voorwerp a = new Voorwerp(...);  
Voorwerp b = (Voorwerp)a.clone();
```

We demonstreren het gebruik van de clone-method aan de hand van volgend voorbeeld:

In een applicatie hebben we een object Vandaag, een instance van de class Datum. Deze datum is geïnitieerd op 1/7/2005. Op die dag treedt in een bedrijf een nieuwe magazijnier in dienst. Voor deze magazijnier creëren we een nieuw object Magazijnier van de class Bediende. Eén van de data members van deze bediende is zijn DatumInDienst (datumIn). We vullen deze property in door via de clone-method van Datum een kopie te nemen van Vandaag en datumIn ernaar te laten wijzen. Ter controle wijzigen we Vandaag en kijken we of de datumindienst van de magazijnier nog steeds op dezelfde datum staat.

De praktijk nu. We laten de class Datum de interface Cloneable implementeren<sup>1</sup>. Dit betekent dat we de protected method clone() van de class Object moeten overriden en public maken<sup>2</sup>. Deze method heeft als resultaat een Object. Ook de overriden method moet dus een Object retourneren. In ons geval retourneert de method clone() een datum en dat mag aangezien de class Datum net als alle andere classes afgeleid is van Object. De datum die we retourneren (onder de vorm van een Object) initialiseren we a.d.h.v. een nieuwe constructor<sup>3</sup> die als parameter een Datum-object aanvaardt, meer bepaald de eigen class 'this'<sup>4</sup>. Op die manier maken we een kopie van de eigen Datum class.

In de class Werknemer wijzigen we nu de method SetDatumIn. We maken eerst een clone van d via de code `d.clone()` en casten die vervolgens naar een Datum object. Vervolgens kunnen we datumIn zonder probleem naar de clone-datum laten verwijzen<sup>5</sup>.



De code:

```
public class Datum implements Cloneable { 1
    private int dag, maand, jaar;

    public Datum() {
        setDatum(1,1,1584);
    }
    public Datum( Datum d) { 3
        setDatum(d.dag, d.maand, d.jaar);
    }
    @Override
    public Object clone() { 2
        return new Datum( this )4;
    }
    ...
}

public class Werknemer {

    ...

    public void SetDatumIn( Datum d) {
        datumIn = (Datum)d.clone()5;
    }
}
```

Wat is nu het voordeel van de clone()-method t.o.v. de oplossing met de copy-constructor van pagina 63 ? Geen ! De clone()-method zit nu eenmaal ingebakken in het moederobject Object. We leggen de werkwijze ervan dan ook enkel uit om volledig te zijn.

## 8.5 Oefeningen



Maak oefeningen 20 t.e.m. 22 uit de oefenmap.

## Hoofdstuk 9 Packages

---

### 9.1 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.

### 9.2 Naamgeving conventie voor packages

#### 9.2.1 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*.

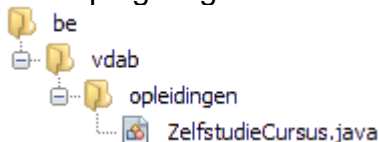
Je schrijft de naam van een package volledig in kleine letters.

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

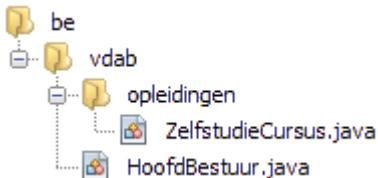
### 9.3 Voorwaarden voor classes in packages

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;`

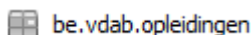
Je IDE (zoals NetBeans) helpen je om aan deze voorwaarden te voldoen.

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.

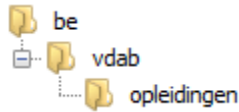
#### 9.3.1 Een package maken in NetBeans

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



### 9.3.2 Een nieuwe class toevoegen aan een bestaande package

Je selecteert het tabblad Projects.

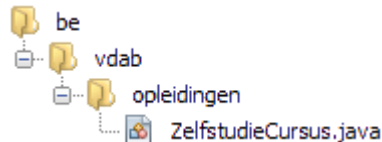
Je klikt met de rechtermuisknop op de package `be.vdab.opleidingen`.

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



### 9.3.3 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.

### 9.3.4 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`;

## 9.4 Verwijzen naar interfaces en classes uit een package

### 9.4.1 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 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 in 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.  
 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'.

#### 9.4.2 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.

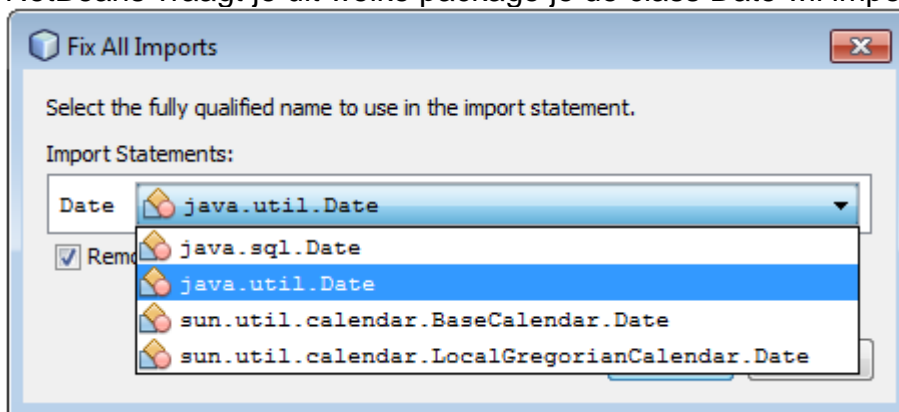
De tweede manier is productiever dan de eerste:

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.

Opmerking:

Als je een class gebruikt uit de package `java.lang`, zoals de class `String`, moet je geen import statement schrijven.

## 9.5 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 maakt 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");
    }
}
```

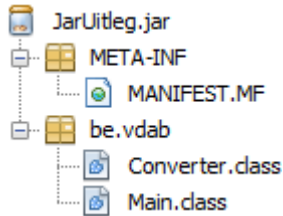
Vooraleer je een Jar bestand maakt van dit project, stel je best eerst de main class van het project in. Dit doe je als volgt: klik met de rechtermuisknop op het project en kies `Properties`. Kies `Run` bij `Categories`. Selecteer via de `Browse` knop bij `Main Class` het juiste uitvoerbare programma (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.

## 9.6 Class path

### 9.6.1 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 huidig 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 de class path uitbreiden met extra directories en JAR bestanden. Java doorzoekt deze daarna ook bij het compileren en uitvoeren van een applicatie.

### 9.6.2 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 in het nieuw project met de rechtermuisknop op het onderdeel Libraries.
- Je kiest Add Jar/Folder...
- Je zoekt het bestand JarUitleg.jar van het vorig bestand en je kiest Open.

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 ConversieProg {
    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");
        }
    }
}
```

## 9.7 Oefeningen



Maak oefening 23 uit de oefenmap.

## Hoofdstuk 10 Exception Handling

---

In elk programma, hoe goed het ook geschreven is, kunnen zich fouten of onverwachte omstandigheden voordoen. We hebben het hier niet over syntaxfouten maar over uitzonderlijke situaties, **exceptions**, bijv. het ontbreken van een bestand dat moet ingelezen worden.

Vroeger werden overal waar er zich een uitzonderlijke omstandigheid kon voordoen de nodige controles in de code toegevoegd. Dit maakte de code bijzonder slecht leesbaar. De fout-veroorzakende en fout-afhandelende code stonden kriskras door elkaar. Bij exception handling wordt er gewoon aangegeven dat er zich in een stuk code mogelijks een bepaalde exception kan voordoen. In het jargon spreekt men dan eerder van een exception *thrown*. Elders in de code wordt deze exception netjes opgevangen in een ... jawel, *catch-block*.

Exceptions zijn, hoe kan het ook anders in een objectgeoriënteerde taal, objects, instances van classes die erven van een class *Throwable*. Throwable heeft twee subclasses: *Error* en *Exception*.

Errors zijn meestal fouten in de Java Virtual Machine die je niet opvangt. Ze zijn meestal ook vrij fataal. Exceptions vang je wel op. Van de class *Exception* worden o.a. *RuntimeExceptions* zoals bijvoorbeeld een *IndexOutOfBoundsException* of een *NullPointerException* afgeleid. In de *java.io* package worden *IOExceptions* gedefinieerd zoals *EOFExceptions*, *FileNotFoundExceptions*,...

### 10.1 Exceptions afhandelen

Beschouwen we de volgende code. In een hoofdprogramma is er een integerlijst gedefinieerd met 5 integers. Via de argumentenlijst vertellen we aan het programma het hoeveelste getal van deze integerlijst we willen zien. Daarna delen we het programma-argument door alle elementen van de array. Zoals je ziet is één van de integers in de array nul.

```
public class ExcepTryout {  
    public ExcepTryout() {}  
  
    public static void main(String[] args) {  
  
        int[] lijst = { 2, 1, 3, 0, 5};  
        int hoeveelste = Integer.parseInt(args[0]);  
  
        System.out.println(lijst[hoeveelste]);  
  
        for (int i=0;i<lijst.length;i++)  
            System.out.println(hoeveelste/lijst[i]);  
  
    }  
}
```



Eerst een woordje uitleg over de argumenten van het programma. Zoals je ziet heeft de `main()`-method een parameter `args`. Dit is een array van strings.

Als we dus een getal willen meegeven als argument aan ons programma, zal in ons programma dit getal (oorspronkelijk type `String`) moeten omgezet worden naar een `integer`. Dit kan niet via type casting. De volgende code geeft een foutmelding:

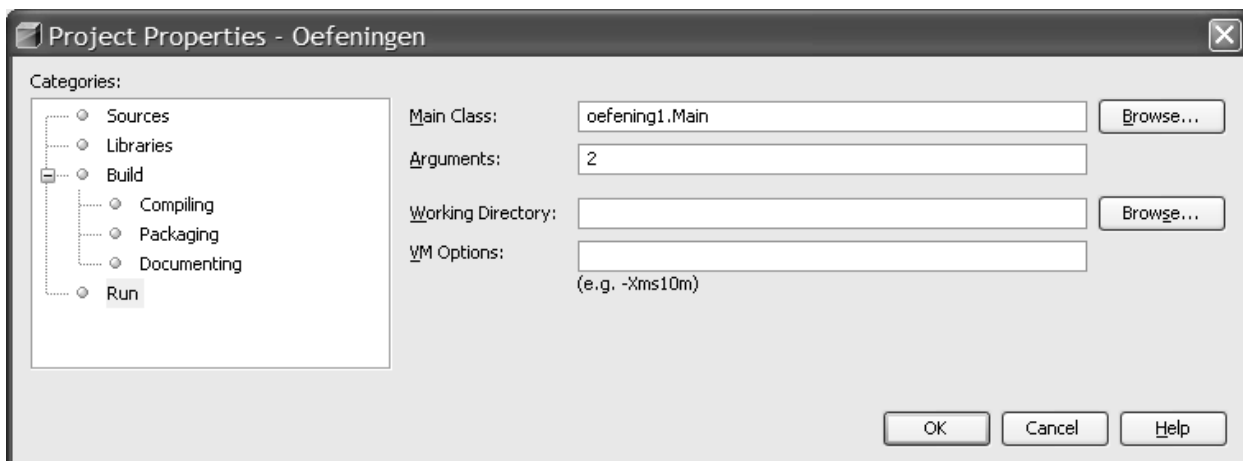
```
int hoeveelste = (int)args[0];
```

```
Inconvertible types  
found : Java.lang.String  
required : int
```

In Java bestaat er zoiets als *wrapper classes*. Elk primitive data type heeft een bijhorende class. Voor het type `int` is dit de class `Integer`. Voor een `float` bijvoorbeeld zal dit de class `Float` zijn.

Deze wrapper classes hebben een aantal data members en methods. Een voorbeeld van een property van de wrapper class `Integer` is bijvoorbeeld `MAX_VALUE`, de grootst mogelijke integer. Een veelgebruikte method van `Integer` is de method `parseInt(String)`. Deze method zet een `String` wél om naar een `int`. De andere wrapper classes hebben allen een gelijkaardige method: `parseFloat()`, `parseLong()`,... In onze code gebruiken we de wrapperclass `Integer` om het argument om te zetten naar een `int` via de method `parseInt()`.

Maar hoe proberen we deze code nu uit in NetBeans met een getal als argument? Helemaal links op het tabblad `Projects` klik je met de rechtermuistoets op de naam van het project – hieronder was dit oefeningen – en je kiest `Properties`.



Aan de linkerkant van het venster kies je onder `Categories` `Run`. Aan de rechterkant kan je dan naast `Arguments` één of meerdere argumenten intikken.

Je klikt dan op `OK` en je kan het programma uitproberen.

Afhankelijk van de waarde die je ingevuld hebt in het vakje `Arguments`, kan je nu een aantal `Exceptions` zien optreden. Laten we beginnen met een letter “b” als argument. Dit veroorzaakt een `NumberFormatException`. Is er geen argument of als argument een getal dat geen geheel getal is tussen 0 en 4, dan krijg je een `ArrayIndexOutOfBounds`.

*BoundsException*. Slaag je er uiteindelijk toch in een geldig argument mee te geven, bijvoorbeeld 3, dan krijg je een *ArithmeticException* wegens een deling door nul.

Laten we beginnen met de *NumberFormatException* op te vangen.

```
public class ExcepTryout {  
    public ExcepTryout() {}  
    public static void main(String[] args) {  
        int[] lijst = { 2, 1, 3, 0, 5};  
        try {  
            int hoeveelste = Integer.parseInt(args[0]);  
            System.out.println(lijst[hoeveelste]);  
            for (int i=0;i<lijst.length;i++)  
                System.out.println(hoeveelste/lijst[i]);  
        }  
        catch (NumberFormatException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

We passen de code aan door alle instructies, behalve de declaratie en initialisatie van de getallenlijst, in een block te plaatsen voorafgegaan door het keyword **try**. Daarna volgt het keyword **catch**, tussen ronde haken het type exception dat we opvangen en een naam voor de opgevangen exception. Tenslotte plaatsen we de code die de fout afhandelt in een catch-block. In dit geval is dit het tonen van het resultaat van de getMessage()-method van de exception op het scherm.

Je zou dit als volgt kunnen interpreteren: “Voer de code in het try-block uit en als er zich een *NumberFormatException* voordoet, voer dan de code in het catch-block uit die de foutboodschap op het scherm toont.

Als je de code nu uitprobeert met als argument een letter, dan crasht het programma niet meer, maar wordt er een boodschap op het scherm getoond. De resterende code van het try-block, dus de code na de argumentconversie, wordt niet meer uitgevoerd.

De volgende exception die we opvangen is de *ArrayIndexOutOfBoundsException*. We plaatsen de coderegel waar een x<sup>e</sup> element uit de lijst wordt opgehaald in een try-block. Doet er zich een exception voor, dan tonen we een passende tekst op het scherm.



```
public class ExcepTryout {  
    public ExcepTryout() {}  
  
    public static void main(String[] args) {  
        int[] lijst = { 2, 1, 3, 0, 5};  
  
        try {  
            int hoeveelste = Integer.parseInt(args[0]);  
  
            try {  
                System.out.println(lijst[hoeveelste]);  
            }  
            catch (ArrayIndexOutOfBoundsException e) {  
                System.out.println("Foutieve arrayindex: "  
                    + hoeveelste );  
            }  
  
            for (int i=0;i<lijst.length;i++)  
                System.out.println(hoeveelste/lijst[i]);  
        }  
        catch (NumberFormatException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Vooraleer het programma uit te voeren wijzig je in de getallenlijst het getal 0 in bijvoorbeeld 9. Voeren we deze keer het programma uit met als argument bijvoorbeeld het getal 8, dan crasht het programma niet, maar krijgen we netjes een foutboodschap en gaat de uitvoering van het programma gewoon door. De for-lus wordt uitgevoerd omdat ze niet in het betreffende try-block zit.

Verander in de getallenlijst de 9 terug naar een 0. We gaan nu de deling door nul opvangen. We plaatsen daartoe de coderegel met de deling in een try-block. In een bijhorend catch-block geven we op het scherm weer dat er zich een deling door nul heeft voorgedaan. Ook nu gaat de uitvoering van het programma gewoon door na het optreden van de exception.

```
public class ExcepTryout {  
    public ExcepTryout() {}  
  
    public static void main(String[] args) {  
        int[] lijst = { 2, 1, 3, 0, 5};  
  
        try {  
            int hoeveelste = Integer.parseInt(args[0]);  
  
            try {  
                System.out.println(lijst[hoeveelste]);  
            }  
        }  
    }  
}
```

```
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Foutieve arrayindex: " +
                hoeveelste );
        }

        for (int i=0;i<lijst.length;i++)
            try {
                System.out.println(hoeveelste/lijst[i]);
            }
            catch (ArithmeticException e) {
                System.out.println("Deling door nul");
            }
        }
        catch (NumberFormatException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Met deze kennis beschouwen we even de volgende code:

```
public class ExcepTryout {

    public ExcepTryout() {}

    public static void main(String[] args) {

        int[] lijst = { 2, 1, 3, 0, 5};

        try {
            System.out.println(lijst[Integer.parseInt(args[0])]);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Foutieve arrayindex");
        }

        catch (NumberFormatException e) {
            System.out.println(e.getMessage());
        }

    }
}
```

In deze code staan twee risico-operaties op dezelfde lijn. Om toch een gepaste foutboodschap weer te geven, kunnen we twee catch-blocks aan één try-block koppelen. Op deze manier worden twee verschillende exceptions opgevangen.

Een andere, minder elegante, oplossing is één catch-block te voorzien die een *RuntimeException* opvangt. Zowel *ArrayIndexOutOfBoundsException* als *NumberFormatException* hebben een moederclass *RuntimeException*.



```
public class ExcepTryout {  
    public ExcepTryout() {}  
  
    public static void main(String[] args) {  
        int[] lijst = { 2, 1, 3, 0, 5};  
  
        try {  
            System.out.println(lijst[Integer.parseInt(args[0])]);  
        }  
        catch (RuntimeException e) {  
            System.out.println(  
                "Foutieve arrayindex of ongeldig argument");  
        }  
    }  
}
```

In het volgende voorbeeld wordt *eenGetal* gedeeld door alle getallen van de array *lijst*. De deling door 0 veroorzaakt een *ArithmeticException* die opgevangen wordt in het catch-block. De vermenigvuldiging met 0 wordt wel uitgevoerd, omdat deze zich in het **finally-block** bevindt. De code van het finally-block wordt immers altijd uitgevoerd. Dit finally-block wordt vaak gebruikt voor het afsluiten van resources, bijv. sluiten van de databaseconnectie, sluiten van bestanden....

```
public class ExcepTryout {  
    public ExcepTryout() {}  
  
    public static void main(String[] args) {  
        int[] lijst = { 2, 1, 3, 0, 5};  
  
        int eenGetal = 3;  
  
        for (int i=0;i<lijst.length;i++) {  
            try {  
                System.out.println(eenGetal + " / " + lijst[i] +  
                    " = " + eenGetal/lijst[i]);  
            }  
            catch (ArithmeticException e) {  
                System.out.println("Deling door nul");  
            }  
            finally {  
                System.out.println(lijst[i] + " x " + eenGetal +  
                    " = " + eenGetal * lijst[i]);  
            }  
        }  
    }  
}
```



## 10.2 Eigen exceptions maken

Naast de voorgedefinieerde exceptions die we in de vorige paragraaf leerden kennen, kan je ook zelf exceptions maken. Je kan dus aangeven dat een bepaalde method een gebeurtenis kan veroorzaken. Als je bijvoorbeeld reservaties verwerkt voor een voorstelling, dan kan het inbrengen van een negatief of te groot aantal tickets zorgen voor een Exception.

Deze exceptions leid je dan af van `RuntimeException` of van `Exception`. Het verschil hier is dat een exception, afgeleid van `RuntimeException` **kan** opgevangen worden, een exception afgeleid van de *Exception class*, **moet** opgevangen worden.

Laten we een voorbeeld nemen. In een hoofdprogramma hebben we een class `Gemeente` nodig. Deze gemeente heeft een naam en een postcode. Voor de postcode kunnen we eisen stellen. De code moet groter zijn dan 999 en kleiner dan 10000. We houden het dus relatief eenvoudig.

Hoe gaan we nu tewerk? We definiëren een eigen exception class en in de definitie van iedere method die deze exception kan veroorzaken geven we aan dat de method deze eigen exception class kan throwen. Dit gebeurt met een keyword '*throws*'. Het oproepen van een method die de exception class kan throwen, plaatsen we in een try-block. We vangen de exception op in een catch-block.

Het voorbeeld. We maken eerst een eigen exception class, afgeleid van de class `Exception`.

```
public class PostNrException extends Exception {
    public PostNrException() {}
    public PostNrException(String omschrijving) {
        super(omschrijving);
    }
}
```

We definiëren een class `PostNrException` die afgeleid is van de class `Exception`. Zoals je al eerder in een vorige paragraaf zag, geven we meestal bij het opvangen van een exception een stukje tekst weer dat verduidelijkt welke soort fout er zich heeft voorgedaan. Dit is meestal de message-property van de exception. We voorzien dus ook een constructor die een stringparameter aanvaardt. Deze parameter geven we door aan de moederclass `Exception`.

Laten we nu de class `Gemeente` definiëren:

```
public class Gemeente {

    private String naam;
    private int postNr;

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

    public void setPostNr(int postNr) throws PostNrException {
        if (postNr<1000 || postNr > 9999)
            throw new PostNrException("Verkeerd Postnr");
        this.postNr=postNr;
    }
}
```



```

public Gemeente(String naam, int postNr) throws PostNrException {
    setNaam(naam);
    setPostNr(postNr);
}

@Override
public String toString() {
    return postNr + " " + naam;
}
}

```

Onze class `Gemeente` heeft twee property's: `naam` en `postNr`. Voor beide voorzien we een method die de property's invult. Zoals gezegd kan er zich een (eigen) exception voordoen als we het postnummer proberen in te vullen. Dit geven we aan met het keyword '*throws*' en dan de *naam van de exception*. In de method testen we of het postnummer geldig is. Is dit niet het geval dan throwen we de eigen exception. Verder voorzien we nog een constructor met twee parameters. Aangezien deze constructor de method `setPostNr` aanroept kan ook de constructor een exception veroorzaken. Dus ook hier voegen we "*throws PostNrException*" toe. Tenslotte definiëren we een `toString()`-functie.

In een eenvoudig hoofdprogramma maken we een instance aan van de class `Gemeente`.

```

public class ProgGemeente {
    public static void main(String[] args) {
        try {
            Gemeente eenGemeente=new Gemeente("Ertvelde",123);
        }
        catch (PostNrException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

De risicocode, het oproepen van de constructor van de class `Gemeente`, zetten we in een try-block. We vangen een `PostNrException` op en geven een boodschap weer op het scherm.

Voor de gebruiker is het nu duidelijk dat er zich een fout heeft voorgedaan, namelijk een fout i.v.m. het postnummer. Hiermee weet de gebruiker echter nog niet over welk postnummer het ging. Aangezien het `Gemeente`-object in het catch-block niet meer bestaat, kunnen we best het foutieve postnummer eveneens opnemen in de `PostNrException` class. We passen de `PostNrException` class aan:

```

public class PostNrException extends Exception {
    private int verkeerdPostNr;
    public PostNrException() {}
    public PostNrException(String omschrijving) {
        super(omschrijving); }

    public PostNrException(String omschrijving, int verkeerdPostNr) {
        super(omschrijving);
        this.verkeerdPostNr=verkeerdPostNr; }

    public int getVerkeerdPostNr(){

```

```
        return verkeerdPostNr; }  
    }
```

We voegen in de exceptionclass een int-property `verkeerdPostNr` bij. Verder komt er een extra constructor bij met deze keer twee parameters: een foutomschrijving en het verkeerde postnummer. Tenslotte maken we een publieke method die de property `verkeerdPostNr` retourneert.

In de class `Gemeente` werpen we niet alleen een foutomschrijving maar ook het foutveroorzakende postnummer.

```
public class Gemeente {  
    private String naam;  
    private int postNr;  
    public void setNaam(String naam) {  
        this.naam=naam;  
    }  
    public void setPostNr(int postNr) throws PostNrException {  
        if (postNr<1000 || postNr > 9999)  
            throw new PostNrException("Verkeerd Postnr", postNr);  
        this.postNr=postNr;  
    }  
    public Gemeente(String naam, int postNr) throws PostNrException {  
        setNaam(naam);  
        setPostNr(postNr);  
    }  
    public String toString() {  
        return postNr + " " + naam;  
    }  
}
```

Dit stelt ons nu in staat om in het hoofdprogramma in het catch-block niet alleen een boodschap weer te geven maar ook het postnummer dat de fout veroorzaakte:

```
public class ProgGemeente {  
    public static void main(String[] args) {  
        try {  
            Gemeente eenGemeente=new Gemeente("Ertvelde",123);  
        }  
        catch (PostNrException e) {  
            System.out.println(e.getMessage()+" "+e.getVerkeerdPostNr());  
        }  
    }  
}
```

## 10.3 Oefeningen



Maak oefening 24 en 25 uit de oefenmap.



## Hoofdstuk 11    BigDecimal

---

### 11.1 Probleemstelling

Wie in Java een float of een double gebruikt, komt af en toe voor een verrassing te staan. In het onderstaande voorbeeld tellen we 1000 keer 0,01 op. Uiteraard verwacht je dan als som 10 maar helaas is niets minder waar.

```
double totaal=0.0D;
for (int i=0;i<1000;i++) {
    totaal += 0.01D;
}
System.out.println(totaal);

float totaalf=0.0F;
for (int i=0;i<1000;i++) {
    totaalf += 0.01F;
}
System.out.println(totaalf);
```

Dit bovenstaande programma levert volgende output :

9.999999999999831

10.0001335

Het probleem is dat floating-point getallen weliswaar snel tot een oplossing komen maar dat deze oplossing niet altijd de volledige precisie heeft. Zelfs niet bij een eenvoudig voorbeeld zoals hierboven.

De class `BigDecimal` biedt een oplossing voor dit probleem: het kan decimale getallen exact weergeven doordat het 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 maar bijvoorbeeld aan een al dan niet bewegende grafische toepassing. Voor business applications waar elke cent belangrijk is gebruik je beter `BigDecimal`.

### 11.2 De `BigDecimal` class

#### 11.2.1 Constructors

Om een `BigDecimal` aan te maken dat een geheel getal voorstelt gebruik je de constructor `BigDecimal(int)` of `BigDecimal(long)`.

Bijvoorbeeld:

```
BigDecimal geluksGetal = new BigDecimal(7);
```

Je gebruikt de constructor `BigDecimal(String)` om een `BigDecimal` te maken die een getal voorstelt met decimalen.

Bijvoorbeeld:

```
BigDecimal increment = new BigDecimal("0.01");
```

Er bestaan ook een constructor `BigDecimal(double)` maar dan loop je opnieuw gevaar afrondingsfouten te krijgen.

Vergeet niet `java.math.BigDecimal` te importeren wanneer je gebruik maakt van `BigDecimal`s.

Voor een aantal waarden hoef je geen constructor te gebruiken: `BigDecimal.ZERO`, `BigDecimal.ONE` en `BigDecimal.TEN` leveren de waarden 0, 1 en 10 op.

### 11.2.2 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 te vergelijken. Ook de methods `abs()`, `negate()` en `toString()` spreken voor zich.

Een woordje uitleg wel bij 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 = new BigDecimal("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. Een getal als 2.362 wordt in beide gevallen naar 2.36 afgerond.

Naast deze twee afrondingsmethodes zijn er nog verschillende andere die we hier niet allemaal gaan behandelen.

Om de enumeratie `RoundingMode` te kunnen gebruiken moet je eerst een import doen van `java.math.RoundingMode`.

## 11.3 De oplossing mét BigDecimal

Als we nu ons voorbeeld aanpassen door gebruik te maken van de `BigDecimal` class dan krijgen we wel het goede resultaat.

```
BigDecimal totaalBD = BigDecimal.ZERO;
BigDecimal incremBD = new BigDecimal("0.01");

for (int i=0; i<1000; i++) {
    totaalBD = totaalBD.add(incremBD);
}

System.out.println(totaalBD);
```

Dit levert volgende output :

10.00



## Hoofdstuk 12 Collections

---

### 12.1 Inleiding

#### 12.1.1 Wat is een collection?

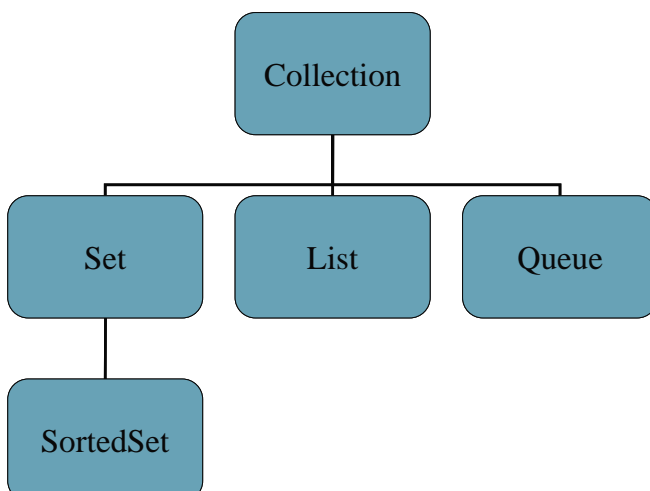
In het hoofdstuk “Arrays” is het concept van arrays reeds besproken. In een array kunnen we een aantal objecten van dezelfde class (van hetzelfde type) bewaren. Een array is een klassieke en krachtige datastructuur die heel veel wordt gebruikt, maar een array heeft een aantal belangrijke nadelen:

- 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 groepeer**t. 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.
- Nagaan of een object aanwezig is in de collection.
- Itereren over alle objecten in de collection.

Een overzicht van de **Collection interface**:



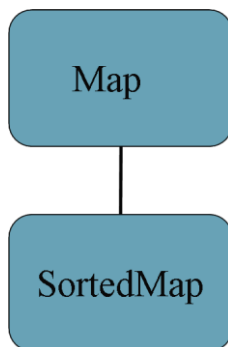
In bovenstaande afbeelding kan je zien dat van de Collection interface nog andere interfaces zijn afgeleid. Een korte toelichting bij deze afgeleide interfaces:

<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 uitgebreid terug op deze interfaces. Ze komen allen aan bod, behalve Queue.



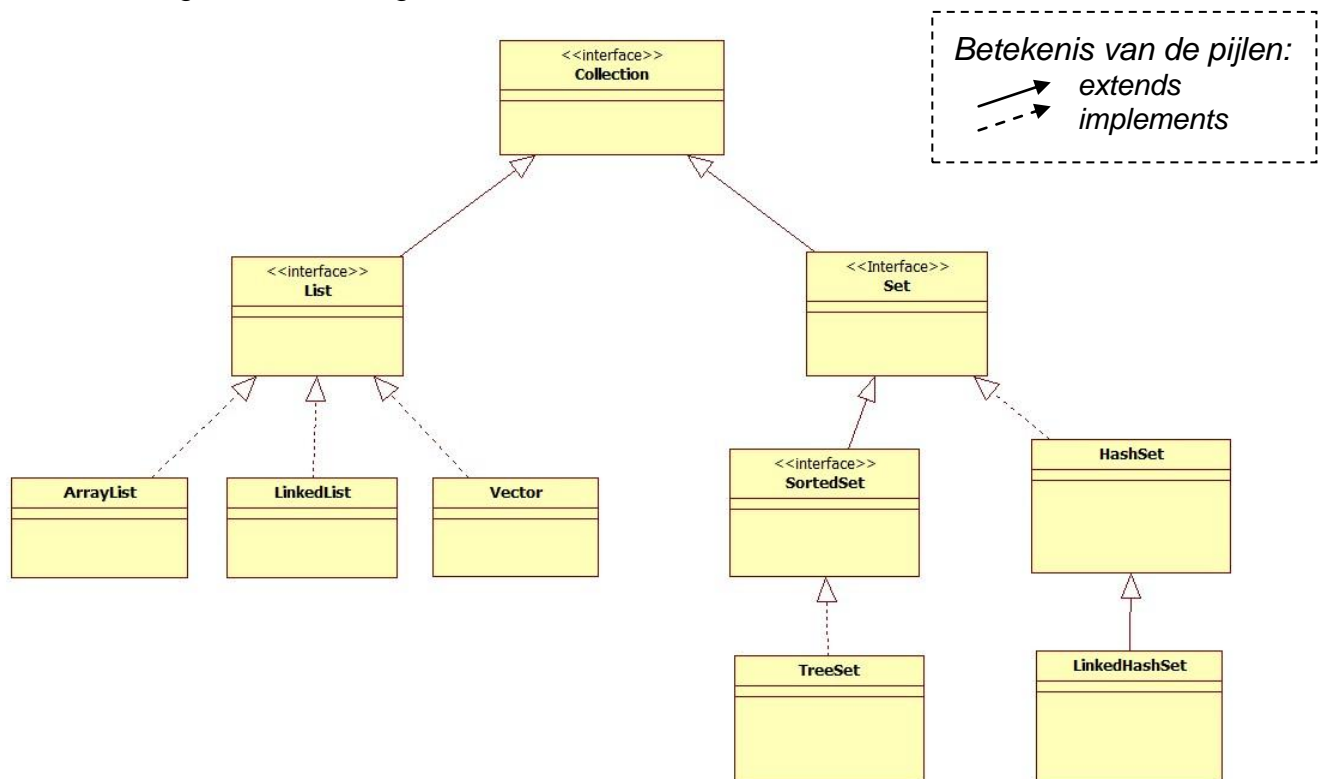
### 12.1.2 Concepten van collections

We gaan kort enkele principes van collections opsommen. Later zullen we deze principes bij de uitleg van de interfaces en de voorbeelden toelichten.

- In een collection kunnen enkel objecten bewaard worden. Indien we dus een verzameling willen van primitieve types (bijv. integers), moeten we deze types verpakken in **wrapper classes** (zie p. 106).
- In een collection kunnen we verschillende soorten objecten bewaren omdat de argumenten van de collectionmethods objecten aanvaarden. Je kan dus bijv. objecten van de class Persoon, objecten van de class Bedrijf en objecten van de class Datum samen bewaren in één collection. Het nut hiervan laten we even buiten beschouwing.
- Wanneer we objecten uit een collection halen, zijn dit steeds objecten van het type Object (de moeder van alle classes). Dus moeten we deze objecten typecasten naar hun eigenlijke type om terug over de volledige functionaliteit van het object te kunnen beschikken. Een oplossing voor het probleem van typecasten is **Generics**. Dit wordt behandeld in een apart hoofdstuk.
- Elke collection heeft haar eigen opslagstructuur. De organisatie van deze opslagstructuur is intern aan de collectie.
- Een collection moet doorlopen kunnen worden. Dit kan via een speciale class, die we een **iterator** noemen. Iterators worden binnen in de collection gedefinieerd, omdat zij moeten vertrouwd zijn met de interne structuur van de collection.

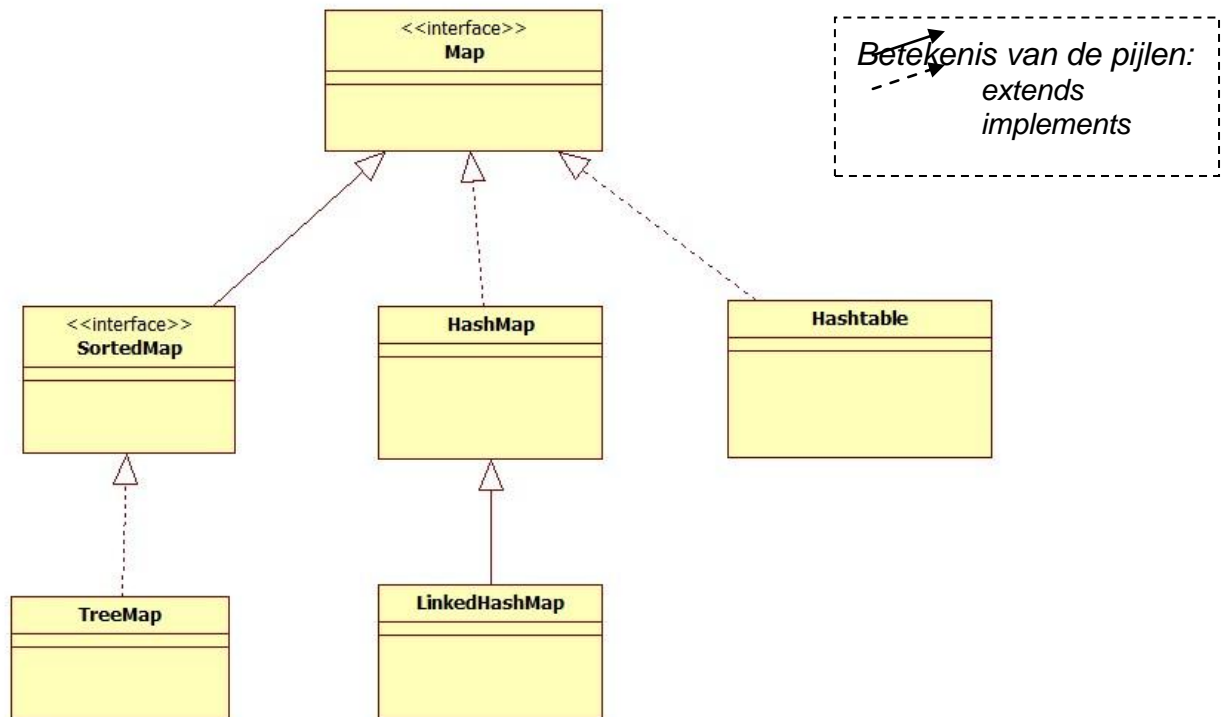
## 12.2 De collection interface

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, 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. In paragraaf 12.5 Generics (zie p. 157) komen we daar op terug.

De collection wordt doorlopen met een **iterator**. We vragen deze iterator 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> .

Een voorbeeld om alles te verduidelijken:

```

Collection coll = new HashSet();                                     (1)
coll.add( new Persoon("Jan", new BigDecimal("2100.85")) );
coll.add( new Persoon("Jos", new BigDecimal("1890.10")) );
coll.add( new Persoon("Jef", new BigDecimal("2400.55")) );         (2)

BigDecimal somLoon = BigDecimal.ZERO;

Iterator i = coll.iterator();                                       (3)
while( i.hasNext() )                                               (4)
{
    Object o = i.next();                                           (5)
    if ( o instanceof Persoon ) {                                   (6)
        Persoon p = (Persoon) o;
        somLoon = somLoon.add(p.getLoon());
    }
}

```

- (1) Er wordt een collectie gemaakt van het type `HashSet`. Dit is een implementatie van de `Set` Interface (zie het class-diagram; wordt verder in de cursus nog behandeld).
- (2) Er worden 3 objecten van de class `Persoon` toegevoegd aan deze collectie. We gebruiken hiervoor de constructor met argumenten *naam* en *loon*.
- (3) De iterator van de collection wordt gecreëerd om over alle elementen van de collection te kunnen itereren. De method `iterator()` van het object `coll` (een `HashSet` collection) wordt uitgevoerd. Zoals reeds gezegd worden iterators binnen in de collection gedefinieerd, omdat zij moeten vertrouwd zijn met de interne structuur van de collection.  
Deze method levert een object op van de class `Iterator` en bijgevolg hebben we methods (o.a. `next()`, `hasnext()`, en `remove()`) ter beschikking voor het doorlopen van de collection en voor het ophalen van objecten uit de collection.
- (4) Zolang er nog een volgend element in de collection aanwezig is, wordt de lus opnieuw doorlopen. Op deze manier wordt de volledige collection doorlopen.
- (5) De method `next()` haalt dit volgend element uit de collection (dit element is altijd een object, dus van het type `Object`).

- (6) Controleer of dit element van de class Persoon is om een typecasting te kunnen doen, om vervolgens het loon te kunnen opvragen en te sommeren.

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	Return-waarde	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.

Verder verwijzen we naar de API-documentatie van de interface Collection voor een volledig overzicht van alle beschikbare methods.

In de volgende paragrafen worden nu de verschillende interfaces besproken.

## 12.3 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 en elk element heeft een integer-positie in de verzameling.

Onder List verstaan we een collectie met een ordening: 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.

Verder verwijzen we naar de API-documentatie van de interface List voor een volledig overzicht van alle beschikbare methods.

### 12.3.1 ListIterator

De List Interface heeft twee soorten iterators om de collection te doorlopen:

- Iterator: de standaard iterator om de collection van voor naar achteren te doorlopen.
- ListIterator: een iterator om de collection ook in omgekeerde volgorde te doorlopen of om de collection vanaf een bepaalde positie te doorlopen.

De List Interface bevat twee methods om een ListIterator op te vragen:

Method	Return-waarde	Toelichting
<code>listIterator()</code>	<code>ListIterator&lt;E&gt;</code>	Geeft een list iterator van de elementen in de lijst (in de juiste volgorde).
<code>listIterator(int index)</code>	<code>ListIterator&lt;E&gt;</code>	Geeft een list iterator van de elementen in de lijst, beginnend bij de positie opgegeven door index.

De ListIterator is een interface, afgeleid van Iterator. Deze kan een list in beide richtingen doorlopen. Tevens kan de index-positie van het vorige/volgende element opgevraagd worden.

Verder kan via de ListIterator eveneens een element toegevoegd worden aan de collection of vervangen worden door een ander element.

We sommen de extra methods even op:

Method	Return-waarde	Toelichting
<code>add(E o)</code>	<code>void</code>	Voegt element E tussen aan de list.
<code>hasPrevious()</code>	<code>boolean</code>	Is er nog een vorig element?
<code>nextIndex()</code>	<code>int</code>	Geeft de index terug van het volgend element.
<code>previous()</code>	<code>E</code>	Geeft het vorig element terug.
<code>previousIndex()</code>	<code>int</code>	Geeft de index terug van het vorig element.
<code>set(E o)</code>	<code>void</code>	Vervangt het element dat het laatst is teruggegeven door <code>next()</code> of <code>previous()</code> door het opgegeven element.

Een ListIterator heeft geen current element. Zijn cursorpositie bevindt zich steeds tussen het element dat zal worden gereturd door een `previous()` en het element dat zal worden gereturd door een `next()`.



Een iterator van een List met lengte  $n$  heeft  $n+1$  mogelijke cursorposities. Zie hieronder aangegeven door  $\wedge$ :

```

cursor positions:  ^      ^      ^      ^      ...      ^
                   Element(0) Element(1) Element(2) ... Element(n-1)
  
```

De twee belangrijkste implementaties van de List zijn de ArrayList en de LinkedList. Deze worden in de volgende twee paragrafen behandeld.

### 12.3.2 ArrayList

Een ArrayList is eigenlijk een resizable-array implementatie van de List-interface.

Een ArrayList is een object dat 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 het null-element bewaren en tevens kunnen dubbels bewaard worden.

We sommen enkele extra methods van de ArrayList op:

Method	Return-waarde	Toelichting
<code>ensureCapacity (int minCapacity)</code>	<code>void</code>	Vergroot, indien nodig, de capaciteit van de ArrayList om er zeker van te zijn dat ze het aantal elementen, gespecificeerd door <code>minCapacity</code> , kan bewaren.
<code>removeRange (int fromIndex, int toIndex)</code>	<code>void</code>	Verwijdert uit deze ArrayList alle elementen waarvan de index ligt tussen <code>fromIndex</code> en <code>toIndex</code> (incl. <code>fromIndex</code> en excl. <code>toIndex</code> ).

### 12.3.2.1 *ArrayList: een voorbeeld*

We bouwen een voorbeeld van de `ArrayList` in stappen op:

```
System.out.println("List op basis van ARRAYLIST");
List al = new ArrayList();                                (1)
vul(al);                                                  (2)
toon(al);                                                  (3)

private static void vul(List lijst)
{
    lijst.add("fiets");                                    (4)
    lijst.add(null); //null-element toegestaan
    lijst.add("even");
    lijst.add("dak");
    lijst.add("citroen");
    lijst.add("citroen"); //dubbels toegestaan
    lijst.add("boom");
    lijst.add("aap");
}

private static void toon(List lijst)
{
    System.out.println("*** Met een for-statement ***");
    for (int i=0; i<lijst.size(); i++)                    (5)
    {
        String woord = (String) lijst.get(i);            (6)
        System.out.println(woord);
    }
}
```

- (1) Er wordt een collectie gemaakt van het type `ArrayList`.
- (2) De method *vul* wordt opgeroepen en de arraylist wordt als parameter meegegeven.
- (3) Nadat de arraylist gevuld is, wordt de method *toon* opgeroepen en ook hier wordt de arraylist als parameter meegegeven.
- (4) Er worden verschillende string-objecten toegevoegd aan de arraylist. Ook het null-element en een dubbel wordt toegevoegd.
- (5) In een lus wordt de arraylist overlopen. De method `size()` vraagt het aantal elementen op van de arraylist zodat je de lus kan laten lopen vanaf het nulde-element tot en met het laatste element.  
Het eerste element wordt ook hier aangesproken vanaf 0, net zoals dat bij arrays het geval is.
- (6) Met de method `get(i)` haal je het zoveelste (i-de) element uit de arraylist. Een element dat je uit een collection haalt, is van het type `Object`. Daarom is hier typecasting nodig om het element om te zetten naar een string-object en te bewaren in een string-variabele *woord*. Vervolgens wordt deze string getoond.



De uitvoer van deze code is:

```
List op basis van ARRAYLIST
*** Met een for-statement ***
fiets
null
even
dak
citroen
citroen
boom
aap
```

Merk op dat de volgorde behouden blijft: in de volgorde zoals de elementen worden toegevoegd aan de ArrayList, worden ze er ook weer uitgehaald.

Je kan de ArrayList nog op een **tweede manier doorlopen**, nl. met een for-each statement, ook wel een collection-based for loop of enhanced for loop genoemd. Dit kan vanaf java 5. Voeg deze tweede manier als volgt toe aan de method `toon()`:

```
private static void toon(List lijst)
{
    ...
    System.out.println("*** Met een for-each ***");
    for (Object obj : lijst )                (7)
    {   String woord = (String) obj;         (8)
        System.out.println(woord);
    }
}
```

- (7) De collectie of ArrayList *lijst* wordt helemaal doorlopen. Je begint vooraan en gaat door tot het einde van de lijst.  
Je kan de regel bijna vertalen als volgt: “Voor elk Object *obj* uit de lijst gebeurt er wat tussen de accolades { } staat”.
- (8) Bij iedere cyclus van de loop wordt één element (uit de lijst) ter beschikking gesteld en in een variabele *woord* gestopt. Vermits er uit een collectie steeds objecten gehaald worden van het type Object, is typecasting nodig.

De uitvoer van dit stukje code is:

```
List op basis van ARRAYLIST
*** Met een for-each ***
fiets
null
even
dak
citroen
citroen
boom
aap
```

Uiteraard is de uitvoer dezelfde als bij de vorige manier.



Er is nog een **derde manier** om de collectie te doorlopen, nl. met de iterator.  
Voeg onderstaande derde manier toe aan de method `toon()`:

```
private static void toon(List lijst)
{
    ...
    System.out.println("*** Met de iterator ***");
    for ( Iterator i = lijst.iterator(); i.hasNext(); )           (9)
    {   String woord = (String) i.next() ;                       (10)
        System.out.println(woord) ;                             (11)
    }
}
```

- (9) De iterator van de ArrayList wordt 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.
- (10) Met de method `next()` wordt het eerstvolgend element uit de collectie opgehaald. Je itereert steeds van het begin tot het einde in de collectie.  
Typecasting wordt toegepast om het object om te zetten naar een String (de method `next()` returnt een Object).
- (11) Het element wordt getoond.

De uitvoer van dit stukje code is:

```
List op basis van ARRAYLIST
*** Met de iterator ***
fiets
null
even
dak
citroen
citroen
boom
aap
```

Ook hier is de uitvoer natuurlijk dezelfde als bij de vorige 2 mogelijkheden.



We gaan het voorbeeld uitbreiden en twee nieuwe elementen toevoegen aan de ArrayList:

```
System.out.println("List op basis van ARRAYLIST");
List al = new ArrayList();
vul(al);
toon(al);

System.out.println("Nieuwe elementen");
al.add(3, "test");           (1)
al.add("beer");             (2)
toon(al);

private static void vul(List lijst)
{
    lijst.add("fiets");
    lijst.add(null);         //null-element toegestaan
    lijst.add("even");
    lijst.add("dak");
    lijst.add("citroen");
    lijst.add("citroen");    //dubbels toegestaan
    lijst.add("boom");
    lijst.add("aap");
}
```

- (1) 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.
- (2) In de ArrayList wordt **achteraan** het element “beer” toegevoegd.

Dit programma voert na de toevoeging van de twee nieuwe elementen de lijst als volgt uit :

```
fiets
null
even
test
dak
citroen
citroen
boom
aap
beer
```

We proberen nog enkele methods van de ListIterator uit:

```
System.out.println("List op basis van ARRAYLIST");
List al = new ArrayList();
vul(al);
toon(al);

System.out.println("Nieuwe elementen");
al.add(3, "test");
al.add("beer");
toon(al);

//tweede String "citroen" vervangen door "tweede citroen"
System.out.println("Element vervangen");
try { ListIterator li = al.listIterator(6);           (1)
    li.next();                                     (2)
    li.set("tweede citroen") ;                     (3)
    li.add("kers");                                 (4)
}
catch (IndexOutOfBoundsException e) {
    System.out.println("element niet gevonden (index out of bound): "
        + e.getMessage());
}
catch (NoSuchElementException e) {
    System.out.println("geen element beschikbaar: " + e.getMessage());
}

System.out.print("ArrayList afgedrukt vanaf element 3");
toonVanafIndex(al, 2);                               (5)

System.out.print("\nArrayList omgekeerd afgedrukt");
toonVanafIndexOmgekeerd(al, al.size());              (6)

private static void vul(List lijst)
{ ...
}

private static void toon(List lijst)
{ ...
}

private static void toonVanafIndex(List l, int startPos)
{ System.out.println();
  ListIterator li ;

  for ( li = l.listIterator(startPos) ; li.hasNext() ; )   (7)
  {   System.out.println( li.next() );                     (8)
  }
}

private static void toonVanafIndexOmgekeerd(List l, int startPos)
{ System.out.println();
  ListIterator li;

  for ( li = l.listIterator(startPos) ; li.hasPrevious() ; )   (9)
  {   System.out.println(li.previous() );                     (10)
  }
}
```



- (1) De `listIterator` van de `ArrayList` wordt opgevraagd vanaf positie 6. Let op: er wordt geteld vanaf 0.  
Wanneer de gevraagde positie niet beschikbaar is, wordt er een *`IndexOutOfBoundsException`* gethrowed.
- (2) De method `next()` haalt het element op positie 6 uit de collectie. Dit is de tweede string "citroen".  
Wanneer er gepositioneerd is precies na het laatste element, kan er geen element opgevraagd worden en zal `next()` leiden tot een *`NoSuchElementException`*.
- (3) De method `set()` wordt uitgevoerd op het element dat het laatst uit de collectie is gereturt d.m.v. de method `next()` of `previous()`.  
In dit voorbeeld wordt het tweede string-object "citroen" gewijzigd naar een nieuw string-object met waarde "tweede citroen".
- (4) Vervolgens wordt een nieuw object toegevoegd, nl. string-object "kers". In tegenstelling tot de method `add()` van de collection interface die het element achteraan in de collectie toevoegt, zal de method `add()` van de `ListInterface` het string-object "kers" tussenvoegen op de plaats waar hij op dat moment staat. Dus in dit geval na het string-object "tweede citroen".
- (5) De method *`toonVanafIndex`* wordt opgeroepen met 2 argumenten: het eerste argument is de `ArrayList` en het tweede argument is de positie vanaf waar in de collectie de elementen zullen getoond worden. Hier is dat vanaf positie 2, dus vanaf het derde element (er wordt geteld vanaf 0).
- (6) De method *`toonVanafIndexOmgekeerd`* wordt opgeroepen met 2 argumenten: het eerste argument is de `ArrayList` en het tweede argument is de positie vanaf waar in de collectie de elementen zullen getoond worden.  
Hier is dat de grootte van de collection die opgevraagd wordt met de method `size()`.
- (7) De `listIterator` van de `ArrayList` wordt opgevraagd met als index de opgegeven startpositie. Hiermee itereer je over de collection vanaf deze startpositie. 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 `listIterator` een element verder in de collectie verplaatst.
- (8) Met de method `next()` wordt het eerstvolgend element uit de collectie gehaald. Het element wordt getoond. Typecasting naar een `String`-object hoeft niet expliciet te gebeuren: de *`System.out.println`* zal de `toString()` method van het object oproepen en die is overridden in de class `String`.
- (9) De `listIterator` van de `ArrayList` wordt opgevraagd met als index de opgegeven startpositie. Hiermee itereer je over de collection vanaf deze startpositie. Zolang er elementen in de collection aanwezig zijn (`hasPrevious()` levert dan `true` op), wordt de lus doorlopen.  
Het derde argument van de for-lus blijft leeg. Een 'verlaging' van de iterator is

niet nodig vermits in de body van de lus de method `previous()` wordt uitgevoerd en hiermee wordt de listIterator een element terug naar voren in de collectie verplaatst.

Vermits we achteraan de collectie beginnen (bij het laatste element) en steeds met `previous()` een element naar voren opschuiven, zal de collectie in omgekeerde volgorde getoond worden.

- (10) Met de method `previous()` wordt het vorige element uit de collectie gehaald. Het element wordt getoond. Typecasting naar een String-object hoeft niet expliciet te gebeuren: de instructie `System.out.println` zal de `toString()`-method van het object oproepen en die is overridden in de class String.

Na de uitvoer die we ook al in het vorige voorbeeld kregen, krijgen we het volgende te zien :

```
Element vervangen
ArrayList afgedrukt vanaf element 2
even
test
dak
citroen
tweede citroen
kers
boom
aap
beer
```

```
ArrayList omgekeerd afgedrukt
beer
aap
boom
kers
tweede citroen
citroen
dak
test
even
null
fiets
```

Stel vast dat de tweede string “citroen” vervangen is door “tweede citroen”. Daarachter is de string “kers” tussengevoegd.

Vervolgens worden de gegevens in de collectie getoond: een keer vanaf index-positie 2 en een keer allemaal in omgekeerde volgorde.



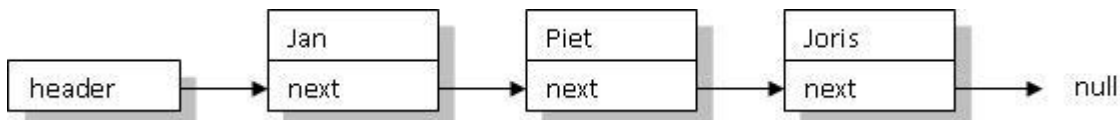
In deze voorbeelden zijn steeds String-objecten gebruikt voor het vullen van een collectie.

Een collectie is een verzameling van objecten, d.w.z. dat er eender welke objecten bewaard kunnen worden in een collectie.

### 12.3.3 LinkedList

Een linked list 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.

#### 12.3.3.1 Voorbeeld van een LinkedList

We hebben reeds een voorbeeld voor de ArrayList in stappen opgebouwd. Ditzelfde voorbeeld werkt ook voor een LinkedList. We passen het aan voor de LinkedList:

```

System.out.println("List op basis van LINKEDLIST");
List ll = new LinkedList();
vul(ll);
toon(ll);

System.out.println("Nieuwe elementen");
ll.add(3, "test");
ll.add("beer");
toon(ll);

//tweede String citroen vervangen door "tweede citroen"
System.out.println("Element vervangen");
ListIterator li = ll.listIterator(6);
li.next();
  
```

(1)

```
li.set( (String) "tweede citroen" ) ;
li.add("kers");

System.out.print("LinkedList afgedrukt vanaf element 2");
toonVanafIndex(11, 2);

System.out.print("\nLinkedList omgekeerd afgedrukt");
toonVanafIndexOmgekeerd(11, 11.size());

private static void vul(List lijst)
{
    lijst.add("fiets");
    lijst.add(null);           //null-element toegestaan
    lijst.add("even");
    lijst.add("dak");
    lijst.add("citroen");
    lijst.add("citroen");     //dubbels toegestaan
    lijst.add("boom");
    lijst.add("aap");
}

private static void toon(List lijst)
{
    System.out.println("*** Met een for-statement ***");
    for (int i=0; i<lijst.size(); i++) {
        String woord = (String) lijst.get(i);
        System.out.println(woord);
    }

    System.out.println("*** Met een for-each ***");
    for (Object obj : lijst ) {
        String woord = (String) obj;
        System.out.println(woord);
    }

    System.out.println("*** Met de iterator ***");
    for ( Iterator i = lijst.iterator(); i.hasNext(); ) {
        String woord = (String) i.next() ;
        System.out.println(woord) ;
    }
}

private static void toonVanafIndex(List l, int startPos)
{
    System.out.println();
    ListIterator li ;

    for ( li = l.listIterator(startPos) ; li.hasNext() ; )
    {
        System.out.println( li.next() );
    }
}

private static void toonVanafIndexOmgekeerd(List l, int startPos)
{
    System.out.println();
    ListIterator li;

    for ( li = l.listIterator(startPos) ; li.hasPrevious() ; )
    {
        System.out.println(li.previous() );
    }
}
```



- (1) Er wordt een collectie gemaakt van het type `LinkedList`. Verder zijn het dezelfde statements als bij de `ArrayList`.

De uitvoer van deze code is:

```
List op basis van LINKEDLIST
```

```
*** Met een for-statement ***
```

```
fiets
null
even
dak
citroen
citroen
boom
aap
```

```
*** Met een for-each ***
```

```
fiets
null
even
dak
citroen
citroen
boom
aap
```

```
*** Met de iterator ***
```

```
fiets
null
even
dak
citroen
citroen
boom
aap
```

```
Nieuwe elementen
```

```
*** Met een for-statement *** (enkel op deze manier getoond)
```

```
fiets
null
even
test
dak
citroen
citroen
boom
aap
beer
```

```
Element vervangen
```

```
LinkedList afgedrukt vanaf element 2
```

```
even
test
dak
citroen
tweede citroen
kers
boom
```

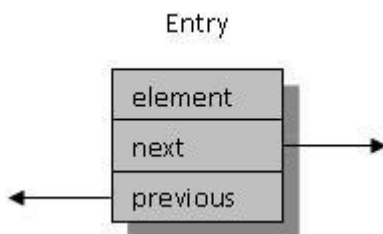


```
aap
beer

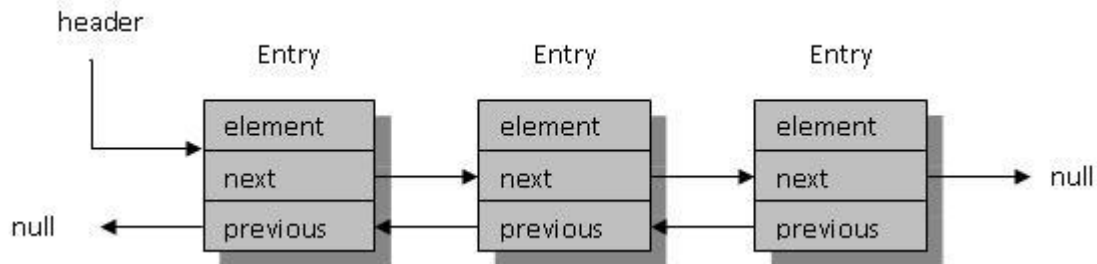
LinkedList omgekeerd afgedrukt
beer
aap
boom
kers
tweede citroen
citroen
dak
test
even
null
fiets
```

### 12.3.3.2 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:



We sommen enkele extra methods van de LinkedList op:

Method	Return-waarde	Toelichting
<code>addFirst(E e)</code>	void	Voegt element E tussen, aan het begin van de lijst.
<code>addLast(E e)</code>	void	Voegt element E achteraan toe, aan het einde van de lijst.
<code>getFirst()</code>	E	Returnt het eerste element van de lijst.
<code>getLast()</code>	E	Returnt het laatste element van de lijst.
<code>removeFirst()</code>	E	Returnt en verwijdert het eerste element van de lijst.



<code>removeLast()</code>	E	Returnt en verwijdert het laatste element van de lijst.
---------------------------	---	---------------------------------------------------------

Breid het voorbeeld uit met volgende code:

```
...
System.out.print("\nLinkedList omgekeerd afgedrukt");
toonVanafIndexOmgekeerd(ll, ll.size());

//extra methods van de LinkedList
System.out.println("\nExtra methods van de LinkedList");
LinkedList llijst = new LinkedList();           (1)
vul(llijst);                                   (2)

System.out.println(llijst.getFirst() );         (3)
System.out.println(llijst.getLast() );          (4)

llijst.addFirst("eerste");                       (5)
llijst.addLast("laatste");                       (6)

System.out.println(llijst.getFirst() );          (7)
System.out.println(llijst.getLast() );

System.out.println(llijst.removeFirst() );        (8)
System.out.println(llijst.removeLast() );         (9)

System.out.println(llijst.getFirst() );          (10)
System.out.println(llijst.getLast() );
```

- (1) Er wordt een collectie gemaakt van een `LinkedList`. De declaratie van de variabele `llijst` moet van het type `LinkedList` zijn, juist omdat we specifieke methods van de `LinkedList` gaan gebruiken!
- (2) De method `vul` wordt opgeroepen en de `LinkedList` wordt als parameter meegegeven.
- (3) Het resultaat van de method `getFirst()` wordt getoond op scherm. De method `getFirst()` geeft het eerste object uit de collectie terug.
- (4) Het resultaat van de method `getLast()` wordt getoond op scherm. De method `getLast()` geeft het laatste object uit de collectie terug.
- (5) De method `addFirst()` voegt een nieuw eerste element toe aan de collectie.
- (6) De method `addLast()` voegt een nieuw laatste element toe aan de collectie.
- (7) Opnieuw wordt het eerste en laatste element uit de collectie getoond. Dit zijn de zopas toegevoegde elementen “eerste” en “laatste”.
- (8) 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.

- (9) 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.
- (10) Opnieuw wordt het eerste en laatste element uit de collectie getoond. Dit zijn terug de oorspronkelijke elementen, vermits de nieuw toegevoegde elementen weer uit de collectie verwijderd zijn.

De extra uitvoer van deze code is:

```
...  
Extra methods van de LinkedList  
fiets  
aap  
eerste  
laatste  
eerste  
laatste  
fiets  
aap
```

#### 12.3.4 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.

#### 12.3.5 Oefeningen



Maak oefening 26 uit de oefenmap.



## 12.4 De set interface

Eerder is reeds het volgende gezegd:

- Een Set kan geen dubbele elementen bevatten..
- Een SortedSet is een afgeleide interface van Set en kan dus ook geen dubbele elementen bevatten. De SortedSet is altijd gesorteerd in stijgende volgorde.

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 we deze implementaties bespreken, geven we eerst kort wat uitleg over de hashcode en de hashtable.

### 12.4.1 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 ook alle classes. Zo heeft de class String deze method geërfd en zo kan je zelf die method voorzien 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 returnt 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 we zelf een eigen class schrijven is het best dat we, net zoals de `equals()` method, ook altijd een `hashCode()` method schrijven. Standaard geeft de `hashCode()` method van `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 `Set`. 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 geretund worden.

Een **vuistregel** om de hashcode te berekenen:

- Gebruik de sleutel van het object, dus het data member dat je object uniek identificeert (voor een klant is dat bijv. het klantnummer).
- 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. We sommen even de regels in verband hiermee op.

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 gegevens 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.



### 12.4.2 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.

### 12.4.3 Hoe werken de HashSet en de LinkedHashSet?

Vooraleer de HashSet en de LinkedHashSet verder bekeken worden, staan we even stil bij de werking van de Set. We lichten dit toe aan de hand van een voorbeeld.

We beschikken over een class `Coördinaat` met 2 data members nl. de x- en y-coördinaat, beiden integers. Stel nu dat we een collection (een Set) willen 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 noemen we `coll` en we gaan achtereenvolgens een aantal objecten toevoegen aan de collection:

```
1) coll.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 collection:

<i>Tussentabel</i>		<i>Collection coll</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 collection:

```
coll.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 collection:

<i>Tussentabel</i>		<i>Collection coll</i>	
0		0	Coördinaat(11,20) -1
1		1	Coördinaat(13,20) -1
2			
3	0		
4			
5	1		
6			



Het tweede element in de collection 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 Coördinaat, 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 collection:

```
coll.add(new Coördinaat(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 collection: 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 collection 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	Coördinaat(11,20)	2
1	Coördinaat(13,20)	-1
2	Coördinaat(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:

```
coll.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 hashCode() overschrijft. 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!**

#### 12.4.4 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:

```
public static void main(String[] args)
{ System.out.print("Set op basis van HashSet");
  Set hs = new HashSet();           (1)
  vul(hs);                          (2)
  toon(hs);                         (3)
}

private static void vul(Set s)
{ s.add("fiets");                   (4)
  s.add("even");
  s.add("dak");
  s.add("citroen");
  s.add("boom");
  s.add("aap");
}
```



```
private static void toon(Set s)
{ System.out.println();

  for ( Object obj : s) {                                (5)
    System.out.println(obj + "\t" + obj.hashCode() );    (6)
  }
}
```

- (1) Er wordt een collectie gemaakt van het type HashSet.
- (2) De method *vul()* wordt opgeroepen en de hashset wordt als parameter meegegeven.
- (3) Nadat de hashset gevuld is, wordt de method *toon()* opgeroepen en ook hier wordt de hashset als parameter meegegeven.
- (4) Er worden verschillende string-objecten toegevoegd aan de hashset.
- (5) Je itereert over de collectie. Je itereert steeds van het begin tot het einde over de collectie. Je start vooraan de collectie, bij het eerste element en je itereert tot het einde van de collectie.  
Er zijn geen methods ter beschikking om van achter naar voren te itereren over de collectie of om je ergens midden in de collectie te positioneren.
- (6) Het element wordt getoond. Daarnaast wordt, gewoon ter info, ook het resultaat van de *hashCode()* getoond.

De uitvoer van deze code is:

```
Set op basis van 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.

We gaan nu eens proberen om een null-element en een dubbel toe te voegen:

### Bijvoorbeeld:

```
...
private static void vul(Set s)
{ s.add("fiets");
  s.add("even");
  s.add("dak");
  s.add("dak");
  s.add("citroen");
  s.add("boom");
  s.add(null);
  s.add("aap");
}

private static void toon(Set s)
{ System.out.println();

  for ( Object obj : s) {
    //System.out.println(obj + "\t" + obj.hashCode() );
    System.out.println(obj);
  }
}
```

(1)

- (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.

#### 12.4.5 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 linked list worden aangepast en dat kost extra tijd.

Het itereren over de lijst gaat sneller omdat dan de linked list kan worden gebruikt. Dit gaat sneller dan het berekenen van de plaats via een formule.

Een voorbeeld:

```
public static void main(String[] args)
{
    System.out.print("Set op basis van LinkedHashSet");
    Set lhs = new LinkedHashSet();
    vul(lhs);
    toon(lhs);
}

private static void vul(Set s)
{
    s.add("fiets");
    s.add("even");
    s.add("dak");
    s.add("citroen");
    s.add("boom");
    s.add("aap");
}

private static void toon(Set s)
{
    System.out.println();

    for ( Object obj : s) {
        System.out.println(obj + "\t" + obj.hashCode() );
    }
}
```

- (1) Er wordt een collectie gemaakt van het type `LinkedHashSet`. Verder gebeurt in deze code hetzelfde als wat we reeds besproken hebben bij het voorbeeld van de `HashSet`.

De uitvoer van deze code is:

```
Set op basis van LinkedHashSet
fiets      97427969
even       3125530
dak        99214
citroen    785246580
boom       3029739
aap        96336
```

Merk op dat de volgorde behouden blijft! De elementen worden weer uit de collection gehaald in de volgorde zoals ze zijn toegevoegd. Dit komt door het principe van de linked list die gebruikt wordt om de elementen in de collection op te slaan.

We gaan ook hier eens proberen om een null-element en een dubbel toe te voegen:

Bijvoorbeeld:

```
...
private static void vul(Set s) {
    s.add("fiets");
    s.add("even");
    s.add("dak");
    s.add("dak");
    s.add("citroen");
    s.add("boom");
    s.add(null);
    s.add("aap");
}

private static void toon(Set s) {
    System.out.println();

    for ( Object obj : s) {
        //System.out.println(obj + "\t" + obj.hashCode() );
        System.out.println(obj);
    }
}
```

(1)

- (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 LinkedHashSet
fiets
even
dak
citroen
boom
null
aap
```

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*. Het null-element wordt wel bewaard.

### 12.4.6 De TreeSet

Ook in een TreeSet bewaar je objecten. Een belangrijke eigenschap van een TreeSet is dat de elementen gesorteerd worden opgeslagen.

- Is een implementatie van de SortedSet. De elementen uit de collection 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.

We bespreken enkele van deze extra methods:

Method	Returnwaarde	Toelichting
<code>first()</code>	<code>E</code>	Geeft het eerste, dus laagste, element terug van de gesorteerde set.
<code>last()</code>	<code>E</code>	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 verwijzen we naar de documentatie.

#### 12.4.6.1 Natural ordering

In de `TreeSet` wordt de volgorde bepaald door de `compareTo()` method, van de interface `Comparable`.

Eerst een voorbeeld:

```
public static void main(String[] args)
{ System.out.print("Set op basis van TreeSet");
  Set ts = new TreeSet();
  vul(ts);
  toon(ts);
}

private static void vul(Set s)
{ s.add("fiets");
  s.add("even");
  s.add("dak");
  s.add("citroen");
  s.add("boom");
  s.add("aap");
}
```

(1)

```
private static void toon(Set s)
{ System.out.println();

    for ( Object obj : s) {
        System.out.println(obj);
    }
}
```

- (1) Er wordt een collectie gemaakt van het type `TreeSet`. Verder gebeurt in deze code hetzelfde zoals we dat reeds besproken hebben in de vorige voorbeelden.

De uitvoer van deze code is:

```
Set op basis van TreeSet
aap
boom
citroen
dak
even
fiets
```

Merk op dat de volgorde alfabetisch is, bovendien in stijgende volgorde. Deze volgorde is zo 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.

We gaan ook hier eens proberen om een null-element en een dubbel toe te voegen:

Bijvoorbeeld:

```
...
private static void vul(Set s)
{ s.add("fiets");
  s.add("even");
  s.add("dak");
  s.add("dak");
  s.add("citroen");
  s.add("boom");
  s.add(null);
  s.add("aap");
}

private static void toon(Set s)
{ System.out.println();

    for ( Object obj : s) {
        System.out.println(obj);
    }
}
```



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 we de regel `s.add(null)` in commentaar zetten en het programma opnieuw uitvoeren zien we 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.

We gaan de code uitbreiden met enkele extra methods:

Wijzig hiervoor eerst de regel `Set ts = new TreeSet();` naar `SortedSet ts = new TreeSet();` omdat we gaan gebruiken maken van methods die specifiek zijn voor de `SortedSet`.

```
public static void main(String[] args)
{
    ...
    //Extra methods
    System.out.println("*** Extra methods ***");
    System.out.println("Eerste element: " + ts.first() );           (1)
    System.out.println("Laatste element: " + ts.last() );          (2)

    SortedSet ss = ts.subSet("boom", "even");                      (3)
    System.out.print("*** SubSet van de TreeSet vanaf 'boom' tot 'even' ");
    toon(ss);                                                        (4)
    ...
}
```

- (1) De method `first()` geeft ons het eerste of laagste element van de treeset.
- (2) De method `last()` geeft ons het laatste of hoogste element van de treeset.
- (3) Vervolgens creëren we een nieuwe sortedset via de method `subSet (elementVanaf, elementTot )`: 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'.
- (4) Vervolgens wordt deze subset doorgegeven aan de method `toon()` die m.b.v. de iterator over de set itereert om vervolgens elk element te tonen.

De uitvoer van dit extra stukje code is :

```
*** Extra methods ***
Eerste element: aap
Laatste element: fiets
*** SubSet van de TreeSet vanaf 'boom' tot 'even'
boom
citroen
dak
```



### 12.4.6.2 *Natural ordering – compareTo()*

In het vorige voorbeeld lag de natural ordering vast door de `compareTo()` van de class `String`, nl. oplopend alfabetisch. Daar hebben wij niets over te zeggen.

Aan de hand van een voorbeeld laten we zien hoe we zelf de volgorde kunnen bepalen van een aantal objecten die we bewaren in een `TreeSet`.

We maken een class `Cursus`:

```
public class Cursus implements Comparable                (1)
{
    private int cursusNr;
    private String cursusNaam;
    private int prijs;

    public Cursus(int nr, String naam, int prijs)
    { cursusNr = nr;
      cursusNaam = naam;
      this.prijs = prijs;
    }

    public int getCursusNr()
    { return cursusNr; }

    public void setCursusNr(int nr)
    { this.cursusNr = nr; }

    public String getCursusNaam()
    { return cursusNaam; }

    public void setCursusNaam(String naam)
    { this.cursusNaam = naam; }

    public int getPrijs()
    { return prijs; }

    public void setPrijs(int prijs)
    { this.prijs = prijs; }

    @Override
    public String toString()
    { return (cursusNr + "\t" + cursusNaam + "\t" + prijs);
    }

    @Override
    public boolean equals (Object o) {                    (2)
        if (!(o instanceof Cursus)) {
            return false;
        }
        Cursus c = (Cursus) o;
        return cursusNr == c.getCursusNr()
    }

    @Override
    public int hashCode()                                (3)
    { return cursusNr;
    }

    @Override
    public int compareTo(Object o) {                    (4)
```



```
// sorteren op cursusnr
Cursus c = (Cursus) o;
return cursusNr - c.getCursusNr()
}
}
```

- (1) De class Cursus implementeert de interface Comparable. Dat betekent dat de compareTo() method overriden moet worden.
- (2) We schrijven de equals() method uit: 2 cursusobjecten zijn aan elkaar gelijk wanneer hun cursusnummers aan elkaar gelijk zijn.
- (3) Best is om altijd de hashCode() method te voorzien. We houden het hier eenvoudig en geven het cursusnr terug. Het cursusnr is uniek, we gebruiken dit ook bij equals() om te testen op gelijkheid.
- (4) Hier wordt de sortering of vergelijking bepaald van 2 cursus-objecten. Wanneer is een cursusobject kleiner, groter of gelijk aan een ander cursusobject?
- (5) Het vergelijkende object wordt omgezet naar een cursusobject. Indien dit object niet van het type Cursus is, dan wordt er automatisch een ClassCastException gethrowed. Indien het object 'null' is, wordt er automatisch een NullPointerException gethrowed (zie Javadoc).  
We vergelijken hier de cursusnummers met elkaar door het verschil van de cursusnummers terug te geven: een cursus-object is kleiner dan een ander cursusobject wanneer zijn cursusnr kleiner is (het verschil is dan negatief). Het is groter dan een ander cursusobject wanneer zijn cursusnr 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 is **consistent** met de equals() method. Dat is belangrijk! We komen daar nog op terug.

Schrijf hierbij nu het volgend main-programma:

```
public static void main(String[] args)
{
    Set 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 ( Object obj : cursussen) {
        Cursus eenCursus = (Cursus) obj;
        System.out.println(eenCursus);
    }
}
```

- (1) Er wordt een collectie gemaakt van een TreeSet.
- (2) Er worden 5 objecten van de class Cursus toegevoegd aan deze collectie.

- (3) De collectie wordt overlopen, elk object wordt uit de collectie gehaald, omgezet naar het juiste type nl. `Cursus` en getoond (`System.out.println()` gebruikt de `toString()` van `Cursus`).

De uitvoer van deze code is:

```
1   Windows      90
2   Powerpoint   80
3   Excel        110
4   Access       120
5   Word         100
```

Je ziet 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`. Dat is ook de volgorde beschreven in de `compareTo()` method.

We gaan deze volgorde nu aanpassen: we wensen de cursussen te tonen in volgorde van hun cursusnaam. Wijzig daarom in de class `Cursus` de `compareTo()` method als volgt (plaats eerst de code die er nu staat tussen commentaar):

```
@Override
public int compareTo(Object o) {                               (1)

    // sorteren op cursusNaam
    Cursus c = (Cursus) o;
    return cursusNaam.compareTo(c.getCursusNaam());           (2)
}
```

- (1) Hier wordt de sortering of vergelijking bepaald van 2 cursus-objecten. Wanneer is een cursusobject kleiner, groter of gelijk aan een ander cursusobject?
- (2) We vergelijken hier de cursusnamen met elkaar. De membervariabele `Cursusnaam` is een `String` en daarom gebruiken we de `compareTo()` van de class `String`: 2 cursusobjecten zijn aan elkaar gelijk wanneer de cursusnamen aan elkaar gelijk zijn. Opgelet: dat is **niet** meer **consistent** met de `equals()` method. Daar zijn 2 cursusobjecten aan elkaar gelijk wanneer hun cursusnummers aan elkaar gelijk zijn. Dit kan problemen geven, doch dit hoeft nog niet meteen een probleem te zijn: we kunnen ons voorstellen dat de cursusnamen die we willen bewaren in een collection altijd verschillend zijn van elkaar. Dat is niet noodzakelijk het geval wanneer je namen van personen in een collectie gaat bewaren!

Voer het main-programma nog eens uit. De uitvoer is:

```
4   Access       120
3   Excel        110
2   Powerpoint   80
1   Windows      90
5   Word         100
```

Je ziet dat de volgorde opnieuw anders is dan bij het opvullen van de collectie. De volgorde is hier bepaald door de cursusnaam. De cursussen worden getoond in volgorde van hun cursusnaam, alfabetisch in stijgende volgorde. Dat is ook de volgorde beschreven in de `compareTo()` method van de class `String`, die gebruikt wordt in onze eigen class `Cursus`.



We gaan nog een stap verder. We wensen de cursussen te tonen op volgorde van hun cursusprijs.

Wijzig in de class `Cursus` de `compareTo()` method nog eens als volgt (plaats de code die er nu staat tussen commentaar):

```
@Override
public int compareTo(Object o) {                                (1)

    // sorteren op cursusPrijs
    Cursus c = (Cursus) o;
    return prijs - c.getPrijs()                                (2)
}
```

- (1) Hier wordt de sortering of vergelijking bepaald van 2 cursus-objecten. Wanneer is een cursusobject kleiner, groter of gelijk aan een ander cursusobject?
- (2) We vergelijken hier de cursusprijs met elkaar door het verschil van de cursusprijs terug te geven: een cursus-object 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 (het verschil is dan positief) is en de twee cursus-objecten zijn gelijk wanneer de prijs aan elkaar gelijk is (het verschil is dan 0). Opnieuw is dit **niet consistent** met de `equals()` method! Dit is inderdaad geen realistische vergelijking meer, maar we wensen aan te tonen hoe belangrijk de `compareTo()` en de `equals()` is.

Voer het main-programma nog eens uit. De uitvoer is:

```
2    Powerpoint  80
1    Windows     90
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 is hier bepaald door de prijs. De cursussen worden getoond in volgorde van hun prijs. Dat is ook de volgorde beschreven in de `compareTo()` method.

Een probleem gaat ontstaan wanneer je een cursus wenst toe te voegen met een prijs die gelijk is aan één van de andere cursussen. Dat kan al vlugger voorkomen dan dat je een cursus gaat toevoegen met dezelfde naam.

We gaan dit testen: voeg daarom volgende regel toe aan je main-programma:

```
...
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) );
cursussen.add(new Cursus(6, "PhotoShop", 100) );                (1)
...
```

(1) We gaan een cursus toevoegen met een prijs gelijk aan die van de cursus Word.

Voer het main-programma uit en de uitvoer is:

```
2   Powerpoint  80
1   Windows     90
5   Word        100
3   Excel       110
4   Access      120
```

Je stelt vast dat de extra cursus Photoshop niet is bewaard in de collectie: een TreeSet bevat geen dubbels en in de `compareTo()` hebben we beschreven dat 2 cursus-objecten aan elkaar gelijk zijn wanneer hun prijs gelijk is. Hier geeft dat dus een probleem.

Dit kan opgelost worden door de `compareTo()` aan te passen en ervoor te zorgen dat de `equals()` en de `compareTo()` consistent zijn!! Dit is zeer belangrijk!

In dit voorbeeld betekent dat dat de cursussen aan elkaar gelijk zijn wanneer hun cursusnummers aan elkaar gelijk zijn en niet wanneer hun prijs gelijk is.

Wijzig daarom de `compareTo()` van class Cursus als volgt:

```
@Override
public int compareTo(Object o) {

    // sorteren op cursusPrijs
    Cursus c = (Cursus) o;
    if ( this.equals(c) )                (1)
        return 0;
    else
        return prijs == c.getprijs() ? -1 : prijs - c.getprijs();  (2)
}
```

(1) We gaan de cursussen met elkaar vergelijken en geven 0 terug wanneer ze aan elkaar gelijk zijn zoals beschreven in de `equals()` method. In alle andere gevallen zal nooit 0 teruggegeven worden.

(2) Wanneer de prijs gelijk is aan de andere prijs, geven we -1 terug. Ook bij gelijke prijs dient de cursus op de juiste plaats tussengevoegd te worden. In het andere geval geven we het verschil van de prijs terug.

Voer het main-programma opnieuw uit. De uitvoer is:

```
2   Powerpoint  80
1   Windows     90
6   PhotoShop   100
5   Word        100
3   Excel       110
4   Access      120
```

De cursussen worden getoond in volgorde van hun prijs en ook de cursus PhotoShop, met dezelfde prijs als de cursus Word wordt getoond.



### 12.4.6.3 Consistentie van `equals()` en `compareTo()`

We hebben bij het voorbeeld vastgesteld dat het belangrijk is dat de `equals()` en `compareTo()` consistent zijn.

De documentatie van de JDK zegt dan ook over het implementeren van `Comparable`:

*It is strongly recommended, but not strictly required that*

*$(x.compareTo(y) == 0) == (x.equals(y))$*

*Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is “Note: this class has a natural ordering that is inconsistent with `equals`.”*

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. En 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()`.

### 12.4.6.4 Meerdere sorteringen

Je kan in je class slechts één volgorde bepalen in de `compareTo()` method. We hebben steeds de `compareTo()` method gewijzigd. De volgorde die je hier bepaalt, is de **natural ordering** van de objecten van deze class, die gebruikt wordt bij de `TreeSet`.

Verderop in de cursus in het hoofdstuk van Sorted collection implementations zullen we zien hoe we meerdere sorteringen van dezelfde instanties kunnen bekomen.

## 12.4.7 Oefeningen



Maak oefening 27 uit de oefenmap.

## 12.5 Generics

We hebben geleerd dat je objecten van een willekeurig type in een collection kan opbergen. Dat lijkt misschien wel handig, maar dat is het in de praktijk niet, omdat je bij het opvragen van de elementen in het algemeen een typecast moet gebruiken, en dit werkt alleen goed als je weet welk type de elementen in de collection hebben.

In feite komt het erop neer dat je er beter voor zorgt dat alle elementen hetzelfde type hebben, bijvoorbeeld door alle elementen een gemeenschappelijke interface te laten implementeren, of te laten erven van dezelfde superclass.

Dit leidt meteen tot een bezwaar, want er is geen standaardmechanisme om te controleren of de elementen van de collectie werkelijk van hetzelfde type zijn. Je moet er zelf voor zorgen dat alle elementen hetzelfde type hebben. Maar zelfs als je daarvoor hebt gezorgd, moet je een typecast gebruiken bij het opvragen. Het is vervelend om steeds een typecast te moeten gebruiken, en het maakt broncode minder goed leesbaar.

Omdat er veel broncode bestaat die gebruik maakt van deze “oude” wijze, is het toch belangrijk die te kennen.

Vanaf java 5 bestaat er een mechanisme in de vorm van generics dat de gestelde bezwaren ondervangt.

We geven vooraf een voorbeeld waarbij door typecasting de objecten omgezet worden naar hun eigenlijk type om over de volledige functionaliteit van het object te kunnen beschikken:

```
Set coll = new HashSet();

coll.add( new Cursus(5, "Word", 100) );
coll.add( new Cursus(3, "Excel", 110) );
coll.add( new Cursus(1, "Windows", 90) );
coll.add( "test");
coll.add( new Cursus(4, "Access", 120) );
coll.add( new Cursus(2, "Powerpoint", 80) );

int somPrijs = 0;

Iterator it = coll.iterator();
while( it.hasNext() )
{
    Object o = it.next();
    if ( o instanceof Cursus )
    {
        Cursus c = (Cursus) o;
        somPrijs += c.getPrijs();
    }
}
System.out.println("Som van de prijzen: " + somPrijs);
```

- (1) Een collection is een verzameling van objecten, dus er kunnen verschillende objecten bewaard worden in één collection. Echt nuttig is dat niet. Een dergelijke collection is niet erg veilig en handig, maar het kan. Hier worden 5 objecten van type Cursus en 1 string-object bewaard in dezelfde collection.



- (2) Alvorens het object uit de collection zonder meer te typecasten naar type `Cursus`, wordt eerst getest of dit kan. Immers het string-object "test" kan niet getypecast worden naar type `Cursus`, dit zou een exception veroorzaken. Ook het oproepen van de method `getPrijs()` kan niet voor het string-object en mag dus niet gebeuren voor dit object. Op deze manier wordt dit object overgeslaan voor het berekenen van de som van de prijzen.

➔ Dit kan dus eenvoudiger door het gebruik van **generic collections**.

### 12.5.1 Generic classes

Alvorens we generic collections toelichten, eerst iets meer over generic classes, of ook wel generic types genoemd.

Een generic type maak je d.m.v. een class met één of meer parameters. Een voorbeeld is de volgende class voor het generic type `Test<E>`.

```
//Generic class
class Test<E> {
    private E x;

    public Test(E x) {
        this.x = x ;
    }

    public E getX(){
        return x;
    }
}
```

Zo'n class heet ook wel een **geparametriseerde klasse** of in het Engels een **parameterized class**. De parameter heeft in dit voorbeeld de naam `E`. In plaats van `E` kun je elke andere geldige Java-naam nemen, maar het is gebruikelijk zo'n parameter een naam van één letter te geven. De namen `E`, `T`, `K` en `V` komen veel voor in de Java-library.

De parameter `E` staat voor een nog niet bekend type, maar je gebruikt deze `E` alsof het een standaard type is. Pas als je een referentie declareert naar een instantie van de generic class moet je opgeven welk type `E` precies is. Een parameter zoals `E` wordt **type parameter** genoemd.

Overall in de geparametriseerde klasse kun je gebruik maken van de parameter `E`. Als type voor een attribuut, voor een argument of een lokale variabele in een methode, of voor het type van de returnwaarde.



Hieronder zie je drie voorbeelden van het gebruik van dit generic type uitgewerkt voor de class Test:

```
public class VbGenerics {
    public static void main(String[] args)
    {
        //declaratie 1
        Test<Integer> ti;
        ti = new Test<Integer>(3);
        System.out.println(ti.getX() );

        //declaratie 2
        Test<String> ts;
        ts = new Test<String>("Over generics");
        System.out.println(ts.getX() );

        //declaratie 3
        Test<Cursus> tc;
        tc = new Test<Cursus>( new Cursus(7, "Java", 100) );
        System.out.println(tc.getX() );
    }
}
```

De uitvoer zal zijn:

```
3
Over Generics
7   Java   100
```

Een geparametriseerde klasse is eigenlijk geen klasse, maar meer een mechanisme om andere klassen te maken. Zo wordt bij de eerste declaratie, `Test<Integer> ti`, de parameter E vervangen door type Integer. Het gevolg is dat de klasse Test er zo komt uit te zien:

```
class Test<Integer> {
    private Integer x;

    public Test(Integer x) {
        this.x = x ;
    }

    public Integer getX(){
        return x;
    }
}
```

Deze klasse heet een type-instantie van de generic class.

Merk op dat bij de aanroep van de constructor de primitieve int-waarde 3 automatisch wordt omgezet naar de wrapper class Integer. Dit is autoboxing.



Bij de tweede declaratie, `Test<String> ts`, wordt `E` vervangen door `String`. Hierdoor ontstaat de volgende type-instantie:

```
class Test<String> {  
    private String x;  
  
    public Test(String x) {  
        this.x = x ;  
    }  
  
    public String getX(){  
        return x;  
    }  
}
```

Zo wordt bij de derde declaratie het type `Cursus` ingevuld voor `E`:

```
class Test<Cursus> {  
    private Cursus x;  
  
    public Test(Cursus x) {  
        this.x = x ;  
    }  
  
    public Cursus getX(){  
        return x;  
    }  
}
```

Het gebruik van geparametriseerde klassen heeft een aantal voordelen, zeker bij het maken van collecties. Je kunt code voor een collectie schrijven zonder het type van de elementen die in de collectie komen, vast te leggen. Zo zijn de generic collections ontstaan.

### 12.5.2 Interface Comparable<T>

De interface `Comparable` is een generieke interface. Met het type `<T>` geef je aan met wat voor soort objecten de instanties van klassen die de interface implementeren vergeleken kunnen worden. In het algemeen zijn dat objecten van hetzelfde type, bijvoorbeeld :

```
class Integer implements Comparable<Integer>
```

*Dus integers worden met integers vergeleken*

```
class Cursus implements Comparable <Cursus>
```

*Dus cursussen worden met cursussen vergeleken*

Je geeft dus eigenlijk aan dat je objecten van de class `Integer` met objecten van de class `Integer` kunt vergelijken met behulp van de method `compareTo()`.

In het tweede voorbeeld geef je aan dat je objecten van de class `Cursus` met objecten van de class `Cursus` kunt vergelijken met behulp van de method `compareTo()`.

We passen dit toe bij de class Cursus :

```
public class Cursus implements Comparable<Cursus>
{
    private int cursusNr;
    private String cursusNaam;
    private int prijs;

    public Cursus(int nr, String naam, int prijs) {
        cursusNr = nr;
        cursusNaam = naam;
        this.prijs = prijs;
    }

    public int getCursusNr() {
        return this.cursusNr;
    }

    public void setCursusNr(int nr) {
        this.cursusNr = nr;
    }

    public String getCursusNaam() {
        return this.cursusNaam;
    }

    public void setCursusNaam(String naam) {
        this.cursusNaam = naam;
    }

    public int getPrijs() {
        return this.prijs;
    }

    public void setPrijs(int prijs) {
        this.prijs = prijs;
    }

    @Override
    public String toString() {
        return (cursusNr + "\t" + cursusNaam + "\t" + prijs);
    }

    @Override
    public boolean equals (Object o) {
        if (!(o instanceof Cursus)) {
            return false;
        }
        return cursusNr == c.getCursusNr()
    }

    @Override
    public int hashCode() {
        return cursusNr;
    }
}
```



```

@Override
public int compareTo(Cursus c) {
    // // sorteren op cursusnr
    // if (cursusNr < c.getCursusNr() )
    //     return -1;
    // else
    //     if (cursusNr > c.getCursusNr() )
    //         return 1;
    //     else
    //         return 0;

    // // sorteren op cursusNaam
    // return cursusNaam.compareTo(c.getCursusNaam());

    // // sorteren op cursusPrijs: niet consistent met equals()
    // if (prijs < c.getPrijs())
    //     return -1;
    // else
    //     if (prijs > c.getPrijs())
    //         return 1;
    //     else
    //         return 0;

    // sorteren op cursusPrijs: consistent met equals()
    if (this.equals(c))
        return 0;
    else {
        return prijs == c.getprijs() ? -1 : prijs - c.getprijs();
    }
}

```

(1) De code wordt steeds een stuk korter:

- Je hoeft het vergelijkende object niet te casten naar een object van hetzelfde type, hier dus van het type `Cursus`.
- Daardoor valt ook de controle weg of het vergelijkende object wel een object is en niet *null* is, want de methods die je uitvoert op het object `c` (bijv. `c.getCursusNr`) leveren dan een `NullPointerException` op.

Er staan 3 sorteervijzen in commentaar. Je kan er immers maar één hebben in de `compareTo()` method. Ze zijn hier toch weergegeven omdat de class `Cursus` zo is opgebouwd in dit hoofdstuk. Hier zijn alle sorteervijzen aangepast.

### 12.5.3 Generic collections

De ontwerpers van het Collections Framework hebben dankbaar gebruik gemaakt van generic classes (of generic types). Een collection is een class, dus een generic collection is een generic class.

- Als je een generic collection gebruikt, controleert de compiler of de objecten die je in de collectie stopt van het juiste type zijn, zo niet, dan krijg je een foutmelding of waarschuwing. Zo'n bericht in compile-time is veel veiliger dan

een exceptie die in runtime optreedt en waardoor mogelijk het programma vastloopt.

- Dankzij generics hoef je ook geen typecast te gebruiken wanneer je een element of object uit de collectie haalt.

Wanneer we gebruik maken van generics, geven we zowel bij de declaratie van de reference van de collection als bij het oproepen van de constructor van de collection, het type van de inhoud op: type object voor de collection. Dit type vermelden we steeds tussen `< >`. Ook bij de declaratie van de iterator dienen we het type te vermelden. Het returntype van de `next()` method is dan het opgegeven datatype en niet meer `Object`.

We passen het voorbeeld aan:

```
Set<Cursus> coll = new HashSet<Cursus>(); // (1)

coll.add( new Cursus(5, "Word", 100) );
coll.add( new Cursus(3, "Excel", 110) );
coll.add( new Cursus(1, "Windows", 90) );
coll.add( "test" ); // (2)
coll.add( new Cursus(4, "Access", 120) );
coll.add( new Cursus(2, "Powerpoint", 80) );

int somPrijs = 0;

Iterator<Cursus> it = coll.iterator(); // (3)
while( it.hasNext() )
{
    Cursus c = it.next(); // (4)
    somPrijs += c.getPrijs();
}
System.out.println("Som van de prijzen: " + somPrijs);
```

- (1) De declaratie van een referentie en het maken van een instantie van een `HashSet` voor `Cursus`-objecten. Er wordt hier dus een collection gemaakt van `Cursus`-objecten. Dat betekent dat in deze collection slechts enkel objecten van het type `Cursus` bewaard kunnen worden. Zowel bij de declaratie als bij het oproepen van de constructor van de collection plaatsen we het type tussen `< >`.
- (2) Deze regel moet hier geschrapt worden: een `String`-object is geen `Cursus`-object en kan dus niet bewaard worden in deze collection.
- (3) Ook bij de declaratie van de iterator plaatsen we het type tussen `< >`.
- (4) Het returntype van de `next()` method is nu het opgegeven datatype. In dit voorbeeld dus `Cursus`. Typecasting is dus niet meer nodig en de `while`-lus is eigenlijk herleid naar deze 2 regels.

De uitvoer van de code is:

```
Som van de prijzen: 500
```



We voegen nog een tweede mogelijkheid toe om de collection te overlopen, nl. met de enhanced for-lus of de for-each:

```
// andere manier: met de enhanced for-lus of for-each
somPrijs = 0;
for (Cursus c : coll) {                                     (1)
    somPrijs += c.getPrijs();
}
System.out.println("Som van de prijzen: " + somPrijs);
```

- (1) Met de enhanced for lus wordt de collection overlopen. Vermits het een collection betreft van Cursus-objecten, zal er iedere keer een Cursus-object uit de collection gehaald worden en niet zoals voorheen een object van type Object. Dus typecasting is ook hier niet meer nodig. Een for-each is korter, leesbaarder en foutvrijer.

De uitvoer van de code is:

```
Som van de prijzen: 500
Som van de prijzen: 500
```

Wanneer we zouden werken met een ArrayList i.p.v. met een HashSet, zal ook de method `get()` een Cursus-Object geven en geen Object. We kunnen onderstaand code-fragment toevoegen aan de vorig opgebouwde code:

```
List<Cursus> al = new ArrayList<Cursus>();                 (1)

al.add( new Cursus(5, "Word", 100) );
al.add( new Cursus(3, "Excel", 110) );
al.add( new Cursus(1, "Windows", 90) );
al.add( new Cursus(4, "Access", 120) );
al.add( new Cursus(2, "Powerpoint", 80) );

somPrijs = 0;

for (int i=0; i<al.size(); i++) {

    Cursus c = al.get(i);                                   (2)
    somPrijs += c.getPrijs();
}
System.out.println("Som van de prijzen: " + somPrijs);
```

- (1) Er wordt een collection gemaakt van het type ArrayList voor objecten van type Cursus.
- (2) Met de method `get()` wordt het zoveelste element uit de collection gehaald. Dit is een Cursus-object en geen Object, dus typecasting is niet meer nodig.

De uitvoer van de code is:

```
Som van de prijzen: 500
Som van de prijzen: 500
Som van de prijzen: 500
```

We geven nog een voorbeeld en gebruiken hiervoor de classes `Rekening`, `ZichtRekening` en `SpaarRekening` die gemaakt zijn in het hoofdstuk dat gaat over Inheritance. Controleer of je ook de `toString()` method overriden hebt in deze classes. Zo niet, doe dit dan alsnog.

```
public static void main(String[] args)
{
    // -----GENERICICS: eerste collectie van rekeningen -----
    Set<Rekening> setRek = new HashSet<Rekening>();           (1)

    ZichtRekening z1 = new ZichtRekening("001-1234567-11",1000);   (2)
    z1.storten(100.80);
    ZichtRekening z2 = new ZichtRekening("001-1234567-22",1000);
    z2.storten(200.80);
    SpaarRekening s1 = new SpaarRekening("833-1234567-88",3.5 );
    s1.storten(1500.0);
    SpaarRekening s2 = new SpaarRekening("833-1234567-99",3.5 );
    s2.storten(3000.0);

    setRek.add(z1);                                           (3)
    setRek.add(z2);
    setRek.add(s1);                                           (3)
    setRek.add(s2);

    double somSaldo = 0.0;

    for (Rekening rek : setRek)                               (4)
    {
        somSaldo += rek.geefSaldo();
    }

    System.out.println("Totaal saldo: " + somSaldo);          (5)

} // einde main
```

- (1) Er wordt een hashset-collectie gemaakt van het type `Rekening`, het hoogste niveau in de hiërarchie!
- (2) Vervolgens worden 2 zichtrekeningen en 2 spaarrekeningen gemaakt. Tevens wordt een bedrag op de rekening gestort, zodat het saldo niet 0 blijft.
- (3) Deze zicht- en spaarrekeningen worden toegevoegd aan de collectie. Vermits het een collectie betreft van `Rekening` kunnen alle objecten afgeleid van class `Rekening` bewaard worden in deze collectie. Immers een `ZichtRekening` is een `Rekening` en ook een `SpaarRekening` is een `Rekening`.
- (4) Bij het doorlopen van de collectie wordt van elk object (elke rekening) het saldo opgevraagd en gesommeerd.
- (5) Tot slot wordt deze som van het saldo getoond.

De uitvoer van deze code is:

Totaal saldo: 4959.0



### 12.5.4 Argumenten van een wildcard type

Wat we bedoelen met 'argumenten van een wildcard type' leggen we uit aan de hand van een voorbeeld. Breid het vorige main-programma als volgt uit:

```
...
System.out.println("HashSet van Rekeningen");
printRekening(setRek);                                     (1)
} //einde main

public static void printRekening(Collection <Rekening> bank) (2)
{   for(Rekening r : bank)                                  (3)
    System.out.println(r);
}
```

- (1) De method `printRekening` wordt opgeroepen en de `hashset` van `Rekeningen` wordt doorgegeven.
- (2) Deze set van rekeningen wordt ontvangen in een **Collection** `bank` van het type `Rekening`. Dus elke collectie van `Rekening` kan ontvangen worden in de collectie `bank`.  
Bovendien is het een `collection`: het type is niet gespecificeerd! Dus een `hashset` kan ontvangen worden, maar eender welke andere collectie met type `Rekening` kan ontvangen worden in de collectie `bank`!
- (3) Deze collectie wordt doorlopen en elk object (elke rekening) wordt getoond (de `toString()` van elke object wordt uitgevoerd).  
Let op: het is een `hashset`, dus de volgorde wordt bepaald door de `hashCode`. Indien de method `hashCode()` niet aanwezig is in de class `Rekening`, zal ze geërfd worden van `Object`.

De uitvoer van deze code is:

```
Totaal saldo: 4959.0
HashSet van Rekeningen
833-1234567-88  1552.5    3.5
001-1234567-11  100.75    1000
001-1234567-22  200.75    1000
833-1234567-99  3105.0    3.5
```

We breiden het main-programma nogmaals uit:

```
...
System.out.println("HashSet van Rekeningen");
printRekening(setRek);

List<Rekening> alRek = new ArrayList<Rekening>();          (1)
alRek.add(z1);                                             (2)
alRek.add(z2);
alRek.add(s1);
alRek.add(s2);

System.out.println("ArrayList van Rekeningen");
printRekening(alRek);                                     (3)

} //einde main
```



- (1) Er wordt een ArrayList gemaakt van het type Rekening, het hoogste niveau in de hiërarchie!
- (2) Vervolgens worden de 2 zichtrekeningen en 2 spaarrekeningen aan de arraylist toegevoegd. Ook hier geldt dat alle objecten afgeleid van class Rekening bewaard kunnen worden in deze collectie, vermits het een collectie betreft van Rekening.
- (3) De method printRekening wordt opgeroepen en de arrayList van Rekeningen wordt doorgegeven. Ook deze kan ontvangen worden in de collection bank! De volgorde van de elementen in de collection is de volgorde van toevoegen aan de collection.

De uitvoer is:

```
Totaal saldo: 4959.0
HashSet van Rekeningen
833-1234567-88  1552.5    3.5
001-1234567-11  100.75    1000
001-1234567-22  200.75    1000
833-1234567-99  3105.0    3.5
ArrayList van Rekeningen
001-1234567-11  100.75    1000
001-1234567-22  200.75    1000
833-1234567-88  1552.5    3.5
833-123467-99   3105.0    3.5
```

#### 12.5.4.1 Bounded wildcard

We breiden het main-programma nog verder uit:

```
...
System.out.println("ArrayList van Rekeningen");
printRekening(alRek);

Set<ZichtRekening> setZichtrek=new HashSet<ZichtRekening>();      (1)
setZichtrek.add(z1);                                              (2)
setZichtrek.add(z2);

System.out.println("Set van ZichtRekeningen");
printRekening(setZichtrek);                                       (3)

} //einde main
```

- (1) Er wordt opnieuw een HashSet gemaakt maar deze keer van type ZichtRekening.
- (2) Dan worden de 2 zichtrekeningen toegevoegd aan deze HashSet. Enkel ZichtRekening-objecten kunnen bewaard worden in deze collectie.
- (3) De method printRekening wordt opgeroepen en de hashset van zichtrekeningen wordt doorgegeven. Maar zoals je wellicht gemerkt hebt, protesteert de editor omdat de types niet overeenkomen: setZichtrek is een set met type ZichtRekening en kan niet ontvangen worden in de collectie van type Rekening.



We lossen dit op als volgt:

```
...
System.out.println("Set van ZichtRekeningen");
//printRekening(setZichtrek);           (1)
printRekening2(setZichtrek);           (2)

} //einde main

public static void printRekening2(Collection<? extends Rekening> bank)
{                                     (3)
    for(Rekening r : bank)
    {   System.out.println(r);
    }
}
```

- (1) Plaats de regel in commentaar of wis de regel: het kan niet uitgevoerd worden.
- (2) We roepen een nieuwe procedure met de naam `printRekening2` op en geven de hashset door.
- (3) Deze nieuwe procedure `printRekening2` heeft een ander argument, namelijk *`Collection<? extends Rekening> bank`*. Dit betekent dat elke collectie van type `Rekening` of afgeleid van `Rekening` ontvangen kan worden in de collectie `bank`. Dus de `setZichtrek` kan hier ontvangen worden.

We noemen dit een **bounded wildcard**: wildcard omdat je het type niet vastlegt, maar als een joker beschouwd, doch bounded: je legt beperkingen op aan de wildcard: elke collectie afgeleid van een bepaald type, hier van `Rekening`.

De uitvoer is:

```
Totaal saldo: 4959.0
HashSet van Rekeningen
833-1234567-88  1552.5  3.5
001-1234567-11  100.75  1000
001-1234567-22  200.75  1000
833-1234567-99  3105.0  3.5
ArrayList van Rekeningen
001-1234567-11  100.75  1000
001-1234567-22  200.75  1000
833-1234567-88  1552.5  3.5
833-1234567-99  3105.0  3.5
Set van ZichtRekeningen
001-1234567-11  100.75  1000
001-1234567-22  200.75  1000
```

### 12.5.4.2 Wildcard

We gaan nog een stap verder en voegen het onderstaande toe aan de main:

```
...
System.out.println("Set van ZichtRekeningen");
//printRekening(setZichtrek);
printRekening2(setZichtrek);

List <String> woorden = new ArrayList <String> () ;           (1)
woorden.add("eerste woord");                                 (2)
woorden.add("tweede woord");
woorden.add("derde woord");

System.out.println("Collectie van woorden");
printRekening(woorden);                                     (3)
printRekening2(woorden);                                    (4)
print(woorden);                                             (5)

} //einde main

public static void print(Collection < ? > coll) (6)
{
    for(Object obj : coll)
        System.out.println(obj);
}
```

- (1) Er wordt opnieuw een ArrayList gemaakt van type String.
- (2) Aan deze ArrayList worden enkele strings gevoegd.
- (3) De method `printRekening` wordt opgeroepen en de ArrayList van Strings wordt doorgegeven aan de *Collection<Rekening>*. Maar zoals je wellicht gemerkt hebt, protesteert de editor omdat de types niet overeenkomen: `woorden` is een collection van type String en kan niet ontvangen worden in de collectie van type Rekening. Zet de regel tussen commentaar.
- (4) De method `printRekening2` wordt opgeroepen en de ArrayList van Strings wordt doorgegeven aan de *Collection<? extends Rekening>*. Maar zoals je wellicht gemerkt hebt, protesteert ook hier de editor omdat de types niet overeenkomen: `woorden` is een collection van type String en kan niet ontvangen worden in de collectie van type Rekening of type afgeleid van Rekening. Zet de regel tussen commentaar.
- (5) De method `print` wordt opgeroepen en de ArrayList van Strings wordt doorgegeven.
- (6) De nieuwe method `print` heeft als argument *Collection <?> coll*. Dat betekent dat eender welke collectie ontvangen kan worden in dit argument. Er is geen type opgegeven.

We noemen dit een **wildcard**: je legt het type helemaal niet vast. Het type wordt beschouwd als een joker.



De uitvoer is:

```
Totaal saldo: 4959.0
HashSet van Rekeningen
833-1234567-88  1552.5    3.5
001-1234567-11  100.75    1000
001-1234567-22  200.75    1000
833-1234567-99  3105.0    3.5
ArrayList van Rekeningen
001-1234567-11  100.75    1000
001-1234567-22  200.75    1000
833-1234567-88  1552.5    3.5
833-1234567-99  3105.0    3.5
Set van ZichtRekeningen
001-1234567-11  100.75    1000
001-1234567-22  200.75    1000
Collectie van woorden
eerste woord
tweede woord
derde woord
```

We voegen nog een paar laatste regels toe aan de main:

```
...
print(wwoorden);

//Alle collections tonen met de procedure print
print(setRek);
print(alRek);
print(setZichtrek);

} //einde main
```

- (1) We gebruiken de nieuwe procedure *print* om ook onze andere collecties eens te tonen. Steeds wordt de collectie doorgegeven aan het hogere niveau Collection `<?>` en zoals je zal merken kan dit probleemloos uitgevoerd worden.

#### 12.5.4.3 Welke wildcard typen bestaan er?

Zoals in de voorbeelden aangetoond werd, kunnen methods een argument hebben met een vraagteken. Dit vraagteken staat voor een onbekend type, het is een wildcard of joker. Java kent 3 wildcard typen:

- `<?>`
- `<? extends E>`
- `<? super E>`

- 1) Een argument met het wildcard type `Collection<?>` accepteert een referentie van het type `Collection<E>`, waarbij `E` een willekeurig type is.
- 2) Het type `Collection<? extends E>` geeft aan dat het argument een referentie van het type `Collection<S>` accepteert waarbij `S` een subtype is van `E`. Wanneer is `S` een subtype van `E`? We moeten 2 gevallen onderscheiden: is `E` een class of is `E` een interface?

- Als `E` een class is, dan is `S` een subtype van `E` als:
  - ♦ `S` gelijk is aan `E` (elk type is een subtype van zichzelf)
  - ♦ `S` een subclass is van `E`
  - ♦ `S` een subclass van een subclass is van `E`
  - ♦ enz.

Korter: `S` is `E`, of `S` moet direct of indirect van `E` erven.

- Als `E` een interface is, dan is `S` een subtype van `E` als:
  - ♦ `S` gelijk is aan `E` (elk type is een subtype van zichzelf)
  - ♦ `S` een uitbreiding is van `E` (`interface S extends E`)
  - ♦ `S` een interface `E` of één van zijn uitbreidingen implementeert
  - ♦ `S` erft van een class die de interface `E` of één van zijn uitbreidingen implementeert.

Korter: `S` is `E`, `S` is een uitbreiding van `E`, of `S` implementeert `E`.

- 3) Een argument van het wildcard type `Collection<? super E>` accepteert een referentie van het type `Collection<T>` waarbij `T` een supertype is van `E`.

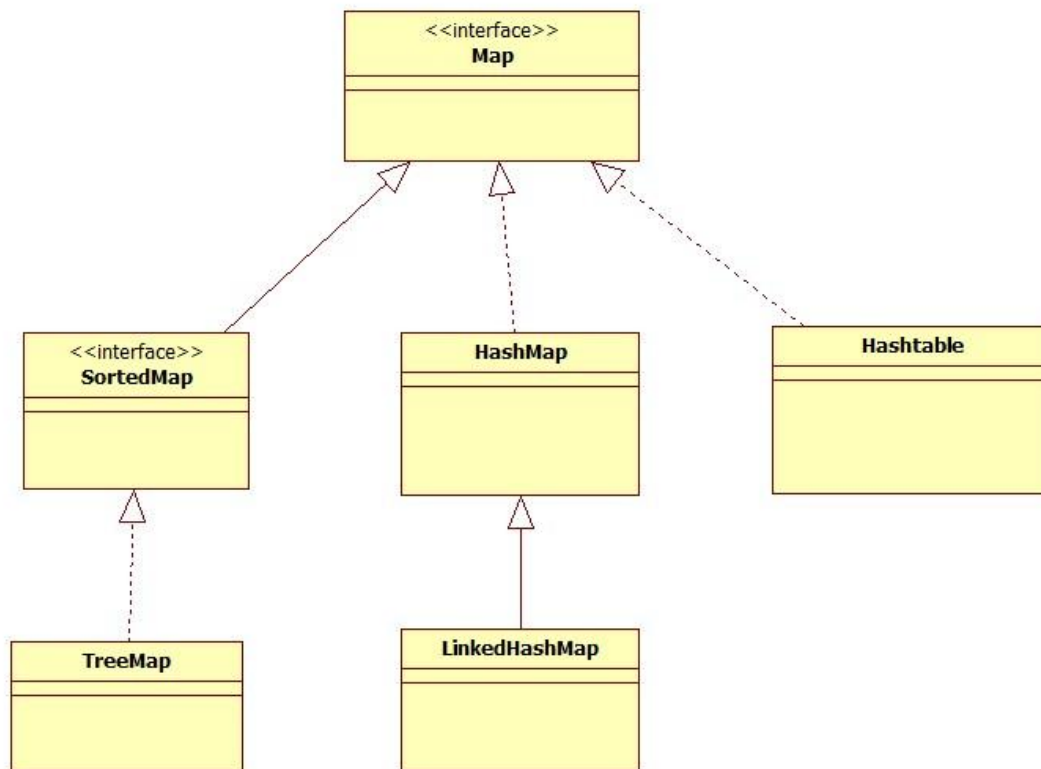
De begrippen subtype en supertype zijn spiegelbeelden van elkaar: `T` is een supertype van `E` als `E` een subtype is van `T`.

Korter: `T` is `E` zelf, of `T` is een superclass van `E`, of `T` is een interface die door `E` wordt geïmplementeerd.

## 12.6 De Map interface

Tot nu toe hebben we de hiërarchie van `Collection` besproken. Daarnaast 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.

We hernemen even het class-diagram van de `Map` interface:



De klassen 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 opgeborgen.

We hebben het volgende reeds verteld :

Map...	...bevat key-value paren. ...kan geen dubbele keys bevatten.
--------	-----------------------------------------------------------------

Bij de ArrayList hebben we gezien dat je een element kan 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 dingen dan een geheel getal. Dat brengt ons bij de Map.

### 12.6.1 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 meestal aangeduid worden met de woorden **key (sleutel)** en **value (waarde)**.

We geven 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 hier het nummer van de cursus en de value, het tweede deel van het paar, is hier de cursusnaam.

In de praktijk komt het vaak voor dat de sleutel 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 sleutel hoort precies één waarde. Dit is een typisch voorbeeld van een map!

### 12.6.2 Interface Map

Eigenlijk spreken we niet 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 sleutels bevatten en een sleutel verwijst slechts naar één waarde.

We sommen slechts enkele methods op, doch verwijzen naar de documentatie voor een volledig overzicht van alle methods:

Method	Returnwaarde	Toelichting
<code>put (K, V)</code>	<code>V</code>	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>	<code>V</code>	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>	<code>V</code>	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>	<code>boolean</code>	Geeft <i>true</i> terug wanneer de map een mapping bevat van de gevraagde sleutel.
<code>containsValue(Object value)</code>	<code>boolean</code>	Geeft <i>true</i> terug wanneer de map één of meerdere keys mapt naar de gevraagde value.
<code>size()</code>	<code>int</code>	Geeft het aantal mappings van de map terug.
<code>keySet()</code>	<code>Set&lt;K&gt;</code>	Geeft een set terug van alle keys in de map.
<code>values()</code>	<code>Collection&lt;V&gt;</code>	Geeft een collectie terug van alle waarden in de map.
<code>entrySet()</code>	<code>Set&lt;Map&lt;K,V&gt;&gt;</code>	Geeft een set terug van de mappings in de map.
<code>clear()</code>	<code>void</code>	Alle mappings worden verwijderd uit de map.

De belangrijkste implementaties van de Map zijn de `HashMap`, de `LinkedHashMap` en de `TreeMap`.

### 12.6.3 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 hebben we reeds eerder besproken bij de `HashSet`.

#### 12.6.3.1 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. Een Collection heeft een iterator, en via deze iterator kun je over de views, en dus over de elementen van een HashMap, itereren.

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.

### 12.6.3.2 Een voorbeeld van een HashMap

```
import java.util.*;

public class BasisOefHashMap {
    public static void main(String[] args) {
        Map<String,String> landen = new HashMap<String,String>();           (1)

        landen.put("B","Belgie");                                           (2)
        landen.put("NL","Nederland");
        landen.put("F","Frankrijk");
        landen.put("D","Duitsland");
        landen.put("L","Luxemburg");
        landen.put(null,null);                                             (3)

        String eenLand = landen.get("F");                                   (4)
        System.out.println("Land met code F: " + eenLand);

        String vorigLand= landen.put("F", "Finland");                      (5)
        System.out.println("Vorig land met code F: " + vorigLand);
        eenLand = landen.get("F");
        System.out.println("Land met code F: " + eenLand);

        System.out.println("*** View van de Keys ***");
        Iterator<String> it = landen.keySet().iterator();                   (6)
        while (it.hasNext()) {
            String landcode = it.next();
            eenLand = landen.get(landcode);
            System.out.println(landcode + " heeft als naam: " +
                               eenLand);
        }

        System.out.println("*** View van de Values ***");
        it = landen.values().iterator();                                    (7)
        while (it.hasNext()) {
            String land = it.next();
            System.out.println(land);
        }

        System.out.println("*** View van de Key-Value-paren ***");
        Iterator it2 = landen.entrySet().iterator();                       (8)
        while (it2.hasNext()) {
            System.out.println(it2.next());
        }
    }
}
```



- (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) Zelfs `null` voor de key en voor de value zijn toegestaan.
- (4) 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.
- (5) 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!
- (6) Een eerste view wordt gemaakt. Via de method `keySet()` bekom je een set van alle keys aanwezig in de map. Van deze set kan je een iterator vragen om over de set te itereren. Zo wordt per aanwezige key in de set 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. Van deze Collection kan je een iterator opvragen zodat je over alle aanwezige values kan itereren om ze vervolgens te tonen.
- (8) Tot slot wordt ook de derde view gemaakt. Via de method `entrySet()` bekom je een set van alle entry-objecten in de map. Via een iterator kan je vervolgens weer itereren over deze set om alle entry-objecten te tonen.

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 ***
null heeft als naam: null
D heeft als naam: Duitsland
F heeft als naam: Finland
NL heeft als naam: Nederland
B heeft als naam: België
L heeft als naam: Luxemburg
*** View van de Values ***
null
Duitsland
```

```
Finland
Nederland
Belgie
Luxemburg
*** View van de Key-Value-paren ***
null=null
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 gezegd, wordt de volgorde bepaald door de hashcode van de key: het is immers een hashmap.



Indien de **sleutel** van een **map** een **getal** is, kan je de primitieve data types (int, long, ...) niet gebruiken.

Je gebruikt dan de bijbehorende wrapper classes (**Integer**, **Long**, ...).

#### 12.6.4 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 entry's 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 entry's aan de HashMap toevoegt. Dit wordt **insertion-order** genoemd.

De LinkedHashMap heeft een extra constructor en enkele extra methods. Bij die extra constructor kan je via een derde argument kiezen voor **access-order** in plaats van insertion-order: de entry's in de linked list krijgen aanvankelijk de volgorde waarin ze in de lijst zijn gestopt, maar elke entry die je met `get()` opvraagt komt daarna achteraan in de lijst te staan. We verwijzen naar de documentatie voor meer uitleg aangaande deze extra's.

##### 12.6.4.1 Voorbeeld van een LinkedHashMap

Wanneer je in het vorige voorbeeld geen HashMap creëert, maar een LinkedHashMap, zal je merken dat dit zonder meer werkt:

```
import java.util.*;

public class BasisOefHashMap {
    public static void main(String[] args) {
        Map<String,String> landen =
            new LinkedHashMap<String,String>();
        // (1)

        landen.put("B", "Belgie");
```



...  
...

- (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 behouden is gebleven en dus de volgorde van toevoegen is:

```
Land met code F: Frankrijk
Vorig land met code F: Frankrijk
Land met code F: Finland
*** View van de Keys ***
B heeft als naam: Belgie
NL heeft als naam: Nederland
F heeft als naam: Finland
D heeft als naam: Duitsland
L heeft als naam: Luxemburg
null heeft als naam: null
*** View van de Values ***
Belgie
Nederland
Finland
Duitsland
Luxemburg
null
*** View van de Key-Value-paren ***
B=Belgie
NL=Nederland
F=Finland
D=Duitsland
L=Luxemburg
null=null
```

### 12.6.5 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 bespreken 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> .

Voor een volledig overzicht verwijzen we naar de documentatie.

Een voorbeeld: we gebruiken hetzelfde voorbeeld dat we reeds bij een `HashMap` en `LinkedHashMap` gebruikt hebben:

```
import java.util.*;

public class BasisOefTreeMap {
    public static void main(String[] args) {
        Map<String,String> landen = new TreeMap<String,String>();           (1)

        landen.put("B","Belgie");                                           (2)
        landen.put("NL","Nederland");
        landen.put("F","Frankrijk");
        landen.put("D","Duitsland");
        landen.put("L","Luxemburg");
        //landen.put(null,null);                                           (3)

        String eenLand = landen.get("F");                                   (4)
        System.out.println("Land met code F: " + eenLand);

        String vorigLand= landen.put("F", "Finland");                      (5)
        System.out.println("Vorig land met code F: " + vorigLand);
        eenLand = landen.get("F");
        System.out.println("Land met code F: " + eenLand);

        System.out.println("**** View van de Keys ****");
        Iterator<String> it = landen.keySet().iterator();                   (6)
        while (it.hasNext()) {
```



```

String landcode = it.next();
eenLand = landen.get(landcode);
System.out.println(landcode + " heeft als naam: " +
                    eenLand);
}

System.out.println("*** View van de Values ***");
it = landen.values().iterator();
while (it.hasNext()) {
    String land = it.next();
    System.out.println(land);
}

System.out.println("*** View van de Key-Value-paren ***");
Iterator it2 = landen.entrySet().iterator();
while (it2.hasNext()) {
    System.out.println(it2.next());
}
}
}

```

- (1) Er wordt deze keer een map gemaakt van het type `TreeMap`.
- (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) *null* voor de key is **niet** toegestaan. Daarom wordt de regel in commentaar gezet.
- (4) 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.
- (5) 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!
- (6) Een eerste view wordt gemaakt. Via de method `keySet()` bekom je een set van alle keys aanwezig in de map. Van deze set kan je een iterator vragen om over de set te itereren. Zo wordt per aanwezige key in de set 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. Van deze Collection kan je een iterator opvragen zodat je over alle aanwezige values kan itereren om ze vervolgens te tonen.
- (8) Tot slot wordt ook de derde view gemaakt. Via de method `entrySet()` bekom je een set van alle entry-objecten in de map. Via een iterator kan je vervolgens weer itereren over deze set om vervolgens alle entry-objecten te tonen.

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 ***
B heeft als naam: Belgie
D heeft als naam: Duitsland
F heeft als naam: Finland
L heeft als naam: Luxemburg
NL heeft als naam: 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
```

Je merkt wellicht op dat de volgorde anders is dan deze van het toevoegen aan de map. Het is een sorteervolgorde volgens de key. Dit is een `String`. Het is de `compareTo()` method van de class `String` die de volgorde bepaalt: oplopend en alfabetisch. Zelf hebben we hiervoor niets moeten doen. Dit is reeds geregeld in de class `String` zelf.

Tot hier zijn dit methods die we reeds kennen van de `HashMap` en de `LinkedHashMap`. We gaan de code uitbreiden met enkele extra methods. Maar alvorens we die kunnen gebruiken, zullen we de declaratie van de variabele `landen` moeten aanpassen naar:

```
TreeMap<String,String> landen = new TreeMap<String,String>();
```

in plaats van:

```
Map<String,String> landen = new TreeMap<String,String>();
```

omdat we specifieke methods gaan gebruiken van de `treemap`!

```
...
System.out.println("*** View van de Key-Value-paren ***");
Iterator it2 = landen.entrySet().iterator();
while (it2.hasNext()) {
    System.out.println(it2.next());
}

// Extra methods
System.out.println("*** Extra methods ***");
```



```

System.out.println("Eerste key: " + landen.firstKey());      (1)
System.out.println("Laatste key: " + landen.lastKey());      (2)

SortedMap<String, String> landenSub = landen.subMap("D", "M"); (3)
System.out.println("*** View van de Key-Value-paren van de
    submap D-M ***");
Iterator itSub = landenSub.entrySet().iterator();            (4)
while (itSub.hasNext()) {
    System.out.println(itSub.next());
}
}
}

```

- (1) De method `firstKey()` geeft ons de eerste of laagste key van de treemap.
- (2) De method `lastKey()` geeft ons de laatste of hoogste key van de treemap.
- (3) Vervolgens creëren we een nieuwe sortedmap via de method `subMap (keyVanaf, keyTot )`. Deze haalt een gedeelte uit de `TreeMap`, nl. alle entry-objecten waarvan de key groter of gelijk is aan `keyVanaf` en kleiner dan `keyTot`. In dit voorbeeld zal een nieuwe sortedmap gecreëerd worden met alle key-value-paren waarvan de key gaat vanaf "D" tot "M".
- (4) Vervolgens wordt van deze submap de entryset opgevraagd. M.b.v. de iterator van deze set wordt over de set geïtereerd om vervolgens elk entry-object te tonen.

De uitvoer van dit extra stukje code is :

```

*** Extra methods ***
Eerste key: B
Laatste key: NL
*** View van de Key-Value-paren van de submap D-M ***
D=Duitsland
F=Finland
L=Luxemburg

```

#### 12.6.5.1 *Belang van Comparable-compareTo()*

We gaan het voorbeeld uitbreiden en we gebruiken nog eens onze eigen class `Cursus`. Deze class implementeert de interface `Comparable`. Dat is een vereiste voor het gebruik van de class als key in een `TreeMap`.

We hebben bij de uitleg van `TreeSet` eerder reeds 3 mogelijkheden gezien van de `compareTo()` in de class `Cursus`: nl. een sortering op `cursusNr`, één op de `cursusNaam` en één op `cursusPrijs`. Deze laatste sortering op `cursusPrijs` is normaal gezien nog de versie die het laatst bewaard is geworden. Met deze sortering gaan we de class `Cursus` gebruiken.

We tonen nog even de code van de `compareTo()` in class `Cursus`:

```

public class Cursus implements Comparable<Cursus>
{
    ...
    @Override

```



```
public int compareTo(Cursus c)
{
    // *** sorteren op cursusnr *****
    //   if (cursusNr < c.getCursusNr() ) return -1;
    //   else
    //       if (cursusNr > c.getCursusNr() ) return 1;
    //       else return 0;
    //
    // *** sorteren op cursusNaam *****
    //   return cursusNaam.compareTo(c.getCursusNaam());
    //
    // *** sorteren op cursusPrijs *****
    //   if ( this.equals(c) ) return 0;
    //   else {
    //       return prijs == c.getprijs() ? -1 : prijs - c.getprijs();
    //   }
}
```

Voeg volgend code-fragment toe aan het voorbeeld van de TreeMap:

```
...
//**** Eigen class Cursus gebruiken om volgorde van de key
//      aan te tonen: compareTo() !!
Map<Cursus,String> cursussen=new TreeMap<Cursus,String>();           (1)

cursussen.put(new Cursus(5, "Word", 100),                          (2)
              "Je leert omgaan met een tekstverwerker");
cursussen.put(new Cursus(3, "Excel", 110),
              "Je leert omgaan met een spreadsheetprogramma");
cursussen.put(new Cursus(1, "Windows", 90),
              "Je leert omgaan met windows");
cursussen.put(new Cursus(4, "Access", 120),
              "Je leert omgaan met een databasetoepassing");
cursussen.put(new Cursus(2, "Powerpoint", 80),
              "Je leert omgaan met een presentatieprogramma");
cursussen.put(new Cursus(6, "PhotoShop", 100),
              "Je leert omgaan met een fotobewerkingsprogramma");
System.out.println("*** View van de Cursussen volgens Keys ***");
Iterator<Cursus> itCursus = cursussen.keySet().iterator();           (3)
while (itCursus.hasNext()) {
    Cursus eenCursus = itCursus.next();
    String infoCursus = cursussen.get(eenCursus);                   (4)
    System.out.println(eenCursus + "   heeft als info: " + infoCursus);
}
}
```

- (1) Er wordt een map gemaakt van het type TreeMap. De key is van het type Cursus en de value is van het type String.  
Bedoeling is om een aantal cursussen te bewaren in de map met als key de cursusgegevens en voor de value een extra toelichting van deze cursus.
- (2) Vervolgens worden key-value-paren toegevoegd aan de map: een cursusobject voor de key en een String-object voor de value.
- (3) We willen dan vervolgens een view volgens de keys: we willen nl. laten zien dat de volgorde bepaald wordt door de `compareTo()` method van het type-object van de key. Hier is dat de `compareTo()` method van de class Cursus: de prijs



van de cursussen wordt met elkaar vergeleken en daarom zullen de cursussen getoond worden in volgorde van de prijs.

Met de method `keySet()` verkrijgen we een set van alle keys. Vervolgens vragen we van deze set de iterator op zodanig dat we kunnen itereren over alle elementen (keys) van de set.

- (4) M.b.v. de method `get()` zoeken we elke key op in de map en zo bekomen we de bijbehorende value van de key. Zo kunnen we dan het geheel tonen: de cursus en de bijbehorende info.

De uitvoer van dit extra stukje code is :

```
** View van de Cursussen volgens Keys **
2 Powerpoint      80      heeft als info: Je leert omgaan met een
presentatieprogramma
1 Windows         90      heeft als info: Je leert omgaan met windows
6 PhotoShop       100     heeft als info: Je leert omgaan met een
fotobewerkingsprogramma
5 Word            100     heeft als info: Je leert omgaan met een
tekstverwerker
3 Excel           110     heeft als info: Je leert omgaan met een
spreadsheetprogramma
4 Access          120     heeft als info: Je leert omgaan met een
databasetoepassing
```

De cursussen worden getoond op volgorde van de prijs, zoals bepaald in de `compareTo()` method van de class `Cursus`.

We breiden het programma nog verder uit:

Wijzig eerst : `Map<Cursus,String> cursussen = new TreeMap<Cursus,String>();`  
naar **`TreeMap<Cursus,String> cursussen = new TreeMap<Cursus,String>();`**  
zodat de `TreeMap`-specifieke methods uitgevoerd kunnen worden.

Nu volgt de uitbreiding:

```
...
System.out.println("** View van de Cursussen volgens Keys **");
Iterator<Cursus> itCursus = cursussen.keySet().iterator();
while (itCursus.hasNext()) {
    Cursus eenCursus = itCursus.next();
    String infoCursus = cursussen.get(eenCursus);
    System.out.println(eenCursus + "      heeft als info: " +
                        infoCursus);
}

// Extra methods
System.out.println("*** Extra methods Cursus ***");
System.out.println("Eerste key: " + cursussen.firstKey());           (1)
System.out.println("Laatste key: " + cursussen.lastKey());           (2)

SortedMap<Cursus, String> cursussenSub =
    cursussen.subMap(new Cursus(0,"", 90), new Cursus(0,"",120) );   (3)

System.out.print("*** View van de Key-Value-paren van de submap");
```

```
System.out.println(" met prijs tss 90-120 ***");
itSub = cursussenSub.entrySet().iterator();
while (itSub.hasNext()) {
    System.out.println(itSub.next());
}
}
```

- (1) De method `firstKey()` geeft ons de eerste of laagste key van de treemap. Hier wordt de “eerste key” bepaald door de prijs: m.a.w. dus de laagste prijs van alle cursussen uit de map.
- (2) De method `lastKey()` geeft ons de laatste of hoogste key van de treemap. Hier dus de cursus met de hoogste prijs.
- (3) Vervolgens creëren we een nieuwe sortedmap via de method `subMap (keyVanaf, keyTot )`: deze haalt een gedeelte uit de `TreeMap`, nl. alle entry-objecten waarvan de key groter of gelijk is aan `keyVanaf` en kleiner is dan `keyTot`. De key is hier van het type `Cursus`, vandaar dat er 2 nieuwe cursusobjecten aangemaakt worden waarbij de cursusprijs belangrijk is! In dit voorbeeld zal een nieuwe sortedmap gecreëerd worden met alle key-value-paren waarvan de cursusprijs gaat vanaf 90 tot 120.
- (4) Vervolgens wordt van deze submap de entryset opgevraagd. M.b.v. de iterator van deze set wordt over de set geïtereerd om vervolgens elk entry-object te tonen.

De uitvoer van dit extra stukje code is :

```
*** Extra methods Cursus ***
Eerste key: 2  Powerpoint      80
Laatste key: 4  Access        120
*** View van de Key-Value-paren van de submap met prijs tss 90-120 ***
1 Windows      90=Je leert omgaan met windows
6 PhotoShop   100=Je leert omgaan met een fotobewerkingsprogramma
5 Word        100=Je leert omgaan met een tekstverwerker
3 Excel       110=Je leert omgaan met een spreadsheetprogramma
```

## 12.6.6 Oefeningen



Maak oefeningen 28 en 29 uit de oefenmap.



## 12.7 Sorted collection implementations

We hebben reeds collection implementaties besproken die hun elementen sorteren:

- `TreeSet`: implementeert de interface `SortedSet`
- `TreeMap`: implementeert de interface `SortedMap`

Voorwaarde voor het gebruik van een `TreeSet` en/of een `TreeMap` is dat de objecten die bewaard worden in een `TreeSet` of als key in een `TreeMap` de interface `Comparable` dienen te implementeren. Het is dan de `compareTo()` method die de volgorde bepaalt van de sortering: de zogenoemde natural ordening. Doch nadeel is dat er in deze method slechts één volgorde bepaald kan worden. Denk aan het voorbeeld met de class `Cursus`.

Vaak zal deze ordening heel geschikt zijn. Doch af en toe zal het voorkomen dat je dezelfde instanties ook volgens een ander criterium wilt ordenen, één die afwijkt van de natural ordening. Dat is mogelijk met de **Comparator**.

Je kan dus de elementen in een collection sorteren volgens een aparte comparator class. Het gebruik van een comparator verhoogt daardoor het aantal sorteermethoden gevoelig.

### 12.7.1 Interface Comparator <T>

De parameter `T` is het type object dat vergeleken kan worden met deze comparator. Deze interface beschikt over de method `int compare(T obj1, T obj2)`. In deze method ga je de vergelijking uitschrijven van de 2 objecten: je bepaalt wanneer `obj1` kleiner, gelijk of groter is dan `obj2`. Eigenlijk is dit niet nieuw, zo deden we dat reeds bij de `compareTo()` method van de interface `Comparable`. De method geeft een `int` terug:

- een negatieve integer: wanneer `obj1` kleiner is dan `obj2`
- nul: wanneer `obj1 = obj2`
- een positieve integer: wanneer `obj1` groter is dan `obj2`

Om deze sortering dan te gebruiken in je collection, maak je eerst een class aan die de `Comparator` interface implementeert en je schrijft de `compare()` method uit. Vervolgens maak je een instantie aan van deze class. We noemen deze instantie een **functieobject**.

Zo'n comparator-functieobject kan je aan een `TreeSet` of `TreeMap` doorgeven via één van zijn constructors.

Een voorbeeld van een comparator-class:

```
import java.util.Comparator; (1)

public class OmgekeerdAlfabetischComparator implements Comparator<String> (2)
{
    public int compare(String s1, String s2) (3)
    {
        if (s1==null || s2==null)
            throw new NullPointerException();
        else
        {
```

```
        if (s1.compareTo(s2) == 0 )
            return 0 ;
        else
            if (s1.compareTo(s2) <0 )
                return 1;
            else
                return -1;
    }
}
```

(4)

- (1) De juiste import dient te gebeuren om te beschikken over de interface `Comparator`.
- (2) De class implementeert de interface `Comparator`. Dat wil zeggen dat de method `compare()` overriden moet worden.
- (3) Hier gaan we 2 Strings met elkaar vergelijken en bepalen wanneer `s1` kleiner is dan `s2`, gelijk is aan `s2` of groter is dan `s2`.
- (4) We willen een alfabetisch dalende volgorde bekomen. We gebruiken de `compareTo()` van de class `String` en draaien de volgorde om door de return-waardes "om te draaien". We zouden de `compare()` method korter kunnen schrijven als volgt :

```
public int compare(String s1, String s2) {
    return -s1.compareTo(s2);
}
```

Vervolgens gaan we deze comparator gebruiken bij een `TreeSet`:

```
import java.util.*;
public class BasisOefComparator {

    public static void main(String[] args) {

        System.out.print("TreeSet in natural ordening");
        Set<String> ts = new TreeSet<String> ();
        vul(ts);
        print(ts);

        System.out.print("TreeSet in volgorde van comparator-class:
            omgekeerd alfabetisch");
        Set<String> ts2=new TreeSet<String> (new OmgekeerdAlfabetischComparator());
        vul(ts2);
        print(ts2);
    }

    private static void vul(Set<String> s) {
        s.add("even");
        s.add("citroen");
        s.add("fiets");
        s.add("boom");
        s.add("aap");
    }
}
```

(1)

(2)



```
s.add("dak");    }

private static void print(Set<String> s) {
    System.out.println();

    for (String woord : s)
    {   System.out.println(woord);
    }
    System.out.println();
}
}
```

- (1) Een blanco TreeSet van Strings wordt gecreëerd. De natural ordening zal de volgorde zijn waarop de elementen gesorteerd worden. Zoals reeds eerder gezien gebruiken we 2 methods: één om de TreeSet op te vullen en één om de TreeSet te tonen.
- (2) Een tweede TreeSet van Strings wordt gecreëerd. Bij de constructor wordt een instantie van een comparator-class meegegeven. Het is een anoniem functieobject. Nu zal niet de natural ordening gehanteerd worden, maar de sortering zoals die beschreven is in de comparator-class. Dat is een omgekeerd alfabetische volgorde.

Voer dit main-programma eens uit en je bekomt volgende uitvoer:

```
TreeSet in natural ordening
aap
boom
citroen
dak
even
fiets
```

```
TreeSet in volgorde van comparator-class: omgekeerd alfabetisch
fiets
even
dak
citroen
boom
aap
```

We breiden ons main-programma uit met het volgend stukje code:

```
public static void main(String[] args)
{
    ...
    Map<String,String> landen = new TreeMap<String,String>(
        new OmgekeerdAlfabetischComparator() );           (1)

    landen.put("B", "Belgie");
    landen.put("NL", "Nederland");
    landen.put("F", "Frankrijk");
    landen.put("D", "Duitsland");
```

```
landen.put("L", "Luxemburg");

System.out.println("*** View van de Key-Value-paren ***");
Iterator itLanden = landen.entrySet().iterator();
while (itLanden.hasNext()) {
    System.out.println(itLanden.next());
}
...
}
```

- (1) Een nieuwe TreeMap wordt gecreëerd. Bij de constructor wordt een instantie van een comparator-class meegegeven. Het is een anoniem functieobject. Nu zal niet de natural ordening van de keys gehanteerd worden, maar de sortering zoals die beschreven is in de comparator-class, dus omgekeerd alfabetisch.

Voer dit main-programma opnieuw uit. De uitvoer van de toegevoegde code is:

```
*** View van de Key-Value-paren ***
NL=Nederland
L=Luxemburg
F=Frankrijk
D=Duitsland
B=Belgie
```

Nu willen we opnieuw een TreeMap aanmaken met cursusobjecten voor de value, die we wensen te tonen in dalende volgorde van de prijs. Daarvoor maken we een nieuwe Comparator-class aan:

```
import java.util.Comparator;

public class DalendePrijsComparator implements Comparator<Cursus>
{
    public int compare(Cursus c1, Cursus c2)                (1)
    {
        if (c1.compareTo(c2) == 0 )                        (2)
            return 0 ;
        else
            if (c1.getPrijs() < c2.getPrijs() )
                return 1;
            else
                return -1;
    }
}
```

- (1) Hier gaan we 2 Cursus-objecten vergelijken met elkaar en bepalen wanneer c1 kleiner is dan c2, gelijk is aan c2 of groter is dan c2.
- (2) We willen een dalende volgorde van de prijs bekomen. De method moet consistent blijven met de `equals()`. Verder vergelijken we de prijzen en zorgen we ervoor dat de hoogste prijs bovenaan komt te staan.



Nu gaan we deze comparator class gebruiken in ons main programma. Breid dit main-programma uit met het volgend stukje code:

```
public static void main(String[] args)
{
    ...
    Map<Cursus,String> cursussen = new TreeMap<Cursus,String>(
        new DalendePrijsComparator() );           (1)

    cursussen.put(new Cursus(5, "Word", 100),
        "Je leert omgaan met een tekstverwerker");
    cursussen.put(new Cursus(3, "Excel", 110),
        "Je leert omgaan met een spreadsheetprogramma");
    cursussen.put(new Cursus(1, "Windows", 90),
        "Je leert omgaan met windows");
    cursussen.put(new Cursus(4, "Access", 120),
        "Je leert omgaan met een databasetoepassing");
    cursussen.put(new Cursus(2, "Powerpoint", 80),
        "Je leert omgaan met een presentatieprogramma");
    cursussen.put(new Cursus(6, "PhotoShop", 100),
        "Je leert omgaan met een fotobewerkingsprogramma");

    System.out.println("*** View van de Key-Value-paren ***");
    Iterator itCursus = cursussen.entrySet().iterator();
    while (itCursus.hasNext()) {
        System.out.println(itCursus.next());
    }
}
```

- (1) Een nieuwe TreeMap wordt gecreëerd. Bij de constructor wordt een instantie van een comparator-class meegegeven. Het is een anoniem functieobject. Nu zal niet de natural ordening van de keys gehanteerd worden, maar de sortering voor de keys zoals die beschreven is in de comparator-class. Dat is een dalende volgorde van de prijs.

Voer dit main-programma opnieuw uit. De uitvoer van de toegevoegde code is:

```
*** View van de Key-Value-paren ***
4 Access      120=Je leert omgaan met een databasetoepassing
3 Excel       110=Je leert omgaan met een spreadsheetprogramma
6 PhotoShop   100=Je leert omgaan met een fotobewerkingsprogramma
5 Word        100=Je leert omgaan met een tekstverwerker
1 Windows     90=Je leert omgaan met windows
2 Powerpoint  80=Je leert omgaan met een presentatieprogramma
```

### 12.7.2 Comparator class als inner class

We staan eens even stil bij de 2 comparator-classes gecreëerd in de vorige paragraaf. De comparator class voor een omgekeerd alfabetische sortering is er één die vaker gebruikt kan worden, ook voor andere sorteringen op alfanumerieke velden. Daarom kan je deze class als externe class behouden.

Maar de comparator class voor een dalende volgorde van de prijs is een volgorde



typisch voor de cursussen. Deze is niet algemeen te gebruiken en kan je dan ook beter integreren in de class `Cursus`.

Integreer volgende method in de class `Cursus`, en vergeet zeker de import niet:

```
import java.util.*;

public class Cursus implements Comparable<Cursus> {
    ...
    ...
    public static Comparator<Cursus> getDalendePrijsComparator() }           (1)

    return new Comparator<Cursus>() {                                     (2)
        @Override
        public int compare(Cursus c1, Cursus c2) {                       (3)
            return c2.getPrijs() == c1.getPrijs() ? -1 : c2.getPrijs() - c1.getPrijs();
        }

    };                                                                    (4)
}
```

- (1) We schrijven een nieuwe method `getDalendePrijsComparator()`.
- (2) Deze method levert een anoniem object terug van een `Comparator` class. Deze comparator-class noemen we een anonieme inner class. Inner omdat ze hier volledig in de method wordt beschreven en anoniem omdat de beschrijving van de class onmiddellijk volgt na `new`. Met `new` wordt een object gemaakt van de class, deze heeft geen naam en is dus anoniem en heeft één method, nl. de `compare()`.
- (3) Deze method `compare()` schrijven we hier uit. We vergelijken hier 2 cursus-objecten. We willen een dalende volgorde van de prijs bekomen en de method moet consistent blijven met de `equals()`. We vergelijken dus de prijzen en zorgen ervoor dat de hoogste prijs bovenaan komt te staan.
- (4) Hier pas eindigt het return-statement. Hier schrijven we dus de punt-komma! Hier eindigt de code van de anonieme inner class.

Nu gaan we deze comparator class gebruiken in ons main-programma. Wijzig dit als volgt:

```
public static void main(String[] args)
{
    ...
    Map<Cursus,String> cursussen =
        new TreeMap<Cursus,String>(Cursus.getDalendePrijsComparator() ); (1)

    cursussen.put(new Cursus(5, "Word", 100),
        "Je leert omgaan met een tekstverwerker");
    ...
}
```

- (1) Een nieuwe `TreeMap` wordt gecreëerd. Bij de constructor wordt via de method `getDalendePrijsComparator()` een instantie van een comparator-class bekomen. Deze method levert ons een comparator-object dat ervoor zorgt dat de



cursussen bewaard worden in de map, gesorteerd volgens dalende volgorde van de prijs.

Voer dit main-programma opnieuw uit. De uitvoer zal hetzelfde zijn, nl.:

```
...
*** View van de Key-Value-paren ***
4 Access      120=Je leert omgaan met een databasetoepassing
3 Excel       110=Je leert omgaan met een spreadsheetprogramma
6 PhotoShop   100=Je leert omgaan met een fotobewerkingsprogramma
5 Word        100=Je leert omgaan met een tekstverwerker
1 Windows     90=Je leert omgaan met windows
2 Powerpoint  80=Je leert omgaan met een presentatieprogramma
```

Best is dus om goed na te denken of je de comparator-class veelvuldig kan gebruiken en dus als aparte class gaat definiëren. In het andere geval, wanneer de volgorde specifiek is en eigen is voor de betreffende class, kan je beter de comparator class intern beschrijven in die class.

Alle classes die geschreven worden, moeten tenslotte ook gedocumenteerd, getest en onderhouden worden. Zo zijn de projecten beter beheersbaar.

## 12.8 Class collections

Het Collections Framework kent een bijzondere *class* met de naam *Collections*. Opgepast: verwar dit niet met de *interface* *Collection*.

De class *Collections* bevat enkele constanten en tamelijk wat algoritmen in de vorm van generieke statische methods. Het zijn algoritmen voor uiteenlopende handelingen die met collecties te maken hebben, zoals zoeken, kopiëren, vullen, omkeren, schudden en sorteren.

### 12.8.1 Methods van de class *Collections*

We sommen slechts enkele methods op, maar verwijzen zeker naar de documentatie voor een volledig overzicht.

Method	Returnwaarde	Toelichting
<code>sort(List&lt;T&gt; list)</code>	<code>&lt;T extends comparable&lt;? super T&gt;&gt;</code>	Sorteert de elementen in <code>list</code> volgens de natuurlijke ordening
<code>reverseOrder()</code>	<code>&lt;T&gt;Comparator&lt;T&gt;</code>	Geeft een comparator terug die, toegepast op een collectie, het omgekeerde van de natural ordening bewerkstelligt.
<code>shuffle(List&lt;?&gt; list)</code>	<code>void</code>	Verplaatst de elementen randomsgewijs in de <code>list</code> , zodat ze in willekeurige volgorde komen staan.
<code>fill(List&lt;? Super T&gt; list, T obj)</code>	<code>&lt;T&gt; void</code>	Vervangt alle elementen die in <code>list</code> zitten door een referentie naar één exemplaar van object

		obj.
<code>min(Collection&lt;? extends T&gt; col)</code>	<code>&lt;T extends Object &amp; Comparable&lt;? super T&gt;&gt;</code>	Levert het kleinste element uit de collectie volgens de natural ordening.
<code>checkedCollection(Collection&lt;E&gt; c, Class&lt;E&gt; type)</code>	<code>&lt;E&gt;Collection&lt;E&gt;</code>	Levert een view van de collectie <code>c</code> die garandeert dat alle objecten die je aan de collectie toevoegt van het type zijn dat door <code>type</code> wordt aangegeven.
<code>swap(List&lt;?&gt; list, int i, int j)</code>	<code>void</code>	Verwisselt de elementen die op positie <code>i</code> en <code>j</code> in de lijst staan.
<code>rotate(List&lt;?&gt; list, int distance)</code>	<code>void</code>	Roteert de elementen van <code>list</code> over de aangegeven <code>distance</code> ; dat wil bijv. zeggen dat wanneer <code>distance==1</code> alle elementen 1 positie verder in de lijst opschuiven waarbij het laatste element vooraan in de lijst komt te staan.
<code>synchronizedCollection(Collection&lt;T&gt; c)</code>	<code>&lt;T&gt;Collection&lt;T&gt;</code>	Levert een gesynchroniseerde (thread safe) versie van de collectie <code>c</code> .
<code>unmodifiableCollection(Collection&lt;? extends T&gt; c)</code>	<code>&lt;T&gt;Collection&lt;T&gt;</code>	Levert een niet te wijzigen versie van <code>c</code> .

## 12.8.2 Toelichting bij de methods

We geven een korte uitleg bij de verschillende methods. Bedoeling is dat je onthoudt dat de class `Collections` bestaat en beschikt over diverse methods om de collection te bewerken.

- **Schudden en sorteren.** De methods `Collections.shuffle()` en `Collections.sort()` zijn elkaars omgekeerde: met `shuffle()` breng je de elementen van een lijst in willekeurige volgorde, en met `sort()` sorteert je de elementen.
- **Omgekeerd sorteren.** met de method `reverseOrder()` kun je de elementen van een collectie in de omgekeerde volgorde sorteren. Omgekeerd t.o.v. de natural ordening, of omgekeerd volgens een `Comparator`.
- **Vullen en kopiëren.** Voor het vullen en kopiëren van een collectie bestaan de methods `fill()`, `nCopies()` en `copy()`. Met `fill()` kun je in een bestaande lijst alle referenties vervangen door referenties naar hetzelfde object, met `nCopies()` maak je een nieuwe lijst met `n` referenties naar hetzelfde object en met `copy()` kun je een lijst kopiëren.



- **Synchronisatie.** Soms moet je in een programma met meerdere threads te werken. Probleem hierbij is dat een handeling niet altijd zomaar onderbroken mag worden door een thread. Dit kan je voorkomen door ervoor te zorgen dat die betreffende handelingen *ondeelbaar* worden, zodat ze dus niet onderbroken kunnen worden door een andere thread. We noemen dit synchronisatie: de handeling is dan gesynchroniseerd.  
De collecties uit het Collections Framework zijn niet gesynchroniseerd. Daarom kan het gebeuren dat in een multithreaded programma 2 verschillende threads tegelijkertijd dezelfde collectie proberen te wijzigen wat ongewenste effecten kan hebben. Een collectie die daar wel tegen beveiligd is, heet **thread safe**. Zo'n collectie krijg je als resultaat van de aanroep van één van de `synchronized()` methods. Er zijn er nl. 6, dus wel wat meer dan beschreven in het bovenstaande kader.  
We noemen dit een **synchronized wrapper**: een wrapper class dus. Wrapper classes leggen een laag om objecten van een bepaalde class.
- **Niet te wijzigen collecties.** Er zijn in het Collections Framework 6 methods die een instantie van een collection leveren die niet te wijzigen is (*unmodifiable* of *immutable* of *read-only*). Een dergelijke collectie is handig als je van tevoren weet dat de gegevens in de collectie gedurende de uitvoering van het programma niet mogen wijzigen.  
We noemen dit een **unmodifiable wrapper**, ook een wrapper class dus.
- **Checked collecties.** Dit zijn collecties waarvan at runtime gecontroleerd wordt of je er elementen van het juiste type in stopt. Checked collecties kunnen van pas komen bij het debuggen van code waarin je generieke collecties mengt met oudere Java-code (toen nog geen generieke collecties bestonden).  
Met de komst van generieke collecties is het de compiler die controleert of je de collecties met objecten van het juiste type vult.

### 12.8.3 Enkele voorbeelden

We passen enkele methods toe in onderstaand voorbeeld:

```
import java.util.*;

public class BasisOefClassCollections
{
    public static void main(String[] args)
    {
        System.out.print("TreeSet: met algoritme van collections class");
        Comparator comp = Collections.reverseOrder();           (1)
        Set<String> ts = new TreeSet<String> (comp);              (2)
        vul(ts);
        toon(ts);

        System.out.print("List: met algoritme van collections class");
        List<String> al = new ArrayList<String> ();               (3)
        vul(al);
        toon(al);
        System.out.print("List: na shuffle");
        Collections.shuffle(al);                                  (4)
        toon(al);
    }
}
```

```
private static void vul(Collection<String> s)
{
    s.add("even");
    s.add("citroen");
    s.add("fiets");
    s.add("boom");
    s.add("aap");
    s.add("dak");
}

private static void toon(Collection<String> s)
{
    System.out.println();

    for (String woord : s)
    {
        System.out.println(woord);
    }

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

- (1) De method `reverseOrder()` levert een comparator op die de elementen van een collectie in omgekeerde volgorde van de natural ordening zal tonen.
- (2) Deze comparator gebruiken we bij de constructor van de `TreeSet`.
- (3) We maken een `ArrayList` van `Strings` die we opvullen en daarna tonen. De volgorde blijft behouden.
- (4) De method `shuffle()` schudt de elementen van de `ArrayList` willekeurig door elkaar.

De uitvoer van dit main-programma is:

`TreeSet`: met algoritme van `collections` class

fiets  
even  
dak  
citroen  
boom  
aap

`List`: met algoritme van `collections` class

even  
citroen  
fiets  
boom  
aap  
dak

`List`: na `shuffle`

aap  
citroen  
even



```
dak  
fiets  
boom
```

Je merkt op dat de volgorde van de elementen van de TreeSet inderdaad een omgekeerd alfabetische volgorde is: de omgekeerde volgorde van de natural ordening van de class String.

Verder heeft de ArrayList de elementen in de volgorde zoals ze zijn toegevoegd, doch na de shuffle() is de volgorde willekeurig. Wanneer je het programma nog eens uitvoert, kan de volgorde na de shuffle() anders zijn.

Voeg volgend stukje code toe onderaan het main-programma:

```
public static void main(String[] args)  
{  
    ...  
    Collections.shuffle(al);  
    toon(al);  
  
    List ual = Collections.unmodifiableList(al); (1)  
    ual.add("test"); (2)  
    toon(ual);  
}
```

- (1) De method `unmodifiableList()` levert een list op met de elementen van de arraylist `al`, maar deze list is niet wijzigbaar.
- (2) We willen toch proberen om een element "test" toe te voegen aan deze list.

Wanneer we het main-programma uitvoeren krijgen we volgende exception:

```
Exception in thread "main" java.lang.UnsupportedOperationException
```

Inderdaad: we kunnen geen element toevoegen aan een unmodifiable list.

## 12.8.4 Oefeningen



Maak oefening 30 uit de oefenmap.

## 12.9 Tot slot...

We hebben veel besproken in dit hoofdstuk, maar het is onmogelijk om volledig te zijn. Belangrijk nog om weten is dat sets veel gebruikt worden bij JSF, Hibernate, en EJB 3.0. In elk programma van een zekere omvang heb je over het algemeen altijd wel één of andere verzameling nodig. In de praktijk zul je vaak gegevens uit een database lezen, deze in een object stoppen en vervolgens bewaren in een collection.

Samenvattend geven we nog enkele tips:



- Programmeer met interfaces, niet met implementaties:  
`List lst = new ArrayList();`, en niet  
~~`ArrayList lst = new ArrayList();`~~
- In het kader van de performantie: gebruik altijd de collectie met de minste features: dus geen TreeSet wanneer een HashSet voldoende is.
- Geef de voorkeur aan een for-each lus om te itereren over een collection i.p.v. met een Iterator. Deze lus is korter, leesbaarder en foutvrijer.
- Raadpleeg de documentatie!



## Hoofdstuk 13 Streams

In een Java-programma zal je vroeg of laat een aantal gegevens willen wegschrijven naar of ophalen uit een bestand. Dit kan door gebruik te maken van *streams*, gedefinieerd in de package *java.io*.

We onderscheiden 3 soorten streams:

- byte streams: voor eenvoudige (binaire) data types zoals bytes en integers
- character streams: om tekstbestanden te verwerken
- object streams: om objecten op te slaan en in te lezen

### 13.1 Byte streams

#### 13.1.1 Eenvoudige in- en output

Als voorbeeld van een byte stream beschouwen we een reeks lottogetallen die we naar een bestand wegschrijven en nadien ook terug inlezen.

Om de getallen weg te kunnen schrijven, gebruiken we een *outputstream*, van het type *FileOutputStream*. Via een *write()-method* kunnen we gegevens naar de outputstream wegschrijven. Deze method bestaat (via overloading) in verschillende varianten: je kan één enkele byte wegschrijven of meerdere bytes in één keer.

Met de *close()-method* sluit je de outputstream. We proberen dit uit in de volgende code:

```
import java.io.FileOutputStream;
import java.io.IOException;

public class StreamsTryout {
    public StreamsTryout() {}
    public static void main(String[] args) {

        byte[] lottocijfers = {2, 13, 14, 22, 29, 32};
        FileOutputStream deLijst = null;

        try {
            //een nieuwe file aanmaken
            deLijst = new FileOutputStream("lotto.dat");

            //alle lottocijfers wegschrijven
            for (byte lottocijfer:lottocijfers) {
                deLijst.write(lottocijfer);
            }
        } catch (IOException e) { System.out.println(e.getMessage()); }
        finally {
            //de file sluiten
            if (deLijst != null)
                try { deLijst.close(); }
                catch (IOException e) { System.out.println(e.getMessage()); }
        }
    }
}
```



In de bovenstaande code importeren we eerst de *java.io* package zodat het programma classes als *FileOutputStream* zou kennen.

In het main-block definiëren we een array van bytes die we meteen initialiseren. We maken ook een referencevariabele van het type *FileOutputStream* die we voorlopig nog laten verwijzen naar null.

We proberen eerst een file aan te maken, gebruik makend van een constructor van *FileOutputStream* die de naam van de file vraagt. Indien dit bestand reeds bestaat, wordt het gewoon overschreven. Er bestaat een andere constructor die naast de naam van de file ook een boolean-parameter bevat. Met deze parameter geef je aan of de file moet overschreven worden (*false*) of geef je aan of de nieuwe gegevens toegevoegd moeten worden aan de bestaande inhoud van het bestand (*true*). Met deze constructor zou de code er dan als volgt kunnen uitzien:

```
...
//een nieuwe file (outputfile) aanmaken
deLijst = new FileOutputStream("lotto.dat", true);
...
```

Indien het creëren van de file mislukt, dan wordt er een *FileNotFoundException* gethrowd. Deze vangen we netjes op in het catch-block.

Eens de file geopend is, kunnen we de lottocijfers één voor één wegschrijven. Dit doen we door de *write()-method* aan te roepen in een *for-lus*.

Deze code kan een heel stuk korter. De *write()-method* heeft een variant waarbij je als parameters een array en twee integers kan meegeven. In de array zitten uiteraard de lottocijfers, verder geef je mee vanaf welke plaats in de array de weg te schrijven bytes zich bevinden en hoeveel er dat zijn. De kortere code:

```
...
//alle lottocijfers wegschrijven
deLijst.write(lottocijfers, 0, lottocijfers.length);
...
```

Opmerking: je kan een volledige lijst van bytes ook in één keer wegschrijven als volgt:

```
deLijst.write(lottocijfers);
```

Opgepast: het schrijven van een array werkt enkel voor een array van bytes, niet voor een array van bijvoorbeeld integers !

Tenslotte sluiten we de file met de *close()-method* in een finally-block. Hier testen we wel eerst of de referencevariabele niet naar null verwijst.

Om na te gaan of de lottogetallen echt weggeschreven zijn naar een bestand, gaan we de getallen opnieuw inlezen en op het scherm weergeven.



We voegen onderaan volgende code toe:

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

...
//lottobestand inlezen vanaf bestand
FileInputStream deInputLijst = null;
try {

    //een nieuwe inputfile aanmaken
    deInputLijst = new FileInputStream("lotto.dat");

    //alle lottocijfers inlezen
    int getal;
    while ((getal = deInputLijst.read()) != -1) {
        System.out.println(getal);
    }
}
catch (IOException e) {
    System.out.println(e.getMessage());
}
finally {
    //de file sluiten
    try {
        if (deInputLijst != null) {
            deInputLijst.close();
        }
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

We beginnen met het aanmaken van een *FileInputStream*-object. We geven de naam van het bestand mee aan de constructor.

Vervolgens lezen we in een lus de getallen in. Dit doen we tot we aan het einde van de file komen. De *read()-method* levert het getal *-1* op als we het einde van de file bereikt hebben. Is het einde van de file niet bereikt, dan tonen we het ingelezen getal op het scherm.

Tenslotte sluiten we opnieuw het bestand.



In een byte-stream staat het woord *byte* voor een stuk data ter grootte van 8 bits en niet noodzakelijk voor het numerieke datatype *byte*.

### 13.1.2 Gebufferde in- en output

Je kan nu terecht opmerken dat het byte per byte inlezen en wegschrijven van de lottogetallen niet echt optimaal is qua snelheid. Je kan beter een veel groter aantal

getallen in een soort buffer inlezen totdat deze buffer vol is, de buffer verwerken en dan opnieuw de buffer vullen en verwerken totdat alle getallen ingelezen zijn.



Het bufferen van data is een toepassing op *filtered streams*. Filtered streams gaan de ingelezen data verwerken vooraleer ze vrij te geven. Dit “verwerken” kan bufferen zijn, zoals in deze paragraaf wordt aangetoond. In een volgende paragraaf zullen een aantal opeenvolgende databytes omgezet worden in een ander gegevenstype, zodat niet alleen bytes maar ook bijvoorbeeld floats kunnen weggeschreven worden.

Om te bufferen gebruik je een class *BufferedInputStream* of *BufferedOutputStream*, om een ander datatype te lezen of te schrijven de class *DataInputStream* of *DataOutputStream*. *BufferedInputStream* en *DataInputStream* zijn zoals gezegd toepassingen op inputfilters en zijn ook afgeleid van de class *FilterInputStream*. *BufferedOutputStream* en *DataOutputStream* zijn afgeleid van *FilterOutputStream*.

Bij wijze van voorbeeld gaan we nu een zestigtal getallen wegschrijven, gebruikmakend van een buffer. Daarna lezen we ze opnieuw in en tonen we de getallen op het scherm. We hoeven ons zelf geen zorgen te maken over hoeveel getallen er in één keer in de buffer ingelezen worden. De class *BufferedInputStream* beslist dit voor ons.

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        byte[] getallen = { 2, 13, 14, 22, 29, 32, 17, 45, 44, 54, 4, 10,
                           99, 16, 26, 55, 77, 34, 51, 2, 41, 18, 20, 94, 8, 13, 42, 53,
                           66, 71, 82, 74, 36, 23, 21, 76, 16, 18, 48, 20, 3, 24, 3, 19,
                           60, 11, 65, 91, 88, 74, 56, 23, 12, 1, 10, 32, 49, 56, 17, 21 };

        //getallen naar bestand schrijven via buffer
        FileOutputStream schrijfLijst = null;
        BufferedOutputStream schrijfBuffer = null;

        try {
            //een nieuwe file en buffer aanmaken
            schrijfLijst = new FileOutputStream("getallen.dat");
            schrijfBuffer = new BufferedOutputStream(schrijfLijst);

            //alle getallen wegschrijven
            for (byte getal:getallen) {
                schrijfBuffer.write(getal);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```
    }
    } catch (IOException ex) {
        System.out.println(ex.getMessage() );
    } finally {
        //de file sluiten
        if (schrijfBuffer != null) {
            try {
                schrijfBuffer.close();
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }
}

FileInputStream leesLijst = null;
BufferedInputStream leesBuffer = null;

//getallen inlezen vanaf bestand via buffer
try {

    //een nieuwe file en buffer aanmaken
    leesLijst = new FileInputStream("getallen.dat");
    leesBuffer = new BufferedInputStream(leesLijst);

    //alle getallen inlezen
    int getal;
    while ((getal=leesBuffer.read())!=-1)
        System.out.println(getal);
}
catch (IOException ex) {
    System.out.println(ex.getMessage() );
}
finally {
    if (leesBuffer!=null)
        try {
            leesBuffer.close();
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
}
}
```

In de code merkt u op slechts twee plaatsen een verschil: enerzijds wordt na het aanmaken van de in- of outputstream ook een *bufferedstream* aangemaakt. Deze wordt gelinkt aan de in- of outputstream. Verder in de code lees en schrijf je naar de buffer in plaats van naar de in- of outputstream. Tenslotte sluit je de buffer. Op dat moment wordt de laatste inhoud van de buffer geschreven naar het bestand. De `close()`-method mag dus zeker niet vergeten worden.

### 13.1.3 In- en output van andere datatypes

Een tweede toepassing op gefilterde streams is het gebruik van *datastreams*. Dit werkt zo: als programmeur schrijf je allerlei data types zoals bijvoorbeeld een float of een long, weg naar een datastream. Deze datastream geeft achter de schermen alles door aan een buffer, die op zijn beurt ervoor zorgt dat de float of de long, byte per byte netjes wordt weggeschreven naar een *byte stream* op schijf. Hetzelfde geldt uiteraard voor het inlezen van de gegevens.

Voor elk primair data type zijn er aparte lees- en schrijfmethode gedefinieerd. Je schrijft een long weg naar een datastream met de *writeLong()-method*, een float inlezen gebeurt met een *readFloat()-method*... De volledige lijst:

- *readBoolean()*, *writeBoolean(boolean)*
- *readByte()*, *writeByte(byte)*
- *readDouble()*, *writeDouble(double)*
- *readFloat()*, *writeFloat(float)*
- *readInt()*, *writeInt(int)*
- *readLong()*, *writeLong(long)*
- *readShort()*, *writeShort(short)*

Er is echter een klein probleem: niet alle leesmethoden kunnen nu een -1 als resultaat hebben. De oplossing bestaat erin het programma over de eof te laten gaan en een *IOException* te laten throwen. Het sluiten van de file gebeurt dan in het catch-block.

In het volgende voorbeeld schrijven we allerlei temperaturen weg, lezen die terug in en geven ze weer op scherm.

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        float[] temperaturen = { 12.76F, 13.41F, 14.44F, 13.41F, 15.77F,
                                17.50F, 15.44F, 14.04F, 10.99F, 16.26F };

        FileOutputStream tempSchrijver = null;
        BufferedOutputStream bufferSchrijver = null;
        DataOutputStream dataSchrijver = null;

        //floats naar bestand schrijven via buffer en datastream
        try {

            //een nieuwe file aanmaken
            tempSchrijver = new FileOutputStream("temp.dat");
            bufferSchrijver = new BufferedOutputStream(tempSchrijver);
            dataSchrijver = new DataOutputStream(bufferSchrijver);

            //alle temperaturen wegschrijven
            for (float temperatuur:temperaturen) {
                dataSchrijver.writeFloat(temperatuur);
            }
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```



```
    } finally {
        if (dataSchrijver != null) {
            try {
                dataSchrijver.close();
            } catch (IOException ex) {
                System.out.println(ex.getMessage());
            }
        }
    }

//temperaturen inlezen vanaf bestand via buffer en datastream
FileInputStream tempLezer = null;
BufferedInputStream bufferLezer = null;
DataInputStream dataLezer = null;
try {

    //te lezen bestand aangeven
    tempLezer = new FileInputStream("temp.dat");
    bufferLezer = new BufferedInputStream(tempLezer);
    dataLezer = new DataInputStream(bufferLezer);

    //alle temperaturen inlezen
    while (true) {
        float temperatuur = dataLezer.readFloat();
        System.out.println(temperatuur);
    }
} catch (EOFException ex) {
    // bestand wordt gesloten in finally block
    // dus hoeft hier niet te gebeuren
} catch (IOException ex) {
    System.out.println(ex.getMessage());
} finally {
    if (dataLezer != null) {
        try {
            dataLezer.close();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
}
```

Zoals je ziet wordt er na de filestream een buffer gedefinieerd en vervolgens ook een datastream. Wanneer alle getallen ingelezen zijn – een EOFException wordt geforceerd - , wordt de stream gesloten in het finally-block.

## 13.2 Character streams

Net zoals we een byte stream byte per byte of via een buffer kunnen inlezen, kunnen we een character stream teken per teken of via een buffer inlezen. In het volgende programma lezen we een stukje tekst teken per teken in, en geven het, opnieuw teken per teken, weer op scherm. Als je de code uitprobeert, mag je de naam van het tekstbestand vervangen door om het even welk tekst-, html- of javabestand; als het maar gewone, platte tekst is.

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        FileReader file = null;
        //tekstbestand teken per teken inlezen en weergeven op scherm
        try {

            //een nieuwe file aanmaken
            file = new FileReader("weerbericht.txt");

            //alle tekens inlezen tot aan eof
            int gelezenTekens;

            while ((gelezenTekens = file.read()) != -1) {
                System.out.print((char)gelezenTekens);
            }
            System.out.println();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        } finally {
            if (file != null) {
                try {
                    file.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

Om het tekstbestand in te lezen gebruiken we een *FileReader* object. Met de *read()*-method lezen we teken per teken in. Dit teken wordt ingelezen als een int. Als deze int de waarde -1 heeft, zijn we aan het einde van de file gekomen.

Willen we het ingelezen teken weergeven op scherm dan moeten we de unicode omzetten naar het bijhorende teken. Dit doen we door de int te casten naar een char.



Hetzelfde programma nu, maar deze keer via een buffer.

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        FileReader file = null;
        BufferedReader buffer=null;

        //tekst teken per teken inlezen en weergeven op scherm
        try {

            //een nieuwe file aanmaken
            file = new FileReader("weerbericht.txt");
            buffer=new BufferedReader(file);

            //alle tekens inlezen tot aan eof
            int gelezenTeken;
            while ((gelezenTeken = buffer.read()) != -1) {
                System.out.print((char) gelezenTeken);
            }
            System.out.println();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        } finally {
            if (buffer != null) {
                try {
                    buffer.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

Je merkt hier opnieuw dezelfde werkwijze als bij de buffered inputstream. We lezen regel per regel uit de buffer en houden er pas mee op als de ingelezen regel helemaal leeg is.

Schrijven naar een tekstbestand doen we met een *FileWriter* object. Ook hier kunnen we, zoals bij de *FileOutputStream*, het object aanmaken met twee verschillende constructors: eentje die enkel een stringparameter heeft, en eentje die een string- én een booleanparameter heeft. In het eerste geval wordt de tekstfile, indien die reeds bestaat, gewoon overschreven. In het tweede geval kan je met de boolean aangeven of de nieuwe gegevens moeten toegevoegd worden aan de inhoud van het bestand of dat het bestand moet overschreven worden.



De `FileWriter` heeft 3 interessante geoverloade `write()`-methods:

- `write(int)` een teken wegschrijven
- `write(char[], int, int)` een tekenarray schrijven met startpunt en aantal tekens
- `write(String, int, int)` een string schrijven met startpunt en aantal tekens

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        FileWriter file = null;

        //alle tekens van het alfabet wegschrijven
        try {

            //een nieuwe file aanmaken
            file = new FileWriter("alfabet.txt");
            for (char letter = 'A'; letter <= 'Z'; letter++)
                file.write(letter);

        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        } finally {
            if (file != null) {
                try {
                    file.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

Ook hier kan je opnieuw via een buffer werken. Je gebruikt dan een *BufferedWriter* class. De werkwijze is verder helemaal dezelfde als bij de methode zonder buffer.

Leuk is ook de method *newLine()* waarmee je een nieuwe lijn forceert. Deze method bestaat niet in het `FileWriter` object.



```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        FileWriter file = null;
        BufferedWriter buffer = null;

        //alle tekens van het alfabet wegschrijven
        try {

            //een nieuwe file aanmaken
            file = new FileWriter("alfabet.txt");
            buffer = new BufferedWriter(file);
            buffer.write("Alle letters uit het alfabet");
            buffer.newLine();
            for (char letter = 'A'; letter <= 'Z'; letter++) {
                buffer.write(letter);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            if (buffer != null) {
                try {
                    buffer.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

### 13.3 Andere file operaties

Het (sequentieel) lezen en schrijven van bestanden is slechts een klein deel van alle mogelijke bewerkingen op bestanden. Je kan ook files hernoemen en verwijderen, directory's aanmaken, checken op het bestaan van bestanden, enz.

Met het object *File* verwijst je naar een bestand of een directory (map). Een *File* aanmaken kan met de volgende drie constructors:

- `File(String)` maakt een nieuwe map
- `File(String, String)` maakt in een map een nieuw bestand
- `File(File, String)` maakt in een map, aangeduid met een *File* object, een nieuw bestand

Een overzicht van enkele methods van het *File*-object:

*getName()*: deze method retourneert de naam van het bestand of de map.

*exists()*: deze method retourneert een boolean die aangeeft of de file (of folder) al dan niet bestaat.

*renameTo(File)*: deze method retourneert een boolean die aangeeft of een hernoemoperatie gelukt is. Naar welke file er moet hernoemd worden, wordt aangegeven door een File object.

*delete()*: deze method verwijdert het bestand of de map waarnaar de file verwijst.

*mkdir()* en *createNewFile()*: als je een nieuw File object maakt, maak je enkel een verwijzing naar een file of map. Je creëert geen bestand of map op je schijf. Je doet dit wel met de methods *mkdir()* voor een map en *createNewFile()* voor een bestand.

In de volgende code vind je enkele voorbeelden van deze File methods:

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        //een allegaartje van fileoperaties

        //een nieuwe verwijzing maken naar een map en file
        File nieuweMap = new File("c:\\windows");
        File eenLeegBestand = new File(nieuweMap, "win.ini");

        //testen of de file bestaat
        System.out.println(eenLeegBestand.getName() + " exists = " +
            eenLeegBestand.exists());

        //alfabet.txt uit vorige oefening hernoemen naar alphabet.txt
        //in de hoger gelegen map
        File alfabetFile = new File(".", "alfabet.txt");
        File nieuweFile = new File("../", "alphabet.txt");
        System.out.println("hernoeming gelukt: " +
            alfabetFile.renameTo(nieuweFile));

        //de file alphabet.txt verwijderen
        System.out.println("tekstfile verwijderd: " +
            nieuweFile.delete());

        try {
            File testDir = new File("testdir");
            File testFile = new File(testDir, "testfile.txt");
            System.out.println("creatie testfile in testmap gelukt: " +
                (testDir.mkdir() && testFile.createNewFile()));
        }
        catch (IOException e) { System.out.println(e.getMessage()); }
    }
}
```

Zoals je merkt in bovenstaande code dienen niet alle fileoperaties via een try- catch block opvangen te worden.



## 13.4 Object streams

Naast primaire data types en tekst kunnen ook objecten naar een stream weggeschreven worden of van een stream ingelezen worden.

Een object moet dan wel eerst hiertoe voorbereid worden, of in javatermen *Serializable* gemaakt worden. Dit gebeurt door de class de *interface Serializable* te laten implementeren. Dit is heel eenvoudig: je plaatst gewoon `implements Serializable` na de naam van de class in de class header. Er hoeven verder geen specifieke methods gedefinieerd te worden in de class. De interface *Serializable* bevat immers geen methods, het is een zogenaamde *marker interface*. Het enige doel van `implements Serializable` is aan te duiden dat het object kan weggeschreven/gelezen worden naar/uit een stream.

### 13.4.1 Objecten wegschrijven

Objecten wegschrijven naar een stream gebeurt in drie stappen:

1. een `FileOutputStream` maken
2. een `ObjectOutputStream` maken en linken aan de `FileOutputStream`
3. object(en) wegschrijven

Een voorbeeld:

Source `Werknemer.java`:

```
import java.io.*;

public class Werknemer implements Serializable {

    private String voornaam;
    private String familienaam;
    private BigDecimal wedde;

    public Werknemer(String voornaam, String familienaam, BigDecimal wedde) {
        this.voornaam=voornaam;
        this.familienaam=familienaam;
        this.wedde=wedde;
    }

    public String getVolledigeNaam() {
        return voornaam + " " + familienaam;
    }

    public BigDecimal getWedde() {
        return wedde;
    }

}
```

**Source StreamsTryout.Java:**

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        FileOutputStream file = null;
        ObjectOutputStream obj = null;

        try {
            file = new FileOutputStream("werknemers.dat");
            obj = new ObjectOutputStream(file);
            //werknemerobjecten aanmaken
            Werknemer magazijnier = new Werknemer("Etienne", "Berquin",
                                                    new BigDecimal(1200));
            Werknemer telefoniste = new Werknemer("Larissa", "Verbeke",
                                                    new BigDecimal(1250));
            Werknemer directeur = new Werknemer("Luc", "Vanhoorebeke",
                                                    new BigDecimal(4515));

            // werknemerobjecten verzamelen in array:
            Werknemer[] werknemers = new Werknemer[]{magazijnier,
                                                        telefoniste, directeur};

            //array wegschrijven (array elementen worden automatisch ook
            //weggeschreven)
            obj.writeObject(werknemers);

        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            if (obj != null) {
                try {
                    //outputstream sluiten
                    obj.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

Behalve objecten kunnen ook primaire data types worden weggeschreven naar een `ObjectOutputStream`. Met de methods `write(int)`, `write(byte[])` en `write(byte[],int,int)` schrijf je respectievelijk een int, een array van bytes en tenslotte een array van bytes met startpunt en aantal bytes weg.

Ook voor de primaire data types bestaan methods zoals `writeBoolean(boolean)`, `writeShort(short)`, enz.

Twee methods zijn nog het vermelden waard: `writeBytes(String)` en `writeChars(String)`. De eerste method schrijft een string weg als een reeks bytes, de tweede als een reeks chars.



### 13.4.2 Objecten inlezen

Objecten inlezen gebeurt eveneens in drie stappen:

1. een `FileInputStream` maken
2. een `ObjectInputStream` maken en linken aan de `FileInputStream`
3. object(en) inlezen

In het volgende voorbeeld lezen we de objectstream *werknemers.dat* opnieuw in. Via de method `getVolledigeNaam()` tonen we de voornaam en de familienaam van de werknemer op het scherm. Met de method `getWedde()` tonen we de wedde van de werknemer.

Een mogelijke fout bij het inlezen vangen we op met een *ClassNotFoundException*.

```
import java.io.*;

public class StreamsTryout {

    public static void main(String[] args) {

        FileInputStream file = null;
        ObjectInputStream obj = null;
        try {
            //een nieuwe fileInputStream en objectInputStream maken
            file = new FileInputStream("werknemers.dat");
            obj = new ObjectInputStream(file);
            Werknemer[] werknemers = (Werknemer[]) obj.readObject();
            for (Werknemer werknemer : werknemers) {
                System.out.print(werknemer.getVolledigeNaam());
                System.out.print(':');
                System.out.println(werknemer.getWedde());
            }
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        } finally {
            //de file sluiten
            if (obj != null) {
                try {
                    obj.close();
                } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        }
    }
}
```

Behalve objecten kunnen ook primaire data types worden ingelezen. De method `read()` leest een byte in en geeft hem aan het programma als een `int`, `read(byte[], int, int)` leest een aantal bytes uit een array van bytes vanaf een startpunt. Verder zijn er de gebruikelijke `readBoolean()`, `readFloat()`, enz. Met `readLine()` lees je een `String`.

### 13.4.3 Transient variabelen

Als je een object wegschrijft naar een bestand, dan ben je niet verplicht alle data members weg te schrijven. Het volstaat in de classbeschrijving het woord *transient* vlak

voor het type van het data member te plaatsen, om het te markeren als “niet weg te schrijven”.

Als voorbeeld markeren we in de classbeschrijving van `Werknemer` de wedde van een werknemer als transient. Vervolgens schrijven we in het hoofdprogramma een aantal werknemers naar een stream, om die dan terug in te lezen en af te beelden. Je zal zien dat de wedde telkens op 0 staat. M.a.w. deze variabelen worden terug geïnitieerd op default waarden.

Een reden om één of meerdere data members transient te maken, kan zijn dat je het niet veilig vindt om deze gegevens in een bestand weg te schrijven.

#### Source `Werknemer.java`:

```
import java.io.*;

public class Werknemer implements Serializable {

    private String voornaam;
    private String familienaam;
    private transient BigDecimal wedde;

    public Werknemer(String voornaam, String familienaam, BigDecimal wedde) {
        this.voornaam = voornaam;
        this.familienaam = familienaam;
        this.wedde = wedde;
    }

    public String getVolledigeNaam() {
        return voornaam + " " + familienaam;
    }

    public BigDecimal getWedde() {
        return wedde;
    }
}
```

#### 13.4.4 `SerialVersionUID`

In de vorige paragrafen hebben we probleemloos objecten geserialiseerd. Na het wegschrijven van objecten hebben we deze meestal meteen weer ingelezen. In de praktijk kan tussen het wegschrijven en het inlezen een lange tijdspanne zitten. Is de classdefinitie nog steeds dezelfde als bij het wegschrijven? Zijn er membervariabelen bijgekomen of gewijzigd, methods verdwenen?

Een oplossing voor dit probleem is de `serialVersionUID`. Dit is een static membervariabele van het type `long` waarin we een soort versienummer van de class bewaren. Normaal worden static members niet mee geserialiseerd. Voor de `serialVersionUID` wordt er een uitzondering gemaakt.

Op het moment dat de data ingelezen wordt in een object zal het `serialVersionUID` van het ingelezen object vergeleken worden met de `serialVersionUID` van de actuele versie van de class. Is de `serialVersionUID` verschillend dan krijgen we een exception. Op die



manier wordt er bijvoorbeeld voorkomen dat we membervariabelen of – methods gaan gebruiken die niet meer in de class voorzien zijn.

Een voorbeeld: hieronder vind je een class *TeSerialiseren*.

```
public class TeSerialiseren implements Serializable{
    private static final long serialVersionUID = 1L;
    private int bewaardGetal;
    public TeSerialiseren(int getal) {
        this.bewaardGetal = getal;
    }
    public int getGetal() {
        return bewaardGetal;
    }
    public int getKwadraat() {
        return bewaardGetal * bewaardGetal;
    }
}
```

In het volgende hoofdprogramma zullen we een instance van een dergelijke class wegschrijven naar een bestand en opnieuw inlezen.

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws Exception {
        File file = new File("e:\\out.dat");
        FileOutputStream fos = new FileOutputStream(file);
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        TeSerialiseren SchrijfMeWeg = new TeSerialiseren(1);
        oos.writeObject(SchrijfMeWeg);
        oos.close();

        FileInputStream fis = new FileInputStream(file);
        ObjectInputStream ois = new ObjectInputStream(fis);

        TeSerialiseren dto = (TeSerialiseren)ois.readObject();
        System.out.println("Data:" + dto.getGetal());
        ois.close();
    }
}
```

Wanneer we dit programma uitproberen krijgen we keurig de waarde te zien van de membervariabele *bewaardGetal*.

We hebben nu een bestand, *out.dat*, waarin een instance van onze class in bewaard is.



Als we in de main het bewaren van de class weglaten en dus het object enkel inlezen dan krijgen we opnieuw de waarde van de membervariabele te zien. De class is niet veranderd en dus is er geen probleem.

We maken nu een kleine wijziging in de class: de method *getKwadraat()* wordt niet meer gebruikt en verdwijnt uit de class en er komt een boolean membervariabele *jaOfNee* bij. De *serialVersionUID* verhogen we naar 2.

```
public class TeSerialiseren implements Serializable{
    private static final long serialVersionUID = 2L;

    private boolean jaOfNee;
    private int bewaardGetal;

    public TeSerialiseren(boolean janee, int getal, double getal2) {
        this.jaOfNee = janee;
        this.bewaardGetal = getal;
    }

    public boolean getBoolean() {
        return jaOfNee;
    }

    public int getGetal() {
        return bewaardGetal;
    }

    public int getKwadraat() {
        return bewaardGetal * bewaardGetal;
    }
}
```

Start nu het hoofdprogramma op om de weggeschreven data in een class in te lezen. We krijgen nu een fout. De foutboodschap maakt ons duidelijk dat de *serialVersionUID* verschillend is.

Om deze fout op een nette manier op te vangen passen we de code als volgt aan :

```
public static void main(String[] args) throws Exception {
    File file = new File("e:\\out.dat");
    //FileOutputStream fos = new FileOutputStream(file);
    //ObjectOutputStream oos = new ObjectOutputStream(fos);

    //TeSerialiseren SchrijfMeWeg = new TeSerialiseren(1);
    //oos.writeObject(SchrijfMeWeg);
    //oos.close();
}
```



```
FileInputStream fis = null;
ObjectInputStream ois = null;
try {
    fis = new FileInputStream(file);
    ois = new ObjectInputStream(fis);

    TeSerialiseren dto = (TeSerialiseren)ois.readObject();
    System.out.println("Data:" + dto.getBooleen());
}
catch (InvalidClassException ex) {
    System.out.println(ex.getMessage());
}
finally {
    if (ois != null)
        try {
            ois.close();
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    ois.close();
}
```

De werking van de `serialVersionUID` is nu duidelijk. Maar wat als we die niet implementeren in onze classes die we willen serialiseren? Als we zelf geen `serialVersionUID` voorzien doet java dit voor ons achter de schermen. Java is hierin héél erg strict. Bij de minste verandering van bijvoorbeeld visibility van een membervariabele wordt de `serialVersionUID` gewijzigd. Door de `serialVersionUID` zelf te beheren krijg je de volledige controle zelf in handen. Wees dus consequent en wijzig de `serialVersionUID` wanneer nodig !

## 13.5 Oefeningen



Maak oefening 31 uit de oefenmap.

## Hoofdstuk 14 Multithreading

---

### 14.1 Processen en threads

#### 14.1.1 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.

#### 14.1.2 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.

### 14.2 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 prestatie.

Voorbeeld: een computer met vier processoren moet twee processen uitvoeren, die elk twee threads uitvoeren. De computer moet dus in totaal vier threads uitvoeren.

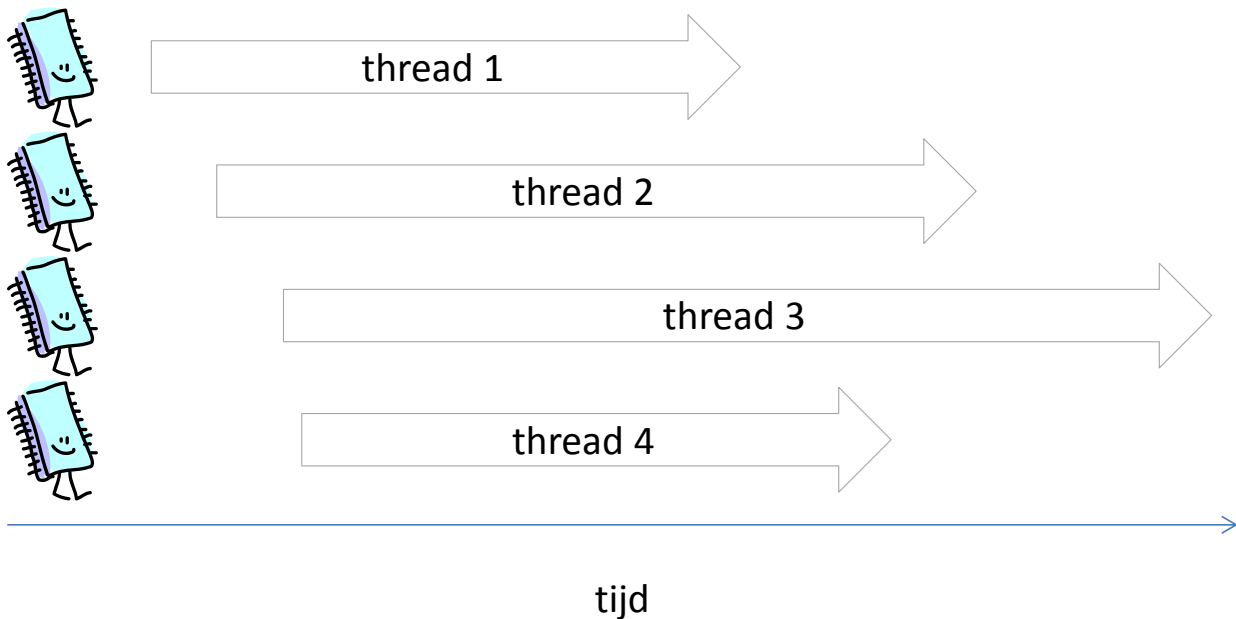
Iedere processor voert één van deze threads uit..

Opmerking:

deze threads hoeven niet op hetzelfde moment te starten, en de uitvoeringstijd hoeft niet even lang te zijn.

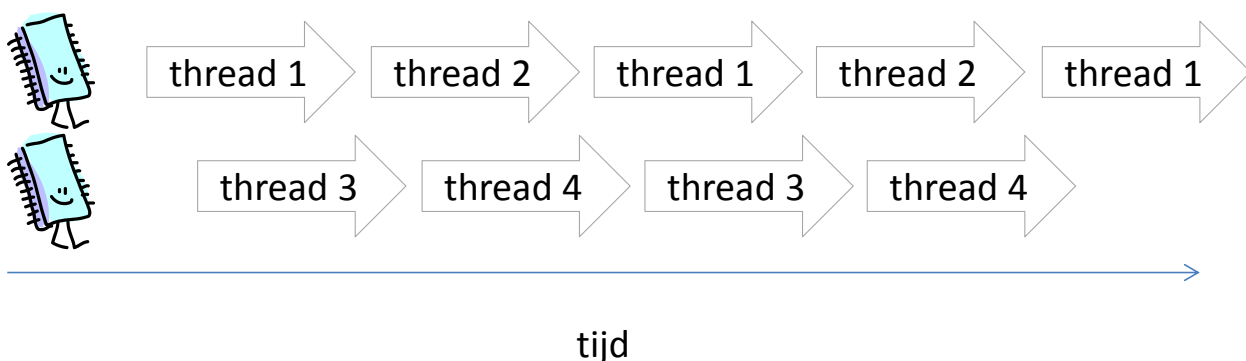


In de volgende afbeelding staan links de vier processoren, en daarnaast de thread die ze gelijktijdig uitvoeren.



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)`.

## 14.3 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.

### 14.3.1 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 = new BigDecimal(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() {
        BufferedReader reader = null;
        try {
            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);
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException ex) {
                    stream.println(ex);
                }
            }
        }
    }
}
```

Je maakt in de package be.vdab ook een class Main

```
package be.vdab;

public class Main {
    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.

### 14.3.2 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 Main

```
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

### 14.4 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 Main toont de 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 `Main` 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).

## 14.5 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 wijzigt de class Main

```
package be.vdab;

public class Main {
    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.

## 14.6 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 beslissen zichzelf stop te zetten.

Je wijzigt de applicatie. Wanneer de gebruiker op Enter drukt, stop je de applicatie. Je controleert het toetsenbord in de thread van de class Main. Wanneer de gebruiker op Enter drukt, voer je de method interrupt uit op het object thread.

Je voegt in de class Main 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.

## 14.7 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 `Main` 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.

## 14.8 Synchronized

Primitieve types (`int`, `long`, ...) en de meeste objecten zijn niet thread-safe.

Dit betekent dat je ze niet gelijktijdig met meerder 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.

### 14.8.1 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 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 wijzigt in de class Main de method main:

```
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 gebeurd: de variabele bevat 1, terwijl ze 2 zou moeten bevatten.

### 14.8.2 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 wachtten op het vrijkomen van de monitor verder werken.

Je wijzigt in de class Pannenkoek 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()
```

### 14.8.3 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 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.

## 14.9 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 meerder 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

## 14.10 Oefeningen



Maak oefeningen 32 en 33 uit de oefenmap.

## Hoofdstuk 15 Graphical User Interface

---

### 15.1 Swing versus AWT

Toen Java begin jaren 90 werd uitgebracht, was er de **AWT**-toolkit voor het bouwen van een **GUI** of Graphical User Interface.

**AWT** staat voor *Abstract Windows Toolkit*: een verzameling van grafische componenten (tekstvelden, knoppen, schuifbalken, ...) die in de praktijk werden gerealiseerd door een beroep te doen op de grafische mogelijkheden van het onderliggende operating-systeem. Deze keuze had twee consequenties:

- Om platformonafhankelijk te blijven, kon men dus enkel beroep doen op die componenten die op alle platformen bekend waren. Men werkte dus met de grootste gemene deler van de verschillende mogelijkheden.
- Door een beroep te doen op het onderliggende besturingssysteem kreeg men geen uniforme look-and-feel. Een knop in een Linux omgeving ziet er iets anders uit dan een knop in een Windows omgeving, ...

Sinds Java 2 is er een alternatief: de **JFC** (Java Foundation Classes), ook gekend als **swing**.

Swing werd gezien als een uitbreiding op standaard Java, vandaar dat steeds **javax.swing.\*** geïmporteerd wordt. Ook alle classes uit de swing-library hebben een naam die begint met een **J** (JButton is een swing-button, Button is een awt-button).

Swing heeft AWT niet verdrongen. Enkel de user-interface componenten (knop, tekstvak, radiobutton, aankruisvak, ...) hebben een swing-versie. Het basisvenster (met tekengebied) wordt nog altijd opgevraagd aan het besturingssysteem, maar de user-interface componenten worden hierop getekend.

Een aantal helper-classes uit AWT blijven bestaan (Graphics, Color, Font, FontMetrics en LayoutManagers) en ook het eventmodel van AWT wordt gebruikt.

Op dit ogenblik heeft de programmeur dus de keuze of hij AWT-componenten of swing-componenten gebruikt. Een combinatie van beide geeft problemen.

Advies i.v.m. gebruikers-componenten (knoppen, tekstvakken, scrollbars, ...): het voortbestaan van de AWT-componenten wordt niet gegarandeerd! Men gebruikt dus bij voorkeur Swing.

Voor Applets is er een bijkomend probleem: oudere browsers ondersteunen geen swing!



## 15.2 “Hello World” in GUI-versie

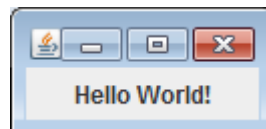
Als voorbeeld maken we een soort “Hello World”-programma in GUI-versie :

```
import javax.swing.*;           (1)
import java.awt.*;              (2)

public class HelloWorld extends JFrame {           (3)
    private JLabel label;                          (4)
    public static void main(String[] args) {
        HelloWorld frame = new HelloWorld();        (5)
        frame.createGUI();                          (6)
        frame.pack();                               (7)
        frame.setVisible(true);                     (8)
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);      (9)
        setLayout(new FlowLayout());                (10)
        label = new JLabel("Hello World! ");        (11)
        add(label);                                 (12)
    }
}
```

Met als resultaat:



Verklaring:

- (1) We gaan met swing-componenten werken, vandaar de import. Let op de **javax**.
- (2) Swing wordt gebouwd bovenop awt; awt is nodig voor o.a. FlowLayout (zie 10).
- (3) Onze eigen ‘vensterapplicatie’ erft van JFrame. JFrame is een toplevel element in swing. Het levert een standaard venster op met standaard knoppen (zoals bepaald door het besturingssysteem). Standaard is dit een venster zonder titel, zonder grootte en waarvan de eigenschap visibility op false staat.
- (4) Een globale variabele textField wordt gedeclareerd en die is van de soort **JTextField**.
- (5) We maken van onze eigen class een **object** (met de naam frame). Dit laat toe om instantievariabelen en niet-statische methods te gebruiken.
- (6) We roepen de method **createGUI()** aan op ons eigen object **frame**.
- (7) De method **pack()** maakt het JFrame juist groot genoeg, zodat alle componenten zichtbaar zijn.
- (8) We zorgen ervoor dat het venster zichtbaar wordt.



De definitie van de method `createGUI()` :

- (9) Het vensterobject dat we gemaakt hebben onder (5) komt met een aantal standaard eigenschappen (titelbalk, minimize-, maximize- en sluitknop, systeemmenu, ...) en ook een standaard gedrag. Tot het standaard gedrag behoort o.a. het reageren op de sluitknop. Klikte men op de sluitknop dan wordt het venster inderdaad gesloten maar dit betekent niet dat het programma gestopt wordt. Dit blijft actief in het geheugen (en kan eventueel gestopt worden via `<ctrl><alt><delete>`). Beter is om in het programma te voorzien wat er moet gebeuren indien het venster gesloten wordt. De optie hier is dan ook om het programma te stoppen indien het venster wordt gesloten.
- (10) Componenten worden toegevoegd aan `JFrame`. Eerst wordt een layout-manager toegekend aan `JFrame`. Er zijn meerdere layoutmanagers (zie verder), voorlopig beperken we ons tot de **FlowLayout** wat wil zeggen dat de elementen in het venster geplaatst worden in de volgorde zoals het in de code wordt uitgevoerd, te beginnen met lijn 1. Is lijn 1 vol, dan wordt er verder gegaan op lijn 2 enz. Iedere lijn is standaard bij `FlowLayout` gecentreerd !
- (11) Het label wordt aangemaakt en krijgt via de constructor een inhoud.
- (12) Het label wordt toegevoegd aan het frame.



Bovenstaande structuur, een class-definitie met een static `main()`-method waarbinnen een object wordt aangemaakt van de eigen class + een afzonderlijke method om de GUI op te bouwen is quasi een standaard structuur voor alle GUI-based applicaties.



### 15.3 Een eerste event-driven voorbeeld

In het volgende voorbeeld brengen we wat meer leven in de brouwerij :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;                                     (1)

public class Begroeting extends JFrame implements ActionListener{ (2)
    private JTextField tekstveld;
    private JLabel tekst, antwoord;
    private JButton knop;                                     (3)

    public Begroeting() {
        setTitle("Welkom");                                   (4)
    }

    public static void main (String[] args) {
        Begroeting frame = new Begroeting();
        frame.createGUI();
        frame.pack();
        frame.setVisible(true);
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout() );
        tekst = new JLabel("Geef uw naam: ");                 (5)
        tekstveld = new JTextField(20);                       (6)
        knop = new JButton("Klik mij");                        (7)
        antwoord = new JLabel();
        add(tekst);
        add(tekstveld);
        add(knop);
        add(antwoord);                                       (8)
        knop.addActionListener(this);                         (9)
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        String t = "    Welkom bij ons " + tekstveld.getText(); (10)
        antwoord.setText(t);                                  (11)
        pack();                                              (12)
    }
}
```

## Verklaring:

- (1) Er wordt met events gewerkt, dus: `import java.awt.event.*`. Voor alle swing-componenten waar er een overeenkomstige awt-component bestaat, wordt er gewerkt met het awt-eventmodel. Voor de swingcomponenten waarvoor geen awt-equivalent bestaat - o.a. JList (de keuzelijst) en JSlider, een component waarmee je de waarde kan selecteren tussen grenswaarden middels het verschuiven van een soort knop - wordt gewerkt met `import javax.swing.event.*`.
- (2) *implements ActionListener*. Events op knoppen, tekstvakken, ... worden opgevolgd door een ActionListener. Door deze interface te implementeren, maken wij van onze class **Begroeting** een ActionListener.
- (3) We gebruiken één JTextField, twee JLabels en één JButton.
- (4) Een constructor method die het venster zijn afmetingen en een titel geeft.
- (5) De JLabel wordt gemaakt. Via de constructor method geven we een waarde aan de text-property van het label.
- (6) Het tekstvak wordt gemaakt met een breedte van 20 tekens. Opgelet, dit zijn 20 gemiddelde tekens van een gemiddeld lettertype. Dit komt dus niet noodzakelijk overeen met een inputveld van exact 20 tekens.
- (7) De knop wordt gemaakt en krijgt een opschrift.
- (8) Alle elementen worden toegevoegd aan het venster. De volgorde van toevoegen bepaalt de volgorde van weergeven.
- (9) De knop wordt opgehangen aan de EventListener. Het keyword **this** wijst erop dat het het eigen object is dat moet verwittigd worden wanneer er een event optreedt. Dit is aanvaardbaar omdat wij van onze class een **ActionListener** hebben gemaakt, zie (2).

## De method actionPerformed():

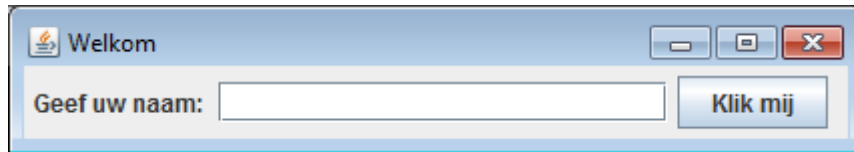
Door de implementatie van de interface zijn we verplicht een concrete realisatie te maken van de public method **actionPerformed(ActionEvent e)**. Telkens er een event optreedt, wordt deze method aangeroepen. Deze method krijgt een object van de class ActionEvent binnen. Aangezien er in onze class slechts één element is dat opgevolgd wordt door de EventListener (nl. de knop), moeten wij ons hier niet afvragen wie het event heeft veroorzaakt. Komen we in deze method, dan weten wij dat er op de knop is geklikt. In het geval er meerdere elementen zijn die een ActionEvent kunnen veroorzaken, zullen wij het **ActionEvent e** moeten ondervragen om te weten welk specifiek element nu gebruikt is.

- (10) De inhoud van het tekstveld wordt opgehaald via de method getText().
- (11) Het label *antwoord* krijgt nu een waarde.
- (12) Het frame wordt m.b.v. de method pack() weer juist groot genoeg gemaakt zodat alle componenten zichtbaar zijn.

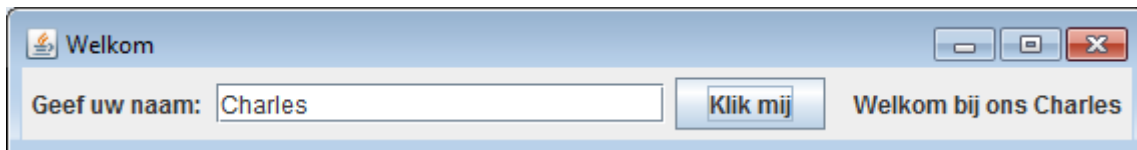


Resultaat:

Vóór er op de knop wordt geklikt:



Na het invullen van de naam en een klik op de knop:



De FlowLayout is moeilijk te gebruiken voor een vaste layout, maar heeft het voordeel dat wanneer de gebruiker het venster groter of kleiner maakt, het zich onmiddellijk aanpast en dat alle elementen steeds zichtbaar blijven. Men kan vermijden dat een gebruiker het venster (JFrame) vergroot of verkleint door de method `setResizable()` de waarde *false* te geven.

Het tekstveld wordt hier niet opgevolgd door de EventListener. Dit zou kunnen door de instructie **`tekstveld.addActionListener(this)`** te voorzien. Dit wordt ten stelligste afgeraden aangezien de `<enter>` het enige event is dat wordt herkend. Verlaat men het tekstveld via de `<tab>`-toets of via een muisklik, dan treedt er geen event op!

## 15.4 De belangrijkste swing-componenten

### 15.4.1 JFrame

Dit is het top-level venster, voorzien van alle 'standaard' onderdelen die door het besturingssysteem geleverd worden.

Standaard is het een venster met als afmetingen 0 pixels breed en 0 pixels hoog en is het onzichtbaar. Via de geëigende methods kan men dit verhelpen.

Opgelet: een venster is een object binnen een class. Sluit men het object (het venster) dan blijft de class nog altijd actief. M.a.w. het programma blijft (onzichtbaar) draaien en is dan alleen af te sluiten via `<ctrl><alt><delete>` (in windows) of via de proceslist in linux.

Men kan dit verhelpen door hetzij de windows events op te vangen, hetzij via een algemene instelling:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Dit is reeds toegepast in de eerste voorbeelden.



Er bestaat ook een class "Window" en "JWindow" die een venster tonen zonder randen, zonder titelbalk, zonder statuslijn en zonder menubalk.

### 15.4.2 JPanel

Een JPanel is een belangrijke (onzichtbare) component waarop andere onderdelen zoals knoppen en labels kunnen geplaatst worden.

Een JPanel heeft 'by default' een flow-layout.

Het JPanel zelf wordt dan toegevoegd aan het bruikbare binnendeel van het venster (JFrame). Het voordeel is dat gegroepeerde knoppen, labels, ... als een groep worden behandeld (ook binnen een flow-layout).

Op een JPanel kan ook rechtstreeks getekend worden.

Aanpassingen aan de vorige code:

```
...
knop = new JButton("Klik mij");
antwoord = new JLabel();

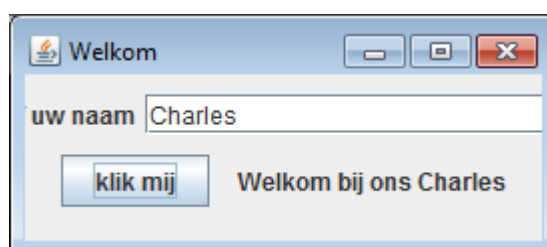
    JPanel paneel = new JPanel();
    paneel.add(tekst);
    paneel.add(tekstveld);
    add(paneel);

add(knop);
add(antwoord);
knop.addActionListener(this);
}
```

(1) Er wordt een JPanel gedeclareerd en gemaakt.

(2) De tekst en het tekstveld worden toegevoegd aan het paneel en het paneel zelf wordt toegevoegd aan het venster.

Gevolg: de tekst en het tekstveld worden niet meer gescheiden, ook niet als het venster niet breed genoeg is om alles op één lijn te tonen. Het standaardgedrag van de flow-layout dat is ingesteld voor het venster, is om de tekst en het tekstvak op twee verschillende lijnen te tonen wanneer het venster niet breed genoeg is.





Om rechtstreeks te tekenen en/of te schrijven in een JPanel zijn wel wat meer aanpassingen nodig: we voegen twee private members toe aan de class: tekenen (class JPanel) en g, een coördinatensysteem (de class Graphics). De volgende twee regels voeg je dus bovenaan toe :

```
...
private JPanel tekenen;
private Graphics g;
...
```

Bij de opbouw van de GUI wordt het tekenpaneel aangemaakt, krijgt het een afmeting (breedte, hoogte) en wordt het toegevoegd aan het venster. Voeg onderaan de createGUI()-method het volgende toe :

```
...
tekenen = new JPanel();
tekenen.setPreferredSize(new Dimension(350,50));
add(tekenen);

knop.addActionListener(this);
}
```

In de afhandeling van het klik-event wordt het coördinatenstelsel van het tekenpaneel opgehaald en kan men via **drawString()** een tekst op het paneel 'tekenen'. De parameters die moeten meegegeven worden zijn:

- De tekst om te tekenen.
- De x-coördinaat.
- De y-coördinaat.

Het coördinatenstelsel is :

- De x-coördinaat: de horizontale as, van links naar rechts, te beginnen met 0.
- De y-coördinaat: de verticale as, van boven naar onder, te beginnen met 0.

Voeg volgende coderegels toe aan de method actionPerformed() :

```
public void actionPerformed(ActionEvent e) {
    String tekst = "    Welkom bij ons " + tekstveld.getText();
    antwoord.setText( tekst);
    g = tekenen.getGraphics();
    g.drawString("Er is op de knop geklikt",30,10);
}
```

Andere tekenmethoden:

- *drawRect()* : de omtrek van een rechthoek tekenen in de voorgrondkleur.
- *fillRect()* : een rechthoek tekenen en opvullen met de voorgrondkleur.
- *drawOval()* : de omtrek van een ellips of een cirkel.
- *fillOval()* : een ellips of cirkel gevuld met de voorgrondkleur.
- *drawLine()* : een lijn.
- *drawPolygon()* : een veelhoek;
- *fillPolygon()* : een gevulde veelhoek.
- ...

(voor een totaal overzicht: zie API: de *Graphics* class).

Het tekenen in kleur gebeurt door ...:

...op niveau van de *Graphics* de voorgrondkleur in te stellen met de method *setColor*.

Bijvoorbeeld :

```
g.setColor(Color.GREEN)
```

...op niveau van de *JPanel* de achtergrondkleur in te stellen met de method *setBackground*. Bijvoorbeeld :

```
tekenen.setBackground(Color.LIGHT_GRAY)
```

### 15.4.3 JButton

Belangrijk om te weten: knop-events kunnen gevolgd worden via een ***ActionListener***. Hiervoor moet men de knop wel aanmelden bij de *EventListener*.

De *ActionListener* interface heeft maar één method: ***actionPerformed(ActionEvent)*** die als parameter het *ActionEvent* heeft (d.i. het object wat het event heeft veroorzaakt).



### Een voorbeeld :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TweeKnoppen extends JFrame implements ActionListener {
    private JButton cmdEen, cmdTwee;
    private JTextField txtResultaat;

    public TweeKnoppen() {
        setTitle("Test met twee knoppen");
    }

    public static void main (String[] args) {
        TweeKnoppen frame = new TweeKnoppen();
        frame.createGUI();
        frame.pack();
        frame.setVisible(true);
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout() );
        cmdEen = new JButton("Een");
        cmdTwee = new JButton("Twee");
        add(cmdEen);
        add(cmdTwee);
        txtResultaat = new JTextField(30);
        add(txtResultaat);
        cmdEen.addActionListener(this);
        cmdTwee.addActionListener(this);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == cmdEen)
            txtResultaat.setText("Er is op knop Een geklikt");
        else
            txtResultaat.setText("Er is op knop Twee geklikt");
        pack();
    }
}
```

- (1) In dit voorbeeld worden twee knoppen op het schermgezet: **cmdEen** en **cmdTwee**.
- (2) Beide knoppen worden geregistreerd bij de EventListener. De EventListener van een button is de ActionListener. Een klik op de button zorgt er dan voor dat de method van de ActionListener (dit is actionPerformed())wordt uitgevoerd.



- (3) Bij de afhandeling van een klik-event wordt er via de method **getSource()** getest op welke knop er geklikt werd. Deze method geeft het object terug!

Een alternatief (in dit geval) is de method **getActionCommand()** die hier de tekst zou weergeven die op de knop staat ("Een" of "Twee");

Het resultaat van een klik op knop Twee:



#### 15.4.4 JLabel

Een JLabel wordt gebruikt om tekst op het scherm te zetten of om de resultaten van een verwerking te tonen. Een JLabel kan niet opgevolgd worden door een Event-Listener.

#### 15.4.5 JList

Een JList is een keuzelijst. Afhankelijk van de settings kunnen er één of meerdere selecties gemaakt worden. Een voorbeeld :

```
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;

public class Keuzelijst extends JFrame
    implements ListSelectionListener {
    private JList lstKies;

    public Keuzelijst() {
        setTitle("Test met een Lijst");
    }

    public static void main (String[] args) {
        Keuzelijst frame = new Keuzelijst();
        frame.createGUI();
        frame.pack();
        frame.setVisible(true);
    }

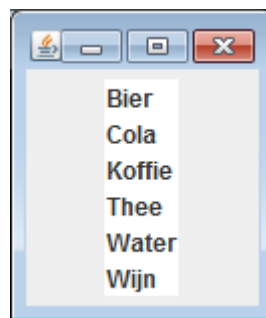
    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout() );
        String[] dranken = {"Bier", "Cola", "Koffie",
                           "Thee", "Water", "Wijn"};
        lstKies = new JList(dranken);
        add(lstKies);
        lstKies.addListSelectionListener(this);
    }
}
```



```
@Override
public void valueChanged(ListSelectionEvent e) {

    Object[] uwKeuze = lstKies.getSelectedValues();
    for(int i = 0; i < uwKeuze.length; i++)
        System.out.println(uwKeuze[i].toString());
    pack();
}
}
```

Het resultaat:



En na een meervoudige selectie kan dat volgende output geven:

```
compile-single:
run-single:
Thee
Thee
Bier
Bier
Cola
Cola
Wijn
Wijn
```

**Opmerking:** de output is niet duidelijk. Er treden bij multiselect teveel events op (keuze van het eerste item, van het volgende item, ...). Single select is daarom eenvoudiger.

Een alternatieve oplossing: de lijst niet laten opvolgen door een EventListener maar een knop plaatsen om de selectie te bevestigen.

Het instellen van de single-select mode:

```
.setSelectionMode(ListSelectionModel.SINGLE_SELECTION)
```

van 1 aaneengesloten range :

```
.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION)
```

of van 1 of meer aaneengesloten ranges :

```
.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION)
```

De laatste optie is de default optie.

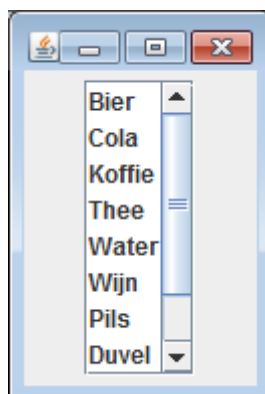
Wanneer de list een SINGLE\_SELECTION lijst is, gebeurt het ophalen van de geselecteerde waarde via de method `.getSelectedValue()`; Deze method geeft een object terug en geen array van objecten.

Een JList heeft geen schuifbalken. Wil men toch schuifbalken gebruiken, dan plaatst men de lijst in een **JScrollPane** :

```
...
setLayout(new FlowLayout() );
String[] dranken = {"Bier", "Cola", "Koffie", "Thee", "Water",
                    "Wijn", "Pils", "Duvel", "Kriek", "Geuze"}; (1)
lstKies = new JList(dranken);
JScrollPane metSchuif = new JScrollPane(lstKies); (2)
add(metSchuif); (3)
lstKies.addListSelectionListener(this);
}
```

- (1) De lijst is wat langer gemaakt.
- (2) Eerst wordt er een JScrollPane gemaakt en, via de constructor, gevuld met de keuzelijst.
- (3) Vervolgens wordt nu het ScrollPane metSchuif (+ inhoud) toegevoegd aan het venster.

Het resultaat:





Wil men de inhoud van een JList dynamisch wijzigen, dan moet dit via een omweg: het **DefaultListModel**. De code :

```
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;

public class DynKeuzelijst extends JFrame
    implements ListSelectionListener {
    private JList lstKies;
    private DefaultListModel mijnLijst;                                (1)

    public DynKeuzelijst() {
        setTitle("Test met een Dynamische Lijst");
    }

    public static void main (String[] args) {
        DynKeuzelijst frame = new DynKeuzelijst();
        frame.createGUI();
        frame.setVisible(true);
        frame.pack();
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout() );
        mijnLijst = new DefaultListModel();                            (2)
        mijnLijst.addElement("Bier");                                  (3)
        mijnLijst.addElement("Koffie");
        mijnLijst.addElement("Water");
        lstKies = new JList(mijnLijst);                                (4)
        add(lstKies);
        lstKies.addListSelectionListener(this);
    }

    public void valueChanged(ListSelectionEvent e){
        Object[] uwKeuze = lstKies.getSelectedValues();
        for(int i = 0; i < uwKeuze.length; i++)
            System.out.println(uwKeuze[i].toString());
        mijnLijst.addElement("Wijn");
        pack();
    }
}
```

(1) Er wordt een DefaultListModel gedeclareerd.

(2) Het DefaultListModel wordt aangemaakt.

- (3) Er worden elementen toegevoegd aan het DefaultListModel.
- (4) De keuzelijst wordt gemaakt, nu met een constructor die gebruik maakt van het DefaultListModel.

Er kunnen op ieder moment nog altijd elementen toegevoegd of verwijderd worden.

Opmerkingen:

Ook hier constateren we dat bij een selectie de valueChanged-method twee maal getriggered wordt: de eerste maal als de 'vorige' selectie verwijderd wordt, de tweede maal als de nieuwe selectie geplaatst wordt. De toevoeging gebeurt dan ook tweemaal. Een manier om dit te vermijden is een controle inbouwen of de waarde er al inzit en er voor zorgen dat ze slechts éénmaal wordt toegevoegd.

Het voorbeeld is ook in die mate flauw dat men altijd hetzelfde element toevoegt. In de praktijk zal men eerder met een tekstvak werken om een nieuwe waarde op te vragen en deze toe te voegen.

#### 15.4.6 JComboBox

Een combobox is een uitklapbare keuzelijst. Ze neemt slechts één regel in beslag en er kan slechts één item gekozen worden.

Bij een combobox is er ook de mogelijkheid om een waarde in te geven in het veld. Dit is standaard niet mogelijk. Daarvoor moet je de comboBox 'editable' maken.

EventListener: **ActionListener()**.

Nuttige method: *public object getSelectedItem()*

#### 15.4.7 JTextField

Een JTextField is een component om 1 regel tekst in te typen.

Er zijn meerdere constructors mogelijk: met default tekst, met lengte, ...

EventListener: **ActionListener()**. Echter alleen een <enter> wordt erkend als event. Het veld verlaten met een <tab> of met een muisklik, veroorzaakt geen event.

#### 15.4.8 JTextArea

Een JTextArea is een component om meerdere regels tekst in te typen.

Bij de constructor kan men een hoogte en een breedte opgeven, uitgedrukt in gemiddelde tekens van een gemiddeld font.

Men kan een JTextArea ook 'verpakken' in een JScrollPane om op die manier schuifbalken te krijgen.



### 15.4.9 JCheckBox

Om het gebruik van een JCheckBox te demonstreren hebben we volgend voorbeeld:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Keuzevakjes extends JFrame implements ItemListener { (1)
    private JCheckBox drinken;
    private JTextField resultaat;

    public Keuzevakjes() {
        setTitle("Test met keuzevakjes en radiobuttons");
    }

    public static void main (String[] args) {
        Keuzevakjes frame = new Keuzevakjes();
        frame.createGUI();
        frame.pack();
        frame.setVisible(true);
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout() );
        drinken = new JCheckBox("Wenst u iets te drinken?", false);
        resultaat = new JTextField(20);
        add(drinken);
        add(resultaat);
        drinken.addItemListener(this);
    }

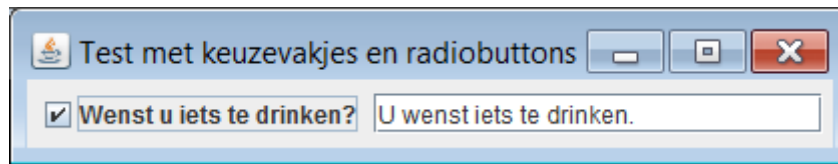
    public void itemStateChanged(ItemEvent e){ (2)
        if (drinken.isSelected()) (3)
            resultaat.setText("U wenst iets te drinken.");
        else
            resultaat.setText("U hebt dus geen dorst.");
    }
}
```

(1) De EventListener is de ***ItemListener()***.

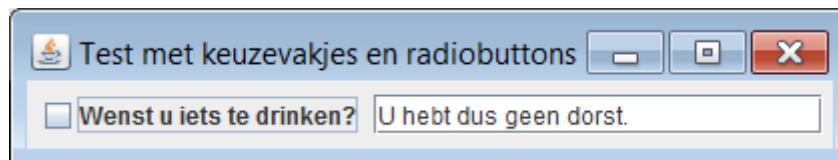
(2) De method van de interface om te definiëren is:  
***public void itemStateChanged(ItemEvent e)***

(3) De method om de status van het aankruisvak op te vragen: ***.isSelected()***. Deze method geeft true of false terug.

Resultaat:



Keuzevak uitvinken:



### 15.4.10 JRadioButton

Om radiobuttons in een groep te laten werken moet men ze toevoegen aan een (onzichtbare) **ButtonGroup**. Iedere radiobutton op zich moet wel toegevoegd worden aan het venster. Als voorbeeld voegen we wat code toe aan het vorige programma :

```
import ...

public class Keuzevakjes extends JFrame implements ItemListener {
    private ...
    private JRadioButton bier, water, wijn;

    ...
    private void createGUI() {
        ...
        drinken.addItemListener(this);
        bier = new JRadioButton("bier", false);
        water = new JRadioButton("water", false);
        wijn = new JRadioButton("wijn", false);
        ButtonGroup groep = new ButtonGroup();
        groep.add(bier);
        groep.add(water);
        groep.add(wijn);
        add(bier);
        add(water);
        add(wijn);
        drinken.addItemListener(this);
        bier.addItemListener(this);
        water.addItemListener(this);
        wijn.addItemListener(this);
    }
}
```

(1)

(2)

(3)



```

public void itemStateChanged(ItemEvent e){
    if (e.getSource()==drinken)
        if (drinken.isSelected())
            resultaat.setText("U wenst iets te drinken.");
        else
            resultaat.setText("U hebt dus geen dorst.");
    if (e.getSource()==bier) resultaat.setText("U wenst bier");
    if (e.getSource()==water) resultaat.setText("U wenst water");
    if (e.getSource()==wijn) resultaat.setText("U wenst wijn");
}
}

```

(4)

Resultaat:



- (1) De radiobuttons worden toegevoegd aan een buttongroep.
- (2) Ze worden ieder apart toegevoegd aan het venster...
- (3) ... en worden ook ieder afzonderlijk opgehangen aan de EventListener. De EventListener is ook hier de **ItemListener()**.
- (4) In de afhandeling van het event moeten we testen welk object het event heeft veroorzaakt, om zo de juiste tekst te tonen in het tekstveld.

Door het feit dat we de radiobuttons niet geplaatst hebben in een JPanel, wordt de layout weer afhankelijk van de breedte van het venster en van de breedte van andere componenten. Door elementen te groeperen in een panel, kunnen ze bij elkaar gehouden worden.

#### 15.4.11 DialogBoxes

Er zijn 4 soorten dialoogvensters, allen **static** methods van JOptionPane:

Naam van de method	Beschrijving
showConfirmDialog	Vraagt een bevestiging met yes/no/cancel knoppen.
showInputDialog	Prompt voor input.
showMessageDialog	Geeft een boodschap.
showOptionDialog	Een combinatie van de drie mogelijkheden.



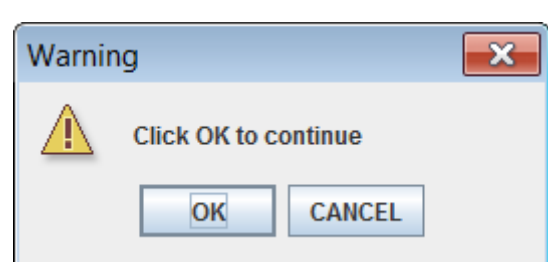
Voorbeelden:

```
import javax.swing.JOptionPane;

public class DialoogVensters {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Dit is een dialoogvenster");
        JOptionPane.showConfirmDialog(null, "Dit vraagt om bevestiging");
        String t = JOptionPane.showInputDialog(null, "Geef een antwoord");

        Object[] options = { "OK", "CANCEL" };
        JOptionPane.showOptionDialog(null, "Click OK to continue",
        "Warning", JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,
        null, options, options[0]);
    }
}
```

Met als resultaten:



Verschillende aspecten zijn in te stellen, de icoontjes die moeten afgebeeld worden, de knoppen die er moeten zijn, de titels van de venstertjes, ...

Voor de afhandeling van de knoppen en verdere details: zie API.

### 15.4.12 JMenu

Om een menusysteem te bouwen hebben we drie elementen nodig:

- Een JMenuBar die toegevoegd wordt aan het venster.
- Eén of meer JMenu's die toegevoegd worden aan de JMenuBar.
- JMenuItems en/of een Separator die toegevoegd worden aan een JMenu. De JMenuItems worden opgevolgd door een **ActionListener()**.



### Een voorbeeld :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Menuutje extends JFrame implements ActionListener {
    private JMenuBar menuBalk;
    private JMenu bestandMenu, wijzigMenu;
    private JMenuItem openItem, saveItem, copyItem, pasteItem, sluitAf;
    private JTextField keuze;

    public static void main (String[] args) {
        Menuutje frame = new Menuutje();
        frame.createGUI();
        frame.pack();
        frame.setVisible(true);
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        menuBalk = new JMenuBar();
        setJMenuBar(menuBalk);
        bestandMenu = new JMenu("Bestand");

        openItem = new JMenuItem("Openen");
        bestandMenu.add(openItem);
        openItem.addActionListener(this);

        saveItem = new JMenuItem("Bewaren");
        bestandMenu.add(saveItem);
        saveItem.addActionListener(this);

        bestandMenu.addSeparator();
        sluitAf = new JMenuItem("Afsluiten");
        bestandMenu.add(sluitAf);
        sluitAf.addActionListener(this);

        menuBalk.add(bestandMenu);
        wijzigMenu = new JMenu("Bewerken");
```

```

copyItem = new JMenuItem("Kopieren");
wijzigMenu.add(copyItem);
copyItem.addActionListener(this);

pasteItem = new JMenuItem("Plakken");
wijzigMenu.add( pasteItem );
pasteItem.addActionListener(this);

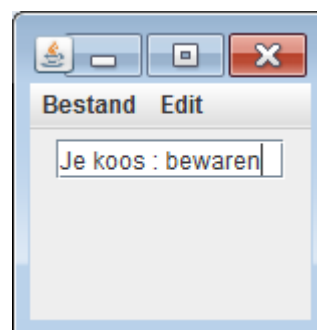
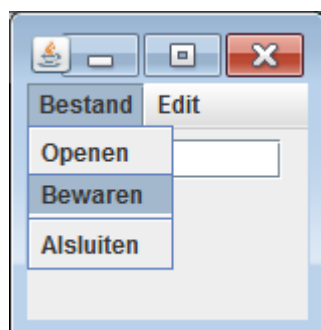
menuBalk.add(wijzigMenu);
keuze = new JTextField(10);
add(keuze);
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == openItem)
        keuze.setText("U koos : Openen ");
    if (e.getSource() == saveItem)
        keuze.setText("U koos: Bewaren");
    if (e.getSource() == sluitAf) System.exit(0);
    if (e.getSource() == copyItem)
        keuze.setText("U koos: Kopieren");
    if (e.getSource() == pasteItem)
        keuze.setText("U koos: Plakken");
}
}

```

- (1) De instructie `System.exit(0);` stopt het programma en geeft een waarde (in dit geval 0) terug aan het operating systeem. Op basis van de teruggegeven waarde kan het operating systeem verdere acties ondernemen.

Resultaat:



### 15.4.13 Andere nuttige swingcomponenten

Volgende andere nuttige swingcomponenten worden hier niet verder behandeld. Voor meer informatie kunt u er de API op naslaan.



- JSlider : Component waarmee je een waarde kan selecteren tussen grenswaarden middels het verschuiven van een soort knop.
- JTabbedPane : Paneel met tabbladen.
- JFileChooser : Mechanisme om een bestand te kiezen.
- JProgressBar : Een balk die de vooruitgang weergeeft van bepaalde taken.
- JColorChooser: Mechanisme om via een palet een kleur te kiezen en/of te manipuleren.
- ...

## 15.5 Layout en layout-managers

Afmetingen en rangschikking bepalen van componenten in een grafische gebruikers-interface is een kunst.

*Platformonafhankelijkheid dicteert dat de GUI van een Java toepassing altijd overzichtelijk en begrijpelijk moet zijn, wat ook de beschikbare ruimte is, althans binnen redelijke grenzen.*

Toepassingen in andere programmeeromgevingen worden vaak ontworpen met *absolute positionering*, dwz dat iedere component van elk venster een welbepaalde plaats en welbepaalde afmetingen krijgt.

**Absolute positionering** in Java is mogelijk, maar *niet wenselijk*, vermits de oplossing dan niet platformonafhankelijk is (verschil in fontsize, pixeldefinitie,...). In een goed geschreven Java-interface zullen de componenten automatisch hun onderlinge ligging en hun afmetingen aanpassen aan de beschikbare ruimte, zelfs als die tijdens het programma wijzigt.

Het object dat verantwoordelijk is voor de dynamische rangschikking en voor het bepalen van de afmetingen van componenten, is een **LayoutManager**. De programmeur kan zelf een class definiëren die deze interface implementeert of kan gebruik maken van één van de bestaande lay-outmanagers.

De belangrijkste zijn:

- FlowLayout (default voor applet, panel, ...)
- BorderLayout (default voor een venster)
- GridLayout

Minder gebruikt:

- GridBagLayout
- BoxLayout
- CardLayout

### 15.5.1 FlowLayout

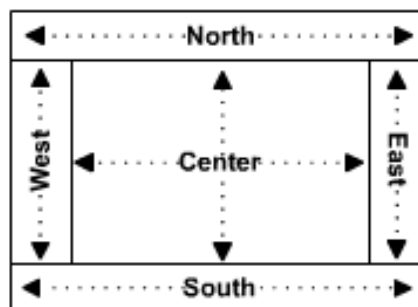
Tot nu toe hebben we altijd met de flow-layout gewerkt. De werking hiervan is dus bekend: alle componenten worden van links naar rechts, van boven naar beneden gerangschikt.

Men kan een flow-layout instellen voor left-alignment, right-alignment of centered (=default).

Men kan elementen bij elkaar houden door ze te groeperen in een panel (JPanel).

### 15.5.2 BorderLayout

Bij de borderlayout wordt het venster ingedeeld in 5 gebieden. De borderlayout rangschikt de elementen in 5 zones: **noord** (boven), **zuid** (onder), **oost** (rechts), **west** (links), en in het **centrum**.



Per gebied kan men één component zetten. Dit kan evenwel een panel zijn waarin meerdere componenten opgeslagen worden !

Niet alle gebieden moeten gebruikt worden.

Voorbeeld:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestBorder extends JFrame implements ActionListener {
    private JButton cmdEen, cmdTwee;
    private JTextField txtResultaat;
    private JPanel paneel;
    (1)

    public static void main (String[] args) {
        TestBorder frame = new TestBorder();
        frame.createGUI();
        frame.setSize(350,200); //instellen breedte,hoogte (ipv pack())
        frame.setVisible(true);
    }
}
```



```
private void createGUI() {  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    setLayout(new BorderLayout() );  
    paneel = new JPanel();  
    cmdEen = new JButton("Een");  
    cmdTwee = new JButton("Twee");  
    paneel.add(cmdEen);  
    paneel.add(cmdTwee);  
    JLabel titel = new JLabel("Ik sta in het noorden");  
    add(titel, BorderLayout.NORTH);  
    txtResultaat = new JTextField(30);  
    add(txtResultaat, "Center");  
    add(paneel, BorderLayout.SOUTH);  
    cmdEen.addActionListener(this);  
    cmdTwee.addActionListener(this);  
}  
  
public void actionPerformed(ActionEvent e) {  
}  
}
```

- (1) Er worden twee knoppen, één tekstveld en één paneel gebruikt.
- (2) De layout van het venster wordt op BorderLayout gezet. Dit is eigenlijk overbodig omdat dit de standaard layout is voor een venster.
- (3) De knoppen worden op een paneel gezet.
- (4) Er wordt een label gemaakt en dit wordt in de **North**-area geplaatst.
- (5) Het tekstvak wordt in de **Center**-area geplaatst.
- (6) Het paneel wordt in de **South**-area geplaatst.

De **East** en de **West**-area worden niet gebruikt. De niet gebruikte ruimte wordt ingenomen door de **Center**-area.

Resultaat:



Opmerking:

Het paneel dat we gebruiken, werkt zelf met een layoutmanager, namelijk de default flow-layout. Ook dit kunnen we anders instellen!

### 15.5.3 GridLayout

Bij een gridlayout wordt het venster ingedeeld in een aantal rijen en kolommen. De elementen worden hiernaar gerangschikt (kan meegegeven worden aan de constructor). Alle rijen hebben dezelfde hoogte en alle kolommen dezelfde breedte. Iedere cel heeft dus dezelfde afmetingen en kan één component bevatten.

De cellen worden in volgorde van toevoegen opgevuld, van links naar rechts en van boven naar onder. Bijvoorbeeld :



Een volgend codevoorbeeld schikken we twee labels, twee tekstvakken en twee knoppen.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestGrid extends JFrame implements ActionListener {
    private JButton cmdEen, cmdTwee;
    private JTextField txtNaam, txtEmail;

    public static void main (String[] args) {
        TestGrid frame = new TestGrid();
        frame.createGUI();
        frame.setSize(450,200); //instellen breedte,hoogte (ipv pack())
        frame.setVisible(true);
    }

    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(3,2));
        cmdEen = new JButton("Controle");
        cmdTwee = new JButton("Wissen");
        JLabel lblNaam = new JLabel("Uw naam:");
        add(lblNaam);
        txtNaam = new JTextField(10);
        add(txtNaam);
    }
}
```



```
JLabel lblEmail = new JLabel("Uw e-mail:");
add(lblEmail);
txtEmail = new JTextField(10);
add(txtEmail);
add(cmdEen);
add(cmdTwee);
cmdEen.addActionListener(this);
cmdTwee.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
}

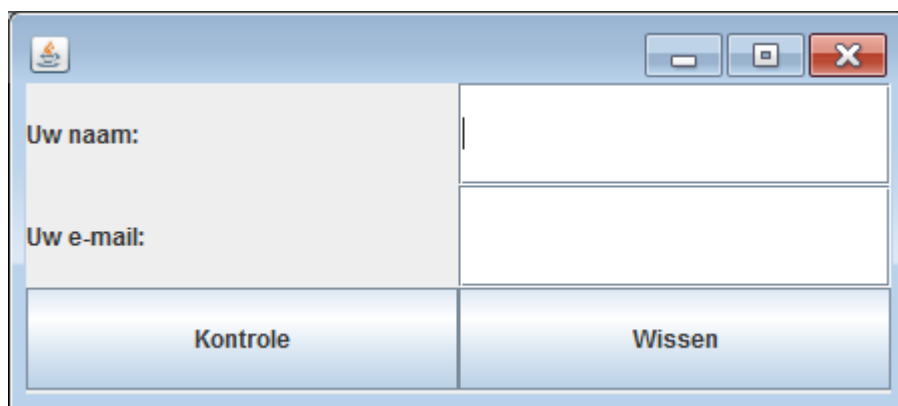
}
```

(2)

(1) De GridLayout wordt ingesteld met 3 rijen en 2 kolommen.

(2) Alle elementen worden aangemaakt en toegevoegd aan het venster. De volgorde van toevoegen bepaalt ook de plaatsing!

Het resultaat:



Opmerkingen:

- Alle 'cellen' krijgen evenveel plaats (alle rijen hebben dezelfde hoogte en alle kolommen dezelfde breedte).
- De component neemt altijd de volledige ruimte in.

#### 15.5.4 Het nesten van layouts

Indien men een eigen, aangepaste layout wil creëren, dan kan dit door gebruik te maken van panels en door per panel een specifieke layout in te stellen.

Volgende standaard layouts worden hier niet verder besproken :

- **GridBagLayout:** rangschikt de elementen in een rechthoekige tabel, waarbij sommige elementen verschillende cellen van de tabel innemen. De hoogte van



de rijen en de breedte van de kolommen wordt onder meer bepaald door gewichtsfactoren.

- **BoxLayout:** is een flexibel alternatief voor de FlowLayout dat geïnspireerd is op de technieken die tekstverwerkers gebruiken om tekst en afbeeldingen op een bladzijde te rangschikken.
- **CardLayout:** laat toe groepen van componenten nu eens te tonen, dan weer te verbergen en te vervangen door een andere groep componenten, bijv. om een zogenaamde *tabbed dialog* te implementeren (JTabbedPane).

## 15.6 Events & EventListeners

Veel componenten kunnen gevolgd worden door een EventListener.

### 15.6.1 Een overzicht

ListenerInterface	Methods in de Interface	Componenten
ActionListener	actionPerformed(ActionEvent)	JButton JList JTextField JMenuItem
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	JScrollbar
ComponentListener	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)	Component en alle afgeleiden!
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)	Container en alle afgeleiden.
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	Component en alle afgeleiden.
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	Component en alle afgeleiden.
MouseListener	mouseClicked(MouseEvent)	Component en



ListenerInterface	Methods in de Interface	Componenten
	mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	alle afgeleiden
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	Component en alle afgeleiden
WindowListener	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)	Window en zijn afgeleiden zoals : JFrame JDialog JFileDialog
ItemListener	itemStateChanged(ItemEvent)	JCheckBox JComboBox JList JCheckBox-MenuItem
TextListener	textValueChanged(TextEvent)	JTextArea JTextField

Opmerking:

Wil men een component laten volgen door een EventListener, dan moet men die component registreren bij die listener.

De method hiervoor is altijd:

*Component.addNaamVanDeListener(Wie\_moet\_het\_signaal\_ontvangen)*

Bijvoorbeeld: een knop registreren:

*Knop.addActionListener(this)*

*'this'* op voorwaarde dat de class de interface implementeert.

Men kan ook een component loskoppelen van een eventlistener. Men gebruikt daarvoor altijd de method:

*Component.removeNaamVanDeListener()*

### 15.6.2 Het gebruik van Adapters

Sommige Listeners hebben behoorlijk wat (abstracte) methods die, bij implementatie, moeten geconcretiseerd worden in de eigen class.

Zie bijvoorbeeld: WindowListener en MouseListener.

Om werk te besparen, bestaan er adapterclasses (WindowAdapter, MouseAdapter, ...). Deze adapterclasses implementeren de interface en hebben een concrete invulling gemaakt van alle abstracte methods. Zij hebben die abstracte methods overriden met lege methods.

Maken we een eigen eventafhandelingsclass en laten wij die erven van zo'n adapter, dan moeten wij enkel die (lege) methods overschrijven die wij nodig hebben. Die eigen eventafhandelingsclass wordt dan de 'target' voor het event.

Bijvoorbeeld: indien we enkel geïnteresseerd zijn in de afsluiting van windows, dan is het volgende een oplossing:

```
class CloseWindowListener extends WindowAdapter {
    public void windowClosing (WindowEvent evt) {
        System.exit(0);
    }
}

...

//in de frame class :
this.addWindowListener(new CloseWindowListener());
```

We schrijven een aparte class om het sluiten af te handelen en benoemen deze class als de bestemming die het event gaat afhandelen.



Een alternatief:

```
this.addWindowListener (
    new WindowAdapter() {
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    }
);
```

Waar we ons venster toevoegen aan de WindowListener, maken we een (anonieme) inner class die het event afhandelt.

Voorbeeld:

We hernemen het voorbeeld uit 14.4.5, het voorbeeld met de JList waar telkens een element dubbel werd toegevoegd. We gaan dit nu oplossen via een **MouseListener** die we via een **MouseAdapter** gaan implementeren. De code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;                                     (1)

public class DynKeuzelijst2 extends JFrame {                 (2)
    private JList lstKies;
    private DefaultListModel mijnLijst;

    public DynKeuzelijst2() {
        setSize(400, 200);
        setTitle("Test met een Lijst");
    }

    public static void main (String[] args) {
        DynKeuzelijst2 frame = new DynKeuzelijst2();
        frame.createGUI();
        frame.setVisible(true);
    }
    private void createGUI() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout() );
        mijnLijst = new DefaultListModel();
        mijnLijst.addElement("Bier");
        mijnLijst.addElement("Koffie");
        mijnLijst.addElement("Water");
        lstKies = new JList(mijnLijst);
        add(lstKies);
        lstKies.addMouseListener(new MijnMuisWaker());      (3)
    }
}
```

```
class MijnMuisWaker extends MouseAdapter { (4)
    @Override
    public void mouseClicked(MouseEvent e) {
        Object[] uwKeuze = lstKies.getSelectedValues();
        for(int i = 0; i < uwKeuze.length; i++) {
            System.out.println(uwKeuze[i].toString());
        }
        mijnLijst.addElement("Wijn");
    }
}
```

Aanpassingen t.o.v. de eerste oplossing:

- (1) Een `MouseListener` behoort tot `java.awt.event`, de oorspronkelijke `ListSelectionListener` behoort tot `javax.swing.event`.
- (2) De listener wordt niet geïmplementeerd.
- (3) De lijst wordt nu opgevolgd door een `MouseListener`, de target voor de eventafhandeling is een nieuw te creëren, anoniem, adapterobject.

De oorspronkelijke regels 34 .. 39 zijn vervangen door:

- (4) De definitie van een geneste innerclass ***MijnMuisWaker*** die erft van de ***MouseAdapter***. Hierdoor beschikt deze class over de concrete, maar lege, methods ***mouseClicked***, ***mouseEntered***, ***mouseExited***, ***mousePressed*** en ***mouseReleased***.

Aangezien wij enkel geïnteresseerd zijn in het clickEvent wordt hier de code geschreven die het clickEvent afhandelt.

Het resultaat is nu zoals het hoort. De juiste selectie wordt getoond en het nieuwe element wordt slechts éénmaal toegevoegd.

Opmerking: een innerclass heeft toegang tot alle elementen van de class waarbinnen ze gedeclareerd wordt. Vandaar dat in de method `mouseClicked()` kan gewerkt worden met de `JList`-elementen.

Voordelen van het gebruik van adapters:

- Men moet minder code schrijven. Abstracte methods (uit de interface) waarin we geen interesse hebben moeten niet vervangen worden door lege, concrete methods.
- Men kan verschillende widgets een verschillende target geven. Een muisklik in een tekstvenster kan naar een andere adapterclass gestuurd worden als een muisklik op een selectielijst. Dit maakt de code gestructureerder en overzichtelijker.



## 15.7 Oefeningen



Maak oefeningen 34 t.e.m. 38 uit de oefenmap.

## 15.8 Een GUI creëren met de NetBeans IDE

Een GUI helemaal zelf creëren vanuit code kan al snel een tijdrovende bezigheid worden. De ontwikkelomgeving NetBeans biedt de mogelijkheid om de GUI op te bouwen door controls, menu's e.d. naar een JFrame Form te slepen.

In deze paragraaf zullen we niet alle mogelijke controls bekijken maar een aantal eenvoudige applicaties maken. Wie nog meer wil lezen over de creatie van swing GUI's met netbeans kan terecht op <http://www.netbeans.org/kb/60/java/quickstart-gui.html> en op <http://java.sun.com/docs/books/tutorial/uiswing/components/index.html>.

### 15.8.1 Een eenvoudig voorbeeld

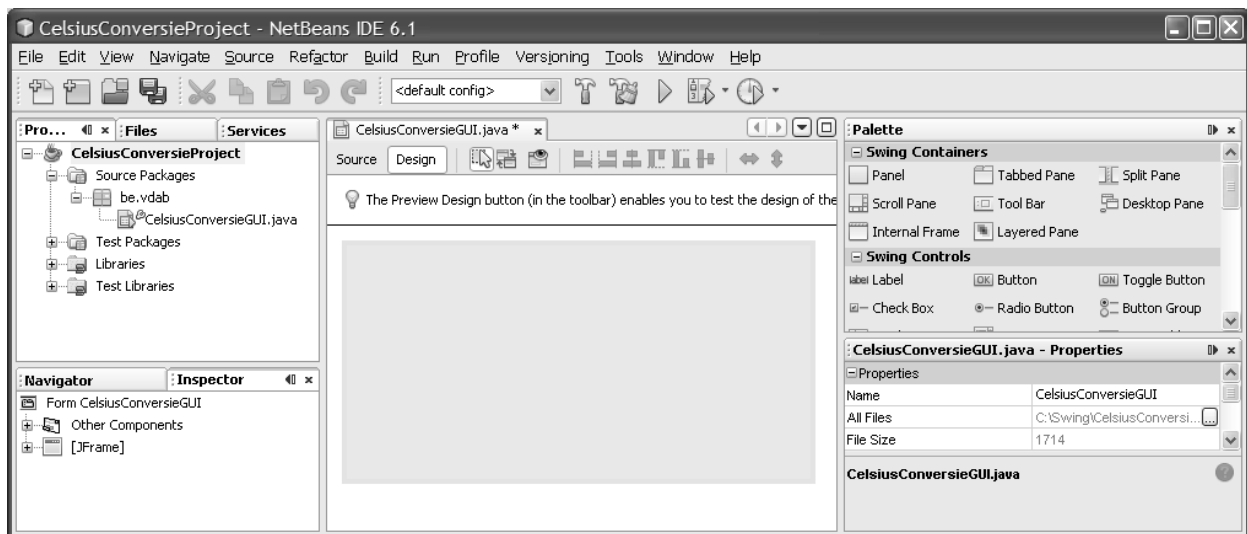
Als eerste voorbeeld bouwen we een swingapplicatie op, die een temperatuur converteert van graden Celsius naar graden Fahrenheit.

In NetBeans maken we een nieuw project via File – New Project... In het dialoogvenster dat verschijnt kiezen we onder Categories voor Java (General bij NetBeans 5.x) en Java Application bij Projects. Kies Next.

Geef het project een naam (bijv. CelsiusConversieProject) en vink 'Create Main Class' uit. We gaan namelijk zelf een class maken die als startpunt zal dienen voor onze applicatie. Klik op Finish.

We voegen nu een JFrame toe aan het project door bijvoorbeeld in het Projects-venster met de rechtermuistoets op de naam van het project te klikken en te kiezen voor New – JFrame Form. Je geeft deze een naam, bijvoorbeeld CelsiusConversieGUI en je stopt deze in een geschikte package. Klik op Finish.

In de NetBeans IDE zie je nu grafische weergave van de JFrame. Bovenaan kan je via de tab Source naar de bijhorende code gaan kijken. Rechts vind je een palet met allerlei swing-controls, -menu's en -containers. Aan de linkerkant zullen we ook even gebruik maken van het Inspector-venster.

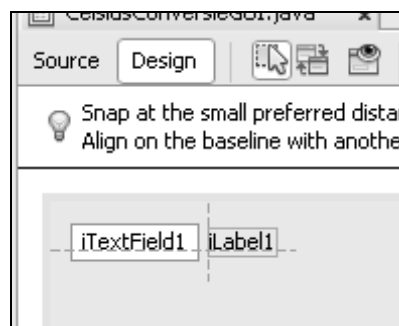


Als we even een kijkje nemen naar de code die voor ons aangemaakt werd, zien we dat netbeans voor ons een private method  `initComponents()` gemaakt heeft. In die method wordt de default close operation ingesteld, diverse GUI componenten geïnitieerd, worden enkele layout specifieke taken verricht en worden alle componenten op het scherm gezet.

We gaan verder met het aanmaken van onze GUI. Eerst stellen we de titel in via het Inspector-venster. Zorg er wel voor dat je JFrame in Design View staat, anders kan je het Inspector-venster niet bereiken. Klik in het Inspector-venster op het JFrame en wijzig in het Property-venster de titel naar 'Celsius Conversie'.

We voegen nu een JTextField toe: sleep vanuit de Palette een TextField-control naar de frame en laat deze los op een kleine afstand van de linkerbovenhoek van het frame. Gestippelde lijnen helpen je om controls t.o.v. mekaar uit te lijnen.

Naast de JTextField control plaatsen we nu een JLabel. De stippellijnen helpen je opnieuw om de nodige afstand te bewaren tussen de controls en ook om de onderkant van de tekst in beide controls met elkaar uit te lijnen.



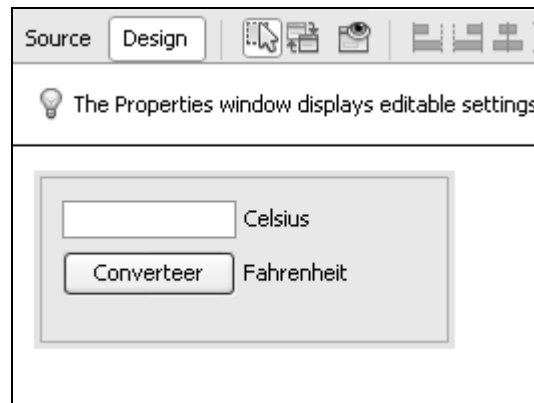
Plaats nu onder de JTextField control een JButton en ernaast een tweede label.

We wijzigen nu het opschrift van de JButton naar 'Converteer'. Dat kan door de button te selecteren in het Design-venster en er op te klikken met de rechtermuistoets. Je kiest dan 'Edit Text'. Op een analoge wijze verander je het opschrift van het bovenste label naar 'Celsius' en het onderste naar 'Fahrenheit'. De inhoud van het JTextField wis je.



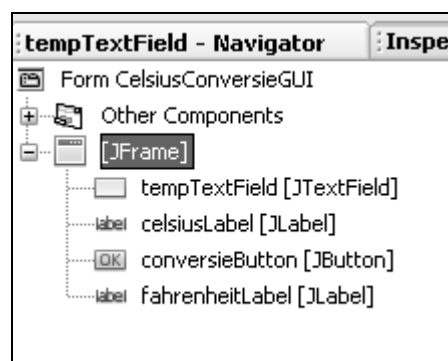
Het JTextField is nu heel erg smal geworden. Om het even breed te maken als de button selecteer je de button en shift-klik je het JTextField. Met de shift-toets voeg je extra controls toe aan de selectie. Rechtsklik nu op de selectie en kies Same Size, Same Width.

Tot slot kan je het frame verkleinen tot de gewenste grootte. Onze GUI is nu klaar en zou er ongeveer als volgt moeten uitzien :



Als je nu even een kijkje gaat nemen in de source (via de tab Source) dan zie je dat netbeans je heel wat werk uit handen heeft genomen.

Omdat het niet handig programmeren is met controlnamen zoals jTextField1 en jLabel2 gaan we nu de namen van deze variabelen aanpassen via het Inspector venster. Vergeet niet eerst eventueel terug te gaan naar de Design. Klik met de rechtermuis-toets op het jTextField en kies Change Variable Name... Je wijzigt het in tempTextField. Wijzig op analoge wijze de naam van het eerste label in celsiusLabel, de naam van de button in conversieButton en de naam van het tweede label in fahrenheitLabel.



Het is nu de bedoeling dat er een stukje code wordt uitgevoerd wanneer de gebruiker op de button klikt. Selecteer de button en klik erop met de rechtermuistoets. Kies dan Events, Action, actionPerformed.

We komen nu in de source terecht, meer bepaald in een method conversieButtonActionPerformed(). Daarin voegen we volgende code toe :

```
int tempFahr = (int)((Double.parseDouble(tempTextField.getText())) *
                    1.8 + 32);
```



```
fahrenheitLabel.setText(tempFahr + " Fahrenheit");
```

Je kan het programma nu laten lopen. NetBeans zal je wel nog komen vragen welke class de main class moet worden. In ons programma is dit de class CelsiusConversie-GUI.

### 15.8.2 Radiobuttons en checkboxes

In het volgende voorbeeld concentreren we ons vooral op radiobuttons en checkboxes. Radiobuttons worden over het algemeen gebruikt om aan de gebruiker aan te geven dat hij uit meerdere mogelijkheden slechts één enkele optie kan kiezen. Checkboxes daarentegen duiden aan dat meerdere opties mogelijk zijn. Dit is althans een aangewezen werkwijze bij het ontwerp van een GUI.

Technisch gesproken wordt het verschil tussen één mogelijke selectie of meerdere bepaald door het gebruik van een buttongroup. Om ervoor te zorgen dat er slechts één enkele radiobutton tegelijk ingedrukt is, voeg je een Swing control 'Button Group' toe aan de frame. Je laat meerdere radiobuttons behoren tot dezelfde button group door hun property buttonGroup in te stellen. Bij checkboxes laat je dus de property buttonGroup bij voorkeur leeg zodat ze onafhankelijk van andere controls kunnen aangevinkt worden.

Een button group heeft visueel geen weergave. Meestal worden groepen van radiobuttons of checkboxes echter in een kadertje weergegeven, eventueel voorzien van een titel. Hiervoor zullen we een panel gebruiken dat voorzien is van een titled border.

Het voorbeeld nu: we maken een zelfbedieningspaneel voor een fastfoodrestaurant. Bovenaan kiezen we een menu, daaronder specificeren we het formaat van het zakje frietjes en de soort en grootte van het drankje. Verder geven we aan welke sausjes erbij moeten, voor welke tafel de bestelling is en geven we eventueel een kredietkaart-nummer op. Helemaal onderaan staan twee knoppen: één om de bestelling te wissen en één om de bestelling door te voeren. Ter info wordt het totaalbedrag van de bestelling weergegeven en ook een samenvatting van de bestelling.



We maken een nieuw project (zonder main class) en voegen een JFrame Form toe. Op de form voegen we vijf panels toe. Voor elk panel klik je in het properties-venster naast de property 'border' op de drie puntjes helemaal rechts. Bij Available Borders kies je Titled Border, eronder vul je de Title property in.

Aan de vier bovenste panels – deze met de radiobuttons – voeg je een Button Group control toe. Geef deze een betere naam zoals we dat ook in de vorige paragraaf voor een aantal controls hebben gedaan.

Voeg nu de radiobuttons toe en stel per panel de property buttonGroup van elke radiobutton in zodat er telkens maar één radiobutton ingeschakeld kan zijn per panel.

Voeg ook de checkboxes toe. Voor deze controls hoeven we geen buttonGroup in te stellen omdat we meerdere checkboxes tegelijk moeten kunnen aanvinken.

Onder de panels voegen we een label toe met de tekst "Tafel" en een Spinner control. Daarnaast komt ook nog een label met de tekst "Kredietkaartnummer" en een Formatted Field. We stellen de format van deze control in door bij de property formatterFactory op de drie puntjes uiterst rechts te klikken. We komen dan in een dialoogvenster waar je onder Category kiest voor mask en daarnaast bij Format voor custom. Rechts in het dialoogvenster kan je dan het masker "#####" intikken.

Daaronder voegen we nog twee buttons toe met het opschrift "Wissen" en "Afrekenen". Naast deze buttons voeg je twee labels toe: één met de tekst "Totaal:" en één zonder tekst.

Onder de buttons tenslotte komen de laatste twee labels, één met de tekst "Bestelling:" en ééntje zonder tekst. Voorzie voldoende ruimte voor dit laatste label want hier komt een aantal lijnen tekst in te staan. Zet ook nog de property resizable van de JFrame uit.

We voegen nu de code toe om de totale kostprijs te berekenen van de bestelling.

Dubbeltklik op de afrekenen-button en voeg de volgende code toe :

```
float werkprijs = 3.0F; (1)
if (jRadioButtonSmall.isSelected())
    werkprijs += 1.4F;
else if (jRadioButtonMedium.isSelected())
    werkprijs += 1.8F;
else if (jRadioButtonLarge.isSelected())
    werkprijs += 2.2F; (2)

if (jRadioButton30cl.isSelected())
    werkprijs += 1.3F;
else if (jRadioButton40cl.isSelected())
    werkprijs += 1.7F;
else if (jRadioButton50cl.isSelected())
    werkprijs += 2.0F; (3)

Component[] sausjesElements = jPanelSausjes.getComponents();
for (int i=0;i<sausjesElements.length;i++) {
    if (sausjesElements[i] instanceof JCheckBox) {
        JCheckBox sausje = (JCheckBox)sausjesElements[i];
        if (sausje.isSelected()) {
            werkprijs += 0.50F; (4)
        }
    }
}

String m = String.valueOf(werkprijs) + " €";
labelTotaal.setText(m); (5)
```

- (1) We zetten de totaalprijs in op 3 euro voor de gekozen menu.
- (2) Afhankelijk van de gekozen portie frietjes tellen we een bedrag bij. Testen of een radiobutton geselecteerd is kan met de method `isSelected()`.
- (3) Idem voor de grootte van het drankje.
- (4) Voor de sausjes moeten we de checkboxen één voor één afgaan en controleren of ze geselecteerd zijn. De checkboxen zijn allemaal componenten van het panel `jPanelSausjes`. Via de method `getComponents()` verkrijgen we een array van `Components`. Voor alle zekerheid testen we nog even of het wel checkboxen zijn vooraleer de componenten te converteren naar checkboxen en na te gaan of ze geselecteerd zijn of niet. Als ze geselecteerd zijn verhogen we de prijs.
- (5) Tenslotte geven we het totaal weer in het label.

We moeten nu nog de tekstuele samenvatting van de bestelling samenstellen. Bij de berekening van de kostprijs van de bestelling hebben we al de sausjes-checkboxen



overlopen. We voegen een beetje code toe (in vetdruk) zodat we een string opbouwen met de aangevinkte sauzen :

```
String sausjes = ""; (1)
Component[] sausjesElements = jPanel1Sausjes.getComponents();
for (int i=0;i<sausjesElements.length;i++) {
    if (sausjesElements[i] instanceof JCheckBox) {
        JCheckBox sausje = (JCheckBox)sausjesElements[i];
        if (sausje.isSelected()) {
            werkprijs += 0.50F;
            sausjes += sausje.getActionCommand(); (2)
            sausjes += ", ";
        }
    }
}
if (! sausjes.equals(""))
sausjes = sausjes.substring(0,sausjes.length()-2); (3)
```

- (1) We starten met een lege string.
- (2) Om de bijhorende tekst van een checkbox op te halen gebruiken we de method `getActionCommand()`. We voegen ook een komma en een spatie toe.
- (3) Als we minstens één sausje geselecteerd hebben mag de laatste komma en spatie er opnieuw af.

De rest van de tekst bouwen we als volgt op :

```
...
String tekst = "Tafel " + jSpinnerTafel.getValue() +
               " bestelde een "; (1)

Enumeration<AbstractButton> elements =
    buttonGroupMenus.getElements();
while (elements.hasMoreElements()) {
    AbstractButton radio = elements.nextElement();
    if (radio.isSelected()) tekst += radio.getActionCommand();
} (2)

tekst += " menu met ";
elements = buttonGroupFriet.getElements();
while (elements.hasMoreElements()) {
    AbstractButton radio = elements.nextElement();
    if (radio.isSelected() &&
        radio.getActionCommand().equals("Small (1,40 €)"))
        tekst += "een kleine portie";
    if (radio.isSelected() &&
        radio.getActionCommand().equals("Medium (1,80 €)"))
        tekst += "een medium portie";
    if (radio.isSelected() &&
        radio.getActionCommand().equals("Large (2,20 €)"))
```

```
        tekst += "een grote portie";
    }
    tekst += " frietjes met daarbij een ";           (3)

    elements = buttonGroupDrank.getElements();
    while (elements.hasMoreElements()) {
        AbstractButton radio = elements.nextElement();
        if (radio.isSelected()) tekst += radio.getActionCommand();
    }
    tekst += " van ";                               (4)

    elements = buttonGroupGrootte.getElements();
    while (elements.hasMoreElements()) {
        AbstractButton radio = elements.nextElement();
        if (radio.isSelected()) tekst +=
            radio.getActionCommand().substring(0,5);
    }                                               (4)
    tekst += ". Sausjes : " + sausjes;              (5)
    tekst += ". Wenst te betalen met kredietkaart " +
        jTextFieldKrediet.getText();
    tekst = "<html>" + tekst + "</html>";           (6)
    jLabelBestelling.setText(tekst);

    String m = String.valueOf(werkprijs) + " €";
    labelTotaal.setText(m);
```

- (1) De waarde van de spinner halen we op met `getValue()`.
- (2) Om uit een `buttongroup` de geselecteerde `radiobutton` te vinden moeten we alle `radiobuttons` overlopen en ze één voor één controleren. Een `buttongroup` heeft een method `getElements()`. Deze levert ons een `Enumeration` van `AbstractButtons` op. We overlopen deze `Enumeration` in een lus en als we een geselecteerde `radiobutton` treffen voegen we zijn `actioncommand` toe aan de tekst.
- (3) De frieten-`buttongroup` overlopen we op een gelijkaardige manier.
- (4) Ook bij het soort drank en de grootte ervan gebruiken we dezelfde techniek.
- (5) Tenslotte voegen we de tekst van de sausjes toe en de inhoud van de `formattedtextfield`...
- (6) ...en geven we de tekst tussen `html`-tags weer in het label. Dit laatste doen we om de lange tekst in het label over meerdere lijnen te spreiden (`wrap text`).

We moeten nu nog code toevoegen om de `radiobuttons` en de `checkboxes` terug te resetten.

Eerst de `checkboxes`: voeg code toe aan de `Wissen`-button door bijvoorbeeld in de design met de rechtermuistoets op de button te klikken en te kiezen voor `Events – Action – actionPerformed`. Voeg de volgende code toe :



```
//sausjes-selectie wissen
Component[] sausjesElements = jPanelSausjes.getComponents();
for (int i=0;i<sausjesElements.length;i++) {
    if (sausjesElements[i] instanceof JCheckBox) {
        JCheckBox sausje = (JCheckBox)sausjesElements[i];
        if (sausje.isSelected()) sausje.setSelected(false);
    }
}
```

Deze code komt ons al enigszins bekend voor. Nieuw is de method `setSelected(false)` maar deze spreekt voor zichzelf.

Om de radiobuttons te 'unselect'-en gebruik je de method `clearSelection()` van de betreffende buttongroup :

```
//menu's-selectie wissen
buttonGroupMenus.clearSelection();

//frieten-selectie wissen
buttonGroupFriet.clearSelection();

//drinks-selectie wissen
buttonGroupDrank.clearSelection();

//drinksize-selectie wissen
buttonGroupGrootte.clearSelection();
```

Er is echter een probleem bij oudere JDK's. Bij JDK's vóór JDK6 kent de buttongroup geen method `clearSelection()`. Wat we dan wél kunnen doen is een dummy-radiobutton toevoegen aan elke buttongroup en deze dummy onzichtbaar maken. Wanneer we deze dan via code selecteren dan zijn alle andere radiobuttons uit de buttongroup niet geselecteerd.

Als we deze dummyradiobuttons in design toevoegen dan wil netbeans de nodige ruimte voorzien voor deze radiobutton terwijl dit helemaal niet nodig is. Het is immers niet de bedoeling dat het panel vergroot voor een radiobutton die toch onzichtbaar blijft. Dus voegen we de dummyradiobuttons via code toe.

Helemaal onderaan de code voeg je een aantal private variabelen toe, één per dummyradiobutton :

```
...
// End of variables declaration
private javax.swing.JRadioButton jRadioButtonDummy1;
private javax.swing.JRadioButton jRadioButtonDummy2;
private javax.swing.JRadioButton jRadioButtonDummy3;
private javax.swing.JRadioButton jRadioButtonDummy4;
}
```

Eigenlijk moeten we nu ook code toevoegen in de initComponents()-method maar deze is beveiligd. De oplossing is code toevoegen in de constructor van onze JFrame net na de oproep van de method initComponents() :

```
public SelfServeGUI() {  
    initComponents();  
    jButtonDummy1 = new javax.swing.JButton();  
    jButtonDummy1.setVisible(false);  
    buttonGroupMenus.add(jButtonDummy1);  
    jButtonDummy2 = new javax.swing.JButton();  
    jButtonDummy2.setVisible(false);  
    buttonGroupFriet.add(jButtonDummy2);  
    jButtonDummy3 = new javax.swing.JButton();  
    jButtonDummy3.setVisible(false);  
    buttonGroupDrank.add(jButtonDummy3);  
    jButtonDummy4 = new javax.swing.JButton();  
    jButtonDummy4.setVisible(false);  
    buttonGroupGrootte.add(jButtonDummy4);  
}
```

We laten de references nu verwijzen naar een nieuwe instance van een jButton en maken de radiobutton onzichtbaar. We voegen de dummy-radiobutton dan toe aan een buttongroup.

De code voor het wissen van de radiobuttons wordt dan heel eenvoudig :

```
//menu's-selectie wissen  
jRadioButtonDummy1.setSelected(true);  
  
//frieten-selectie wissen  
jRadioButtonDummy2.setSelected(true);  
  
//drinks-selectie wissen  
jRadioButtonDummy3.setSelected(true);  
  
//drinksize-selectie wissen  
jRadioButtonDummy4.setSelected(true);
```

Deze bovenstaande code voeg je toe én ook de onderstaande code voor het wissen van de overige controls :

```
jLabelBestelling.setText("");  
jLabelTotaal.setText("");  
jSpinnerTafel.setValue(0);  
jFormattedTextFieldKrediet.setValue("");
```



### 15.8.3 Menu's en dialogs

In deze paragraaf maken we een swingtoepassing waarmee we een tekstbestand kunnen bekijken en voorzien van wat opmaak. Dit doen we aan de hand van menu-items en dialoogvensters.

Maak een nieuw project. Laat netbeans geen main class maken maar voeg zelf een JFrame form toe. Voorzie de frame van een titel.

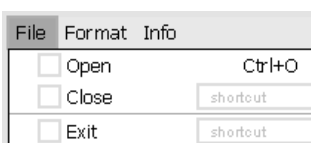
Voeg eerst een Menu Bar toe aan de JFrame. In het Inspectorvenster zie je dat er een JMenuBar is toegevoegd met twee JMenu's. Selecteer het tweede JMenu en klik erop met de rechtermuistoets. Kies 'Edit Text' en vervang 'Edit' door 'Format'. Sleep vanuit de palette een derde Menu-control rechts van het menu Format. Vervang de tekst 'Menu' door 'Info'.

Sleep nu twee maal een Menu Item control bovenop het menu File, vervolgens een Separator en tenslotte nog een Menu Item. Je kan de opschriften van deze menuitems ook wijzigen via het designvenster: klik met de rechtermuistoets op Item en kies Edit Text. Zorg er zo voor dat het menu File de menuitems Open, Close en Exit bevat.

Van het menu-item Close zet je de Enabled property uit. We starten immers met een lege JTextArea en dus heeft het menu-item Close geen zin.

Het menuitem Open geef je de shortcut Ctrl+O. Dubbelklik daartoe op het vakje 'shortcut' en tik in het vak Key Stroke op Ctrl+O. Elk menuitem bevat links een leeg vierkantje. Dit is een placeholder voor een icoontje. Een icoontje voeg je toe door op deze placeholder te dubbelklikken. In dit voorbeeld gaan we geen iconen toevoegen.

Voeg aan het menu Info een menuitem About toe. Aan het menu Format voeg je eerst twee controls van het type Menu Item / CheckBox toe, dan een Separator, twee gewone Menu Items, opnieuw een Separator en tenslotte nog een gewoon Menu Item. Het eerste menu item met checkbox noem je 'Bold' en het tweede 'Italic'. Van beide menuitems zet je de property 'selected' uit. De volgende twee menuitems krijgen de naam 'SetColor' en 'SetBackColor'. Het onderste krijgt het opschrift Font.



Op het JFrame sleep je nu een Swing Text Area. Geef deze de maximale ruimte. De TextArea krijgt best ook een betere naam want we zullen deze dikwijls vanuit code moeten aanspreken.

We implementeren nu de File menu-items. Selecteer het menu-item File en geef deze een betere naam zoals bijv. JMenuItemOpen. Klik met de rechtermuistoets op dit menu-item en kies Events – Action – actionPerformed. Voeg volgende code toe :



```
JFileChooser fc = new JFileChooser();                                (1)
fc.setDialogTitle("Open File Dialog");                            (2)
int returnVal = fc.showDialog(this, "Open");                        (3)

switch (returnVal) {                                              (4)
    case JFileChooser.APPROVE_OPTION:                               (5)
        File tekstfile = fc.getSelectedFile();
        BufferedReader buffer = null;

        try {
            FileReader file = new FileReader(tekstfile);
            buffer = new BufferedReader(file);
            boolean eof = false;
            String deVolledigeTekst = "";
            while (!eof) {
                try {
                    String eenLijn = buffer.readLine();
                    if (eenLijn == null)
                        eof = true;
                    else
                        deVolledigeTekst += eenLijn;
                }
                catch (IOException e) {
                    JOptionPane.showMessageDialog(this, e.getMessage(),
                        "Error", JOptionPane.ERROR_MESSAGE);
                }
                jTextAreaFormat.setText(deVolledigeTekst);
                jMenuItemClose.setEnabled(true);
            }
        }
        catch (FileNotFoundException e) {
            JOptionPane.showMessageDialog(this, e.getMessage(), "Error",
                JOptionPane.ERROR_MESSAGE);
        }
        catch (IOException e) {
            JOptionPane.showMessageDialog(this, e.getMessage(), "Error",
                JOptionPane.ERROR_MESSAGE);
        }
        finally {
            if (buffer != null)
                try { buffer.close(); }
            catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```



```

    }
    break;

    case JFileChooser.CANCEL_OPTION:                                (6)
        break;

    case JFileChooser.ERROR_OPTION:                                (7)
        JOptionPane.showMessageDialog(this, "Error opening file...",
            "Error", JOptionPane.ERROR_MESSAGE);
}

```

- (1) We maken een nieuw `JFileChooser` object aan. Dit object bevat alle nodige keuzelijsten e.d. om een file te kiezen in een filesysteem.
- (2) We stellen de titel van de `JFileChooser` in.
- (3) We tonen het dialoogvenster met de method `showDialog`. De parameters zijn het parentobject (het `JFrame`) en het opschrift van de acceptbutton.
- (4) Afhankelijk van het resultaat van de method `showDialog` verwerken we een file, of we doen helemaal niets (er werd op cancel gedrukt) of we tonen een foutboodschap.
- (5) Een geldige file werd gekozen en dus lezen we deze in. Het resultaat stoppen we in de `JTextArea`. Het menu-item Close mag nu wel toegankelijk zijn.
- (6) Als de gebruiker op Cancel klikt dan hoeft er niets te gebeuren.
- (7) Bij een fout tonen we de foutboodschap in een `messagedialog`.

Het menu-item Close nu. Wanneer de gebruiker Close kiest moet de `JTextArea` gewist worden en mag het menu-item opnieuw disabled worden :

```

private void jMenuItemCloseActionPerformed(
    java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    JTextAreaFormat.setText("");
    jMenuItemClose.setEnabled(false);
}

```

Je voegt deze bovenstaande code toe door rechts te klikken op het menu-item en te kiezen voor `Events-Action-actionPerformed`.

De code voor het laatste menu-item van het menu File is vrij eenvoudig. Via `System.exit(0)` verlaten we het programma :

```

private void jMenuItemExitActionPerformed(
    java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}

```

```
System.exit(0);  
}
```

Je kan het programma alvast eens uitproberen.

Volgende menu is het menu Format. Bedoeling is de tekst in de JTextArea vetjes en cursief te zetten met de bovenste twee menu-items. Wanneer de tekst vet is moet er een vinkje vooraan het menu-item Bold komen. Idem voor het menu-item Italic wanneer de tekst cursief gezet is.

De code :

```
private void jCheckBoxMenuItemBoldActionPerformed(  
    java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    Font f = JTextAreaFormat.getFont();           (1)  
    int style = f.getStyle();                      (2)  
    if (f.isBold()) {  
        style -= Font.BOLD;                       (3)  
    } else {  
        style += Font.BOLD;                       (4)  
    }  
    Font newFont = f.deriveFont(style);           (5)  
    JTextAreaFormat.setFont(newFont);             (6)  
}  
  
private void jCheckBoxMenuItemItalicActionPerformed(  
    java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    Font f = JTextAreaFormat.getFont() ;  
    int style = f.getStyle() ;  
    if (f.isItalic()) {  
        style -= Font.ITALIC;  
    } else {  
        style += Font.ITALIC ;  
    }  
    Font newFont = f.deriveFont(style) ;  
    JTextAreaFormat.setFont(newFont);             (7)  
}
```

- (1) We halen eerst het font van de JTextArea op.
- (2) De eigenschappen bold en italic zitten in de style van het font.
- (3) Is de tekst reeds vet dan zetten we de bold-bit terug uit.
- (4) Is de tekst nog niet vet dan zetten we de bold-bit aan.
- (5) We leiden een nieuw font af van de bestaande font met de gewijzigde style-instellingen.



(6) En we stellen het font van de `TextArea` terug in.

(7) Op een analoge wijze maken we de tekst wel of niet cursief.

Met de volgende twee menu-items `SetColor` en `SetBackColor` stellen we de voor- en achtergrondkleur van de `TextArea` in. Om dit te doen gebruiken we een `JColorChooser` dialoogvenster :

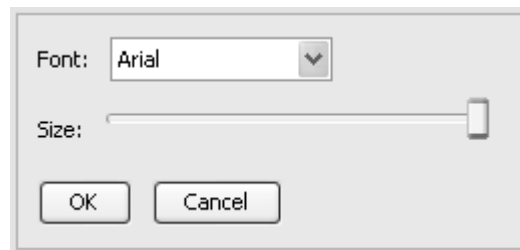
```
private void jMenuItemColorActionPerformed(  
    java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    Color newColor = JColorChooser.showDialog(  
        this, "Choose Foreground Color",  
        JTextAreaFormat.getForeground());  
    if (newColor != null) {  
        JTextAreaFormat.setForeground(newColor);  
    }  
}  
  
private void jMenuItemBackColorActionPerformed(  
    java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    Color newColor = JColorChooser.showDialog(  
        this, "Choose Background Color",  
        JTextAreaFormat.getBackground());  
    if (newColor != null) {  
        JTextAreaFormat.setBackground(newColor);  
    }  
}
```

- (1) We tonen een colorchooser-dialog met de method `showDialog`. Als parameters geven we het parent-object mee, een titel en via de method `getForeground()`, de huidige kleur van de `TextArea`.
- (2) Als er een kleur gekozen is dan stellen we deze met de method `setForeground()`.
- (3) Op een analoge wijze tonen we een dialoogvenster om de achtergrondkleur in te stellen...
- (4) ...en passen we de gekozen kleur toe.

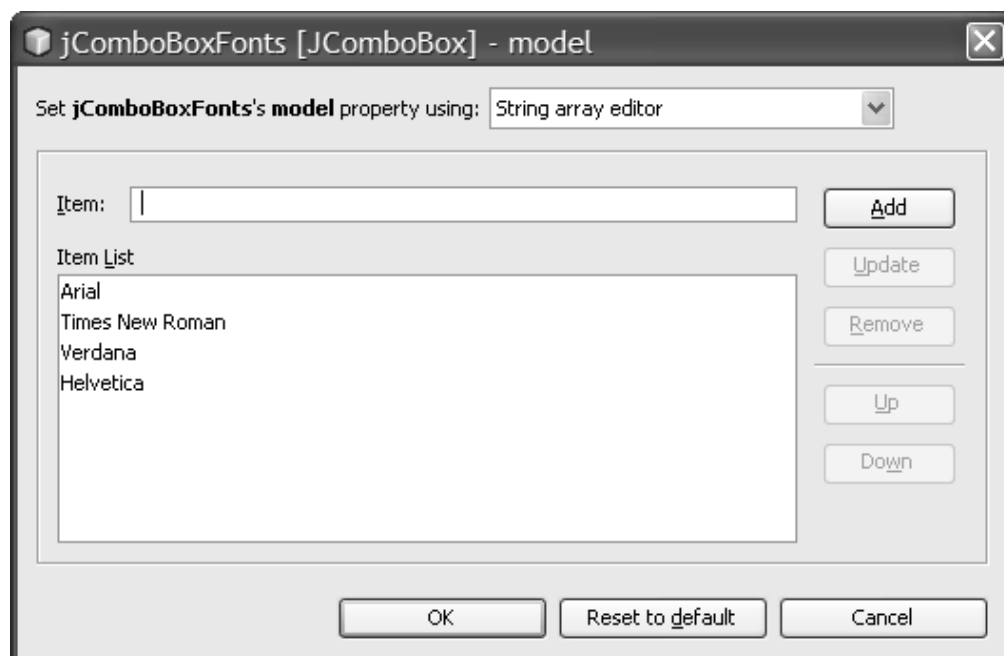
Als laatste menuitem van het menu `Format` hebben we het menuitem `Font`. Het is de bedoeling dat we via een dialoogvenster het lettertype instellen a.d.h.v. een keuzelijst en de lettergrootte met een slider-control.

We ontwerpen eerst het dialoogvenster. Sleep vanaf het `Palette` een `Dialog` naar het frame. Je ziet deze in het `Inspector`-venster verschijnen onder `Other Components`. Geef de dialog de naam `jDialogFont`.

Sleep op deze dialog een label met het opschrift “Font:” en ernaast een Combo Box. Eronder komt een tweede label met opschrift “Size:” en een Slider ernaast. Daaronder sleep je twee buttons met opschrift OK en Cancel.



De combobox, de slider en de twee buttons geef je best een betere naam. We geven de combobox een aantal mogelijke waarden door in de properties te klikken op de drie puntjes naast de property *model*. We krijgen dan volgend dialoogvenster te zien :



In dit dialoogvenster voeg je een aantal lettertypes toe zoals hierboven weergegeven. Daarna klik je op OK.

Van de slider stel je de eigenschappen minimum in op 6 en maximum op 32. De eigenschap *minorTickSpacing* zet je op 1 en *majorTickSpacing* op 6.

Om de font-dialog te laten verschijnen bij het kiezen van het menu-item Format-Font, voeg je volgende code toe:

```
private void jMenuItemFontActionPerformed(  
    java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    jSliderFontSize.setValue(jTextAreaFormat.getFont().getSize()); (1)  
    jComboBoxFonts.setSelectedItem(  
        jTextAreaFormat.getFont().getFamily(); (2)  
    jDialogFont.pack(); (3)
```



```
jDialogFont.setVisible(true); (4)
}
```

- (1) We stellen de slider in op een initiële waarde, gelijk aan de lettergrootte van de textarea.
- (2) Ook de combobox geven we een beginwaarde, namelijk het lettertype van de textarea.
- (3) We geven de dialog de nodige grootte om alle controls te kunnen weergeven...
- (4) ...en we maken de dialog zichtbaar.

Er rest ons nu nog de nodige code toe te voegen aan de OK- en aan de Cancel-knop.

```
private void jButtonOKActionPerformed(
    java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Font f = JTextAreaFormat.getFont(); (1)
    Font nieuwFont = new Font(
        JComboBoxFonts.getSelectedItem().toString(),
        f.getStyle(),
        jSliderFontSize.getValue()); (2)
    JTextAreaFormat.setFont(nieuwFont); (3)
    jDialogFont.setVisible(false); (4)
}

private void jButtonCancelActionPerformed(
    java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    jDialogFont.setVisible(false); (5)
}
```

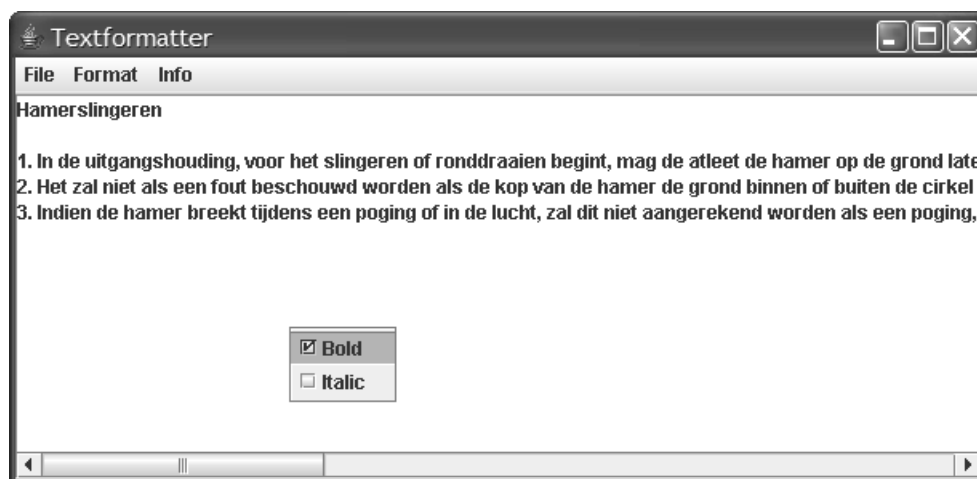
- (1) Als de gebruiker op de OK-knop klikt, stellen we de font van de textarea in met het lettertype en de lettergrootte die de gebruiker gekozen heeft. De overige font-instellingen blijven dezelfde. Dus starten we met het actuele font van de textarea op te halen.
- (2) Dan maken we een nieuw font met het gekozen lettertype, de style van de net opgehaalde font en met de lettergrootte volgens de instelling van de slider.
- (3) We stellen het font van de textarea in met het nieuwe samengestelde font.
- (4) Het dialoogvenster mag terug verdwijnen.
- (5) Wanneer we op Cancel klikken hoeft er helemaal niks ingesteld worden en kan het dialoogvenster meteen verdwijnen.

Met het laatste menu-item Info-About tonen we een informatievenster. De code ziet er als volgt uit :

```
private void jMenuItemAboutActionPerformed(
    java.awt.event.ActionEvent evt) {
    JOptionPane.showMessageDialog(this,
        "Dit is het derde Swing-met-NetBeans voorbeeld.", "About...",
        JOptionPane.INFORMATION_MESSAGE);
}
```

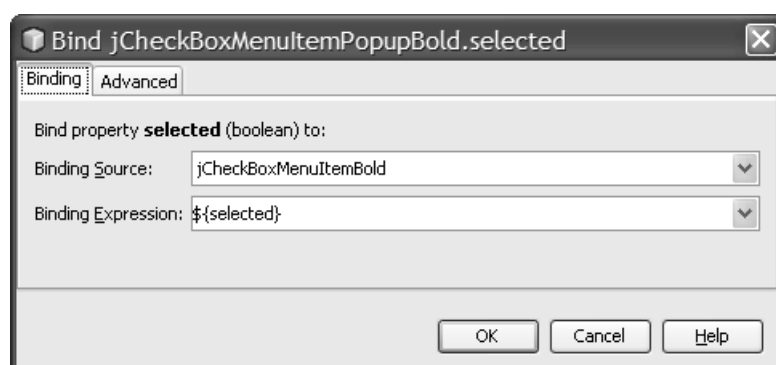
- (1) Deze code gebruiken we al om een eventuele foutboodschap weer te geven bij het inlezen van een file. We tonen een messagedialog van het information-type, in het actuele frame (this), met een bepaalde boodschap en titel.

Om dit hoofdstuk af te sluiten voegen we nog een popupmenu toe aan de toepassing. Als we in de textarea met de rechtermuistoets klikken dan moet er een popupmenu verschijnen waarmee we de tekst vet of cursief kunnen zetten. Dit moet uiteraard gebeuren in harmonie met de menuitems Format-Bold en Format-Italic.



Sleep een Popup Menu vanuit de Palette op de Frame. In het Inspector-venster verschijnt onder Other Components een popupmenu. Geef dit menu een betere naam. Klik in het Inspector-venster met de rechtermuistoets op dit popupmenu en kies Add From Palette – Menu Item / CheckBox. Doe dit nog een tweede keer en geef beide menu-items een betere naam.

Opdat deze twee menu-items in harmonie zouden zijn met de twee menu-items uit het Format-menu moeten we ze aan elkaar koppelen. Klik met de rechtermuistoets op het eerste menu-item uit het popupmenu en kies Bind – selected.





In het dialoogvenster kies je bij Binding Source het menu-item uit het Format-menu waaraan je de selected-property wil koppelen. Eronder kies je bij Binding Expression uiteraard ook de selected-property van het menu-item dat erboven is geselecteerd.

Hetzelfde doe je nu ook voor het tweede menu-item van het popup-menu.

Voorlopig werken de vinkjes synchroon maar gebeurt er nog niets bij het aanklikken van de popupmenu-items. Klik dus in het Inspector-venster met de rechtermuistoets op het popupmenu-item en kies Events – Action – actionPerformed. Aangezien exact dezelfde code moet uitgevoerd worden als bij het Format-menuitem kunnen we volgende code schrijven :

```
private void jCheckBoxMenuItemPopupBoldActionPerformed(  
    java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    jCheckBoxMenuItemBoldActionPerformed(evt);  
}
```

Voeg ook voor het tweede popupmenu-item de gewenste code toe.

Er rest ons nu nog het popupmenu te laten verschijnen wanneer de gebruiker met de rechtermuistoets op de textarea klikt. Klik in Design met de rechtermuistoets op de text area en kies Events – Mouse – mouseClicked. Voeg dan volgende code toe :

```
private void jTextAreaFormatMouseClicked(  
    java.awt.event.MouseEvent evt) {  
    // TODO add your handling code here:  
    if (evt.getButton() == MouseEvent.BUTTON3) { (1)  
        jPopupMenuFormat.show(this,  
            (int) this.getMousePosition().getX(),  
            (int) getMousePosition().getY()); (2)  
    }  
}
```

(1) Als de button die het event veroorzaakte button3 (rechtermuistoets) is...

(2) ...tonen we het popupmenu in het frame (this) en op de positie van de muiscursor.

Het programma is nu compleet en kan uitgevoerd worden.



## Hoofdstuk 16 Bijlagen

---

### 16.1 Het gebruik van de API

#### 16.1.1 Inleiding

Gezien de uitgebreidheid van Java is het onmogelijk om alle details te kennen en te memoriseren. Een java-programmeur is dan afhankelijk van de beschikbaarheid van een goed helpsysteem.

Via de java – sun website krijgt men toegang tot de API (Application Programmer Interface).

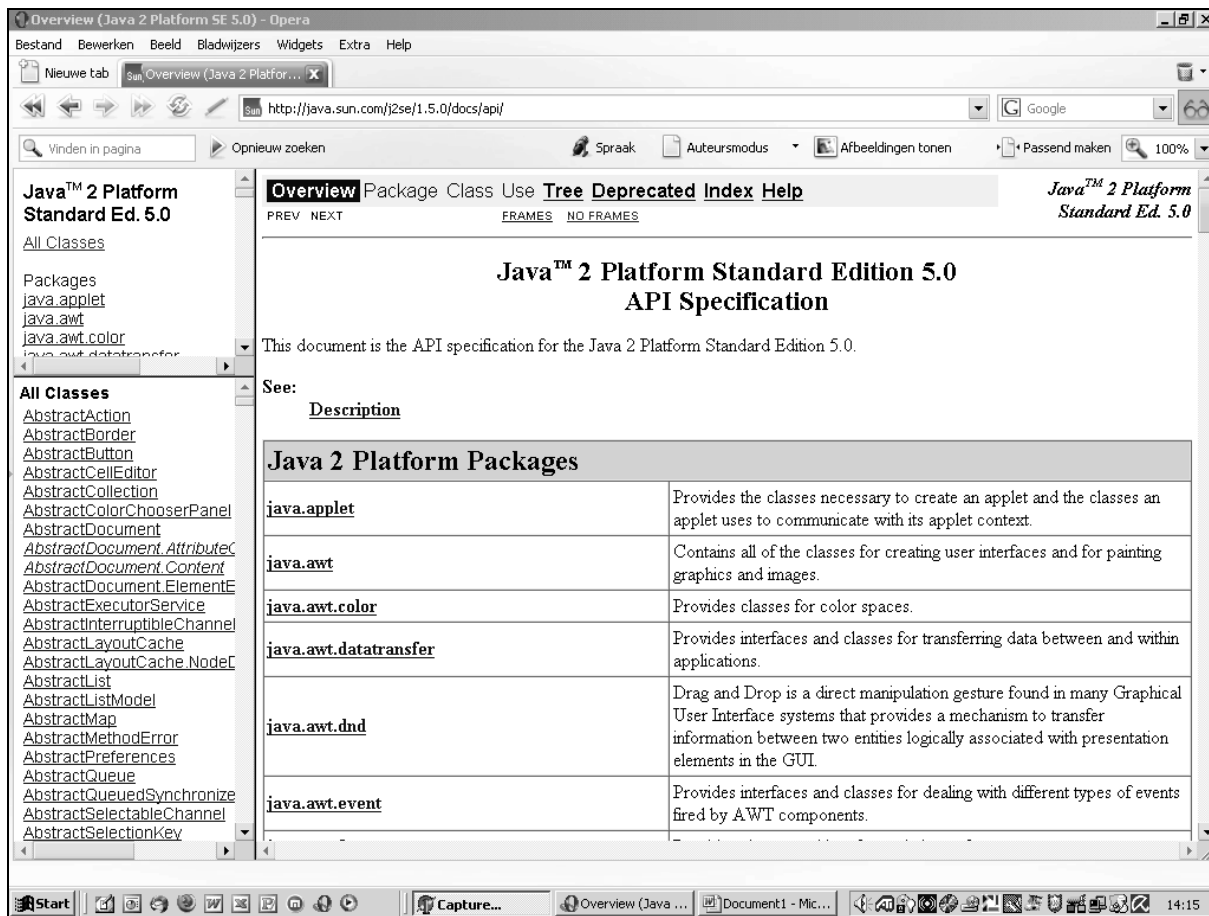


We kiezen de API die bij de cursus hoort **JavaSE** en **J2SE 1.5.0**. De API vind je bij Reference, API & Docs.

De API-documentatie is niet echt een help-systeem, wel een zeer volledig naslag- en opzoeksysteem. Voor een beginner is dit overweldigend. De kunst zal er dan ook in bestaan om selectief te lezen en om die informatie te vinden waar we naar op zoek zijn.

Opmerking:

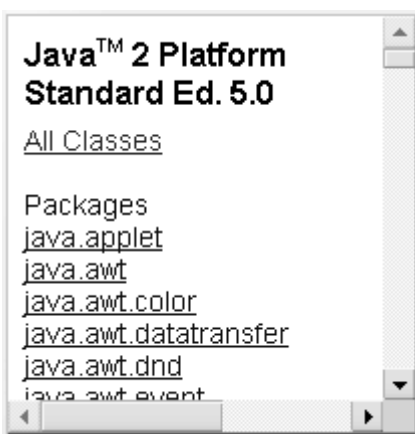
De documentatie kan ook gedownload en lokaal geïnstalleerd worden.



### 16.1.2 Structuur van de API

Het venster is ingedeeld in 3 frames.

Links bovenaan :



Zoals je weet zijn de classes gegroepeerd in **packages**. Weet men in welk package men moet zoeken, dan selecteert men dit package, dit vereenvoudigt de verdere zoekfunctie.

Weet men niet in welk package een bepaalde class zit, dan kiest men voor 'All Classes'.

Links onderaan:



Hier staat een alfabetische lijst van alle classes die horen bij het package dat in het frame links-boven gekozen is. Heeft men gekozen voor 'All classes', dan krijgt men het alfabetisch overzicht van alle classes.

Kijkt men goed toe dan ziet men dat class-namen soms cursief staan: in dit geval gaat het om interfaces.

Het grote frame rechts:

Overview
Package
Class
Use
Tree
Deprecated
Index
Help

PREV
NEXT
FRAMES
NO FRAMES

## Java™ 2 Platform Standard Edition 5.0 API Specification

This document is the API specification for the Java 2 Platform Standard Edition 5.0.

See: [Description](#)

Java 2 Platform Packages	
<a href="#">java.applet</a>	Provides the classes necessary to create applet uses to communicate with its a
<a href="#">java.awt</a>	Contains all of the classes for creating graphics and images.
<a href="#">java.awt.color</a>	Provides classes for color spaces.
<a href="#">java.awt.datatransfer</a>	Provides interfaces and classes for tr applications.
<a href="#">java.awt.dnd</a>	Drag and Drop is a direct manipulation User Interface systems that provides information between two entities logi elements in the GUI.
<a href="#">java.awt.event</a>	Provides interfaces and classes for d fired by AWT components.

Bevat de uitleg die hoort bij de keuze die men links gemaakt heeft. Binnen dit frame zitten nog een paar extra navigatie-mogelijkheden waarvan de Index interessant is. Dit laat namelijk toe om snel te zoeken (op beginletter) binnen de grote verzameling van alle classes. Men kan zowel classes, methods, properties, ... opzoeken via de index.

Bij de start, als men nog geen keuze gemaakt heeft, toont dit frame een beschrijving van de packages.



### 16.1.3 Voorbeeld

We zoeken de beschrijving van de methode `arraycopy`.

Via de index zoeken we deze methode:

**`arraycopy(Object, int, Object, int, int)`** - Static method in class `java.lang.System`  
 Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Via dit opzoeken leren we dat de method **`arraycopy`** een method is van de class **`System`** die op haar beurt behoort tot het package `java.lang`.

We kunnen nu doorklikken op het woordje **`System`** en komen dan in de beschrijving van de ganse class of we kunnen doorklikken op de method **`arraycopy`** zelf en we komen dan rechtstreeks in de beschrijving van de method.

We doen het eerste en klikken op **`System`**.

`java.lang`  
**Class System**

`java.lang.Object`  
 ↳ `java.lang.System`

---

`public final class System`

extends `Object`

The `System` class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the `System` class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

**Since:**  
 JDK 1.0

We zien het hiërarchisch systeem: `System` is een child van `Object`.

Bij de definitie **`public final class System`** leren we dat het een public class is en dus door iedereen mag gebruikt worden en dat ze **`final`** is. Er kunnen geen subclasses van afgeleid worden.

Verder zien we ook nog dat ze tot het package **`java.lang`** behoort, d.w.z. dat er geen import nodig is, `java.lang` is altijd ter beschikking.

Scrollen we verder naar beneden dan vinden we achtereenvolgens :

De field summary (of de properties):

Field Summary	
static <code>PrintStream</code> <code>err</code>	The "standard" error output stream.
static <code>InputStream</code> <code>in</code>	The "standard" input stream.
static <code>PrintStream</code> <code>out</code>	The "standard" output stream.

Van iedere property vinden we het type (printstream, inputstream, ...) de naam (err, in, out) en of het een objectvariabele of een class-variabele is (static).

Doorklikken op de naam van de property geeft meer informatie:

**in**

```
public static final InputStream in
```

The "standard" input stream. This stream is already open and ready to supply input data. Typically this stream corresponds to keyboard input or another input source specified by the host environment or user.

Onder de Field summary vinden we de **Method summary**:

Method Summary	
static void	<u>arraycopy</u> ( <u>Object</u> src, int srcPos, <u>Object</u> dest, int destPos, int length) Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
static <u>String</u>	<u>clearProperty</u> ( <u>String</u> key) Removes the system property indicated by the specified key.
static long	<u>currentTimeMillis</u> () Returns the current time in milliseconds.
static void	<u>exit</u> (int status) Terminates the currently running Java Virtual Machine.
static void	<u>gc</u> () Runs the garbage collector.
static <u>Map</u> < <u>String</u> , <u>String</u> >	<u>getenv</u> () Returns an unmodifiable string map view of the current system environment.
static <u>String</u>	<u>getenv</u> ( <u>String</u> name)

Dit is een opsomming van alle methods die tot deze class (in dit voorbeeld **System**) behoren.

Voor iedere method krijgen we de signatuur:

- De naam
- Het returntype
- De inputparameters: (type en naam)
- Het woord **static** (enkel voor de classmembers)
- Het woord **final** (enkel voor de niet te overriden methods)
- ...

Bovendien krijgen we een korte omschrijving van de functionaliteit.



Doorklikken op de naam levert alle details:

#### **arraycopy**

```
public static void arraycopy(Object src,  
                             int srcPos,  
                             Object dest,  
                             int destPos,  
                             int length)
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array. A subsequence of array components are copied from the source array referenced by `src` to the destination array referenced by `dest`. The number of components copied is equal to the `length` argument. The components at positions `srcPos` through `srcPos+length-1` in the source array are copied into positions `destPos` through `destPos+length-1`, respectively, of the destination array.

If the `src` and `dest` arguments refer to the same array object, then the copying is performed as if the components at positions `srcPos` through `srcPos+length-1` were first copied to a temporary array with `length` components and then the contents of the temporary array were copied into positions `destPos` through `destPos+length-1` of the destination array.

If `dest` is null, then a `NullPointerException` is thrown.

If `src` is null, then a `NullPointerException` is thrown and the destination array is not modified.

Otherwise, if any of the following is true, an `ArrayStoreException` is thrown and the destination is not modified:

- The `src` argument refers to an object that is not an array.
- The `dest` argument refers to an object that is not an array.
- The `src` argument and `dest` argument refer to arrays whose component types are different primitive types.
- The `src` argument refers to an array with a primitive component type and the `dest` argument refers to an array with a reference component type.
- The `src` argument refers to an array with a reference component type and the `dest` argument refers to an array

In de beschrijving vinden we alle informatie over de method `arraycopy` !

Een belangrijk onderdeel van de beschrijving vinden we onderaan: kunnen er fouten optreden bij het uitvoeren van deze method en zo ja welke?

#### **Throws:**

`IndexOutOfBoundsException` - if copying would cause access of data outside array bounds.

`ArrayStoreException` - if an element in the `src` array could not be stored into the `dest` array because of a type mismatch.

`NullPointerException` - if either `src` or `dest` is null.

## 16.2 Debuggen

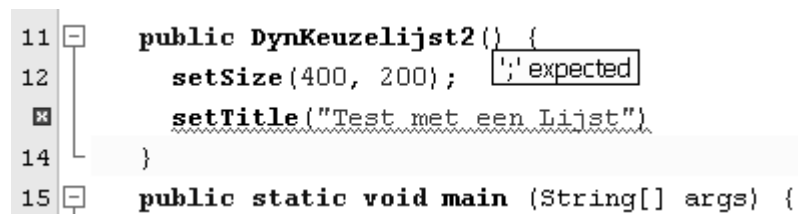
Bij het programmeren maak je wel eens een fout. De ontwikkelomgeving die je gebruikt biedt je meestal een aantal hulpmiddelen om fouten op te sporen. In deze bijlage concentreren we ons op een aantal debugtools die netbeans ons biedt.

### 16.2.1 Soorten fouten

Afhankelijk van het type fout dat je maakt, zijn er andere hulpmiddelen ter beschikking om ze op te lossen. Het is daarom belangrijk onderscheid te maken tussen de soorten fouten:

- **Programmeerfouten**

Zijn een gevolg van onjuiste programmacode of syntaxisfouten. Je hebt misschien een sleutelwoord verkeerd getypt of je bent één of meerdere accolades vergeten. Dergelijke fouten worden direct aangeduid door netbeans met een rode, gekartelde onderlijning en een kruisje in de kantlijn. Als je de muiscursor laat rusten op de foute code dan krijg je een hint over de fout.



```
11 public DynKeuzelijst2() {  
12     setSize(400, 200);  
13     setTitle('Test met een lijst');  
14 }  
15 public static void main (String[] args) {
```

- **Run-time fouten**

Gebeuren tijdens de uitvoering als je programma een instructie uitvoert die onmogelijk is. Alhoewel de syntax van je code correct is, komt de fout pas tot uiting als het programma uitgevoerd wordt. Je kan instructies inbouwen die de fout opvangen (zie Exception handling).

- **Logische fouten**

Alhoewel de syntaxis correct is en er geen run-time fouten optreden, voert het programma toch onjuiste bewerkingen uit. Enkel door het programma te testen kan je ontdekken waar de fout zit.

### 16.2.2 Het programma stap voor stap uitvoeren

Als je niet precies weet waar de code de mist in gaat kan je de code stap voor stap laten uitvoeren. In NetBeans kies je Debug – Step Into (F7) en het programma houdt halt bij de eerste regel code. Bij een nieuwe druk op de toets F7 spring je naar de volgende coderegel. Wordt er een method opgeroepen dan spring je naar de eerste regel van deze method. Wil je dit niet en heb je liever dat de opgeroepen method meteen volledig wordt uitgevoerd dan kies je beter F8 (Step Over). Met Step Out (Ctrl-F7) spring je uit een method terug naar de plaats waar de method-oproep staat. Wil je terug naar een normale uitvoering van alle coderegels, kies dan F5.



### 16.2.3 BreakPoints

Als je min of meer weet waar de fout zich ergens voordoet dan kan je in de buurt een breakpoint zetten. Dit doe je door te klikken in kantlijn. Er verschijnt dan een roze vierkantje en de coderegel krijgt een roze achtergrondkleur. Als je nu het programma start met F5, dan wordt het programma uitgevoerd tot aan de breakpoint. De verdere regels kan je dan met F7 of F8 verder uitvoeren. Je verwijdert een breakpoint door in de marge het roze vierkantje terug aan te klikken.

Een beetje gelijkaardig aan het werken met breakpoints is Run to Cursor (F4). Het programma wordt normaal uitgevoerd tot een de plaats waar de cursor staat in de code. Je kan dan met F7 of F8 de code opnieuw regel per regel uitvoeren.

### 16.2.4 Informatie verzamelen

Met bovenstaande mogelijkheden alleen zijn we nog niet veel. We kunnen bijvoorbeeld wel al achterhalen op welke regel een programma crasht maar meer informatie hebben we nog niet.

Door de cursor te laten rusten op een variabele krijgen we meer info via een infotip :

```
public static void main (String[] args) {
    int test = tekst = (java.lang.String) "test"
    String tekst = "test";
}
```

Verder beschikken we in netbeans ook over diverse vensters zoals o.a. 'Watches', 'Local Variables' en 'Call stack'. In het venster Local Variables vind je een lijst met alle variabelen die op dat moment in gebruik zijn, met hun inhoud.

Watches		Local Variables		BPEL P
Name	Value	Type		
args	#108(length=0)	String[]		
frame	#448	DynKeuzelijst2		
tekst	"test"	String		
test	4	int		

In een ander venster, Watches, kan je een bepaalde expressie evalueren door het verloop van het programma heen.

Watches		Local Variables		BPE
Name	Type	Value		
test < 3	boolean	false		





## Colofon

---

Sectorverantwoordelijke	Ortaire Uyttersprot
Cursusverantwoordelijke	Jean Smits
Medewerkers	Steven Lucas Brigitte Loenders Hans De Smet
Versie	1-10-2013